Lab 2 - Computational Statistics (732A90)

Julius Kittler (julki092), Vinay Bengaluru Ashwath Narayan Murthy (vinbe289)

February 7, 2018

Contents

1 Assignment 1: Optimizing a model parameter

1.1 Task 1: Data Import and Partitioning

First, we import the data into R, add one more variable `LMR` to the data which is the natural logarithm of Rate and split the data into train and test set.

1.2 Task 2: Creating myMSE function for LOESS model

Then, we create the function 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.

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

  cnt <<- cnt + 1 # to increase global counter

  res = loess(Y ~ ., data = cbind(Y = pars$Y, pars$X),
              enp.target = lambda)
```

```
  pred = predict(res, newdata = cbind(Y = pars$Ytest, pars$Xtest))

  SSE = sum((pars$Ytest - pred)^2)
  MSE = SSE / length(pars$Ytest)
  return(MSE)
}
```

1.3 Task 3: Brute force (running myMSE for various lambda values)

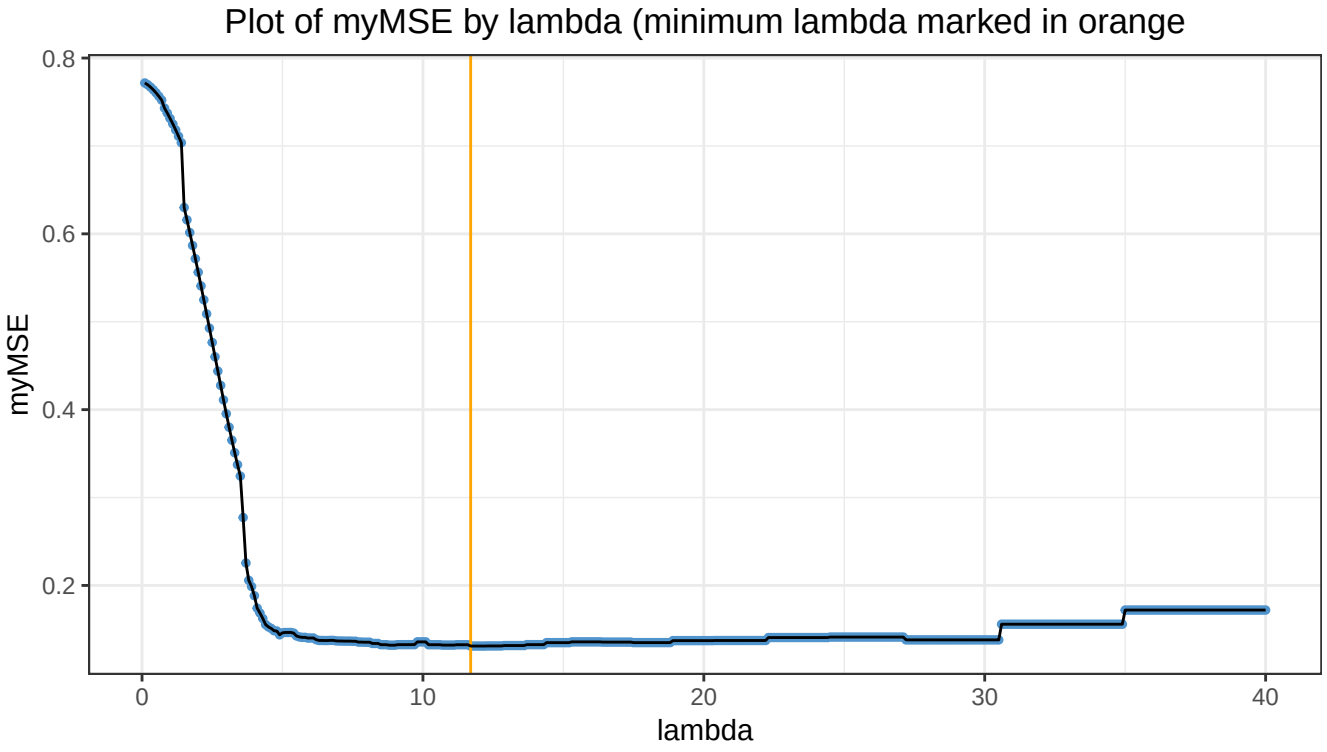
Below, the function `myMSE()` and the training and test data sets with response `LMR` and predictor `Day` are used to estimate the predictive MSE values for the λ values of 0.1,0.2,...,40.

```
cnt = 0
lambdas = as.list(seq(0.1, 40, 0.1))

params = list(Y = train$LMR, X = as.data.frame(as.matrix(train$Day)),
              Ytest = test$LMR, Xtest = as.data.frame(as.matrix(test$Day)))

res = unlist(lapply(lambdas, function(x, params) myMSE(x, params),
                    params = params))
```

1.4 Task 4: Brute force (visualization)



Optimal lambda:

The optimal value of lambda is printed below and its position is also marked with a vertical orange line in the plot.

```
## Best lambda:  11.7
```

Number of evaluations:

```
## Number of myMSE() calls:  400
```

In total, 400 evaluations were conducted to find the minimum lambda. A counter (using the global variable `cnt` was used to identify this number) but one can also just calculate it here like so $40/0.1 = 400$.

One might have considered stopping earlier, e.g. when the value of lambda does not change significantly anymore (somewhere after the minimum). However, this kind of a strategy should be considered thoroughly to avoid getting stuck in a local minimum (when better lambda values could be found).

1.5 Task 5: Using optimize() to find best lambda

Here, the `optimize()` function is used for the same purpose (finding the optimal lambda). Again, the range [0.1,40] is used. Also, the accuracy parameter `tol` is set to 0.01.

Results:

As printed out below, the minimum MSE (of ca. 0.13) is found at a lambda value of ca. 10.7. This lambda value is close to the lambda value of ca. 11.7 from the previous task. Looking at the plot from the previous task, we can see that myMSE is very, very similar for both values of lambda, 10.7 and 11.7. If one does not consider the very last decimals, one can conclude that the `optimize()` function was able to find the optimal MSE value as well.

```
res = optimize(myMSE, interval = c(0.1, 40), tol = 0.01)
```

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

```
## Number of myMSE() calls:  18
```

Number of evaluations:

In total, 18 evaluations of `myMSE()` were conducted to find the minimum lambda. A counter (using the global variable `cnt` was used to identify this number). Overall, we seem to a achieve a good result much, much faster than with the brute force approach from the previous task (where we needed 400 function evaluations).

1.6 Task 6: Using optim() to find best lambda

Here, we use the `optim()` function and BFGS method with starting point $\lambda = 35$ to find the optimal λ value.

Number of evaluations:

We see in the printout below that the function was called only 1 time.

Comparison of results with step 5:

The function converges directly at the initial value. Since convergence happens when the gradient is zero, the reason for the direct conversion might be that the gradient is estimated to be zero. Note that because we don't provide the gradient function with the parameter `gr`, a "finite-difference approximation" is used according to the documentation.

There are two possible explanations for why the gradient is estimated to be zero: either the `optim` only looks to the right of the lambda value 35 or it looks in both directions but at very very small intervals. E.g. `myMSE(35) = 0.17` but also `myMSE(34.975) = 0.17`, i.e. if we decrease lambda by a very small value (here 0.025), we still get the same result.

```
## $par
## [1] 35
##
## $value
## [1] 0.1719996
##
## $counts
## function gradient
##        1        1
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL"
```

2 Assignment 2: Maximizing likelihood

2.1 Task 1: Data Import

First, we load the data into R.

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

2.2 Task 2: Estimating μ and σ with MLE

Log-likelihood function

The likelihood function is the following (where n = 100):

$$\begin{aligned} L(\theta) &= L(y_1, y_2, ..., y_n|\theta) \\ &= \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}\cdot(y_i-\mu)^2} \\ &= \frac{1}{\sigma^n(\sqrt{2\pi})^n} e^{-\frac{1}{2\sigma^2}\cdot\sum_{i=1}^{n}(y_i-\mu)^2} \end{aligned}$$

The log-likelihood function corresponding to this is the following:

$$ln(L(\theta)) = -\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i-\mu)^2 - n\cdot ln(\sigma) - n\cdot ln(\sqrt{2\pi})$$

MLE for μ

To find the MLE for μ, we take the derivative w.r.t. μ. Note that we apply the chain rule and we discard everything from $-n$ onwards because there is no μ present.

$$\begin{aligned} \frac{\partial \ln(L(\theta))}{\partial\mu} &= \frac{\partial}{\partial\mu}\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i-\mu)^2\right) \\ &= 2\cdot(-1)\cdot\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i-\mu)\right) \\ &= \frac{1}{\sigma^2}\sum_{i=1}^{n}(y_i-\mu) \end{aligned}$$

Now, since we want to maximize the log-likelihood w.r.t. μ, we set the partial derivative w.r.t. μ equal to zero and solve for μ.

$$
\begin{aligned}
&\frac{1}{\sigma^2}\sum_{i=1}^{n}(y_i-\mu)=0\\
&\sum_{i=1}^{n}(y_i-\mu)=0\\
&\sum_{i=1}^{n}y_i-n\mu=0\\
&\sum_{i=1}^{n}y_i=n\mu\\
&\mu=\frac{1}{n}\sum_{i=1}^{n}y_i\\
&\mu_{MLE}=\bar{y}
\end{aligned}
$$

In other words, the MLE estimator for μ is the sample mean.

MLE for σ

To find the MLE for σ, we take the derivative w.r.t. σ.

$$
\begin{aligned}
\frac{\partial \ln(L(\theta))}{\partial\sigma} &= \frac{\partial}{\partial\sigma}\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i-\mu)^2-n\cdot ln(\sigma)\right)\\
&= -2\left(-\frac{1}{2\sigma^3}\sum_{i=1}^{n}(y_i-\mu)^2\right)-\frac{n}{\sigma}\\
&= \frac{1}{\sigma^3}\sum_{i=1}^{n}(y_i-\mu)^2-\frac{n}{\sigma}
\end{aligned}
$$

Now, since we want to maximize the log-likelihood w.r.t. σ, we set the partial derivative w.r.t. σ equal to zero and solve for σ.

$$
\begin{aligned}
&\frac{1}{\sigma^3}\sum_{i=1}^{n}(y_i-\mu)^2-\frac{n}{\sigma}=0\\
&\frac{1}{\sigma^3}\sum_{i=1}^{n}(y_i-\mu)^2=\frac{n}{\sigma}\\
&\frac{1}{n}\sum_{i=1}^{n}(y_i-\mu)^2=\sigma^2\\
&\sigma_{MLE}=\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i-\mu)^2}
\end{aligned}
$$

In other words, the MLE estimator for σ is the square root of the sample variance.

Now, we implement the sample mean and the square root of the sample variance as functions in R:

```
mle_mu = function(x = x){

  n = length(x)
  return(sum(x)/n)

}

mle_sigma = function(x = x){

  n = length(x)
  mu = mean(x)
  return(sqrt(sum((x - mu)^2)/n))

}
```

Using these functions, we obtain the following results:

```
## mle_mu(x):  1.275528
```

```
## mle_sigma(x):  2.005976
```

2.3 Task 3: BFGS and CG (with and without gradient specified)

Here we optimize the minus log–likelihood function with initial parameters $\mu = 0$ and $\sigma = 1$. Both, conjugate Gradient method and BFGS algorithm with gradient specified and without are used.

The general negative log-likelihood function looks as follows. Since we went to maximize the log-likelihood, the negative log-likelihood needs to be minimized.

$$-ln(L(\theta)) = \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mu)^2 + n \cdot ln(\sigma) + n \cdot ln(\sqrt{2\pi})$$

However, we actually only need one function since `optim` can optimize two parameters at the same time. Their initial values are passed as a vector to the `par` argument. This vector is represented by θ in the equation above. In our case, we let the vector be `c(0, 1)` where the first element reprents the μ and the second element represents the σ. The corresponding function is implemented below.

```
neg_llh = function(params, x = data){

  mu = params[1]
  sigma = params[2]
```

```
  n = length(x)

  # return(0.5 * n * log(2 * pi * (sigma^2)) + sum(((x - mu)^2 / (2 * sigma^2))))
  return(1/(2*sigma^2)*sum((x - mu)^2) + n*log(sigma) + n*log(sqrt(2*pi)))

}
```

The results without a gradient specified are printed out below.

Results optim() with Conjugate Gradient (gradient NOT specified):

```
## Conjugate Gradient (gradient not specified)--------------------------

## $par
## [1] 1.275528 2.005977
##
## $value
## [1] 211.5069
##
## $counts
## function gradient
##      133       25
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Results optim() with BFGS (gradient NOT specified):

```
## BFGS (gradient not specified)----------------------------------------

## $par
## [1] 1.275528 2.005977
##
## $value
## [1] 211.5069
##
## $counts
## function gradient
##       37       15
##
## $convergence
## [1] 0
##
## $message
## NULL
```

In the following, we consider the same procedure. However, we implement the gradient function and pass it to the `gr` argument of the `optimze` function.

The gradient of the negative log likelihood can be computed as follows:

$$\nabla(-ln(L(\theta))) = \left(-\frac{\partial \ln(L(\theta))}{\partial \mu}, -\frac{\partial \ln(L(\theta))}{\partial \sigma}\right)$$

The first and the second element partial derivatives respectively are computed as follows:

$$\begin{aligned} -\frac{\partial \ln(L(\theta))}{\partial \mu} &= -\frac{1}{\sigma^2}\sum_{i=1}^{n}(y_i - \mu) \\ -\frac{\partial \ln(L(\theta))}{\partial \sigma} &= -\frac{1}{\sigma^3}\sum_{i=1}^{n}(y_i - \mu)^2 + \frac{n}{\sigma} \end{aligned}$$

The corresponding function is implemented below.

```
gradient = function(params, x = data){

  mu = params[1]
  sigma = params[2]
  n = length(x)

  partial_mu = -1/(sigma^2) * sum(x - mu)
  partial_sigma = -1/(sigma^3) * sum((x - mu)^2) + n / sigma
  gradient = c(partial_mu, partial_sigma)

  return(gradient)

}
```

Results optim() with Conjugate Gradient (gradient specified):

```
## Conjugate Gradient (WITH gradient)-------------------------
```

Results optim() with BFGS (gradient specified):

```
## BFGS (WITH gradient)---------------------------------------
## $par
## [1] 1.275528 2.005977
##
## $value
## [1] 211.5069
##
## $counts
## function gradient
##       40       15
```

```
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Bad idea to maximize likelihood rather than maximizing log–likelihood

Generally, the log-likelihood is used because it allows to simplify the formula. We can express it as a sum instead of as a product. If we were to use the likelihood, we might encounter very large values (that R may not be able to handle). One obvious example is the poisson distribution which has the factorial of Y in its denominator. If the mean of Y was at ca. 1500, we would have very, very large values in the denominator `factorial(1500) = Inf` in R. By taking the log of the liklihood, this factorial disappears and turns into a sum.

Using the log-likelihood, it is also easier for us to compute the partial derivative w.r.t. the parameters in order to find the maximum likelihood estimators.

2.4 Task 4: Conclusions

Solver	Gradient	Convergence	Mu	Sigma	Lllh	Function_Calls	Gradient_Calls
CG	FALSE	0	1.275528	2.005977	211.5069	133	25
BFGS	FALSE	0	1.275527	2.005977	211.5069	37	15
CG	TRUE	0	1.275528	2.005977	211.5069	53	17
BFGS	TRUE	0	1.275528	2.005977	211.5069	40	15

Convergence of algorithms:

All algorithms did converge since the printouts above show a value of `0` under `convergence`.

Optimal parameter values:

The optimal parameter values are printed out above under `par`. The first value represents the optimal μ and the second value represents the optimal σ. The values all resemble the MLE estimates from task 3 very strongly which is good.

Number of function and gradient evaluations:

The number of gradient and function evaluations are printed out above under `counts`. Overall, it seems like BFGS requires fewer evaluations than the Conjugate Gradient method.

Recommended settings:

BFGS vs. Conjugate Gradient method

The BFGS method should be used for large scale problems because each iteration is quicker and no numeric inversion is required (see lecture slides).

The Conjugate Gradient method is slower than Quasi-Newton methods (BFGS belongs to this class) but requires significantly less memory (see lecture slides).

In summary, we might prefer BFGS when we care about time complexity more and the Conjugate Gradient method when we care about memory requireents more.

Passing vs. not passing gradient function

Note that passing the gradient function to `optim` seems to speed up the process for the Conjugate Gradient method whereas it slows down the process for BFGS (measured the number of function and gradient evaluations). The optimal parameters do not seem affected by specifying (or not specifying) the gradient. Therefore, we might consider providing the gradient function when we use the Conjugate Gradient method and not providing it when we use the BFGS method.

3 Appendix

```
# Set up general options

knitr::opts_chunk$set(echo = FALSE, message = FALSE, warning = FALSE,
                      cache = TRUE, cache.path = "cache/", fig.path = "cache/")

# options(digits=22)
options(scipen=999)

library(dplyr)
library(ggplot2)
library(magrittr)

set.seed(12345)


# ------------------------------------------------------------------------------
# Assignment 1, Task 1
# ------------------------------------------------------------------------------

df = read.csv2("mortality_rate.csv")
df$LMR = log(df$Rate)

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


# ------------------------------------------------------------------------------
```

```r
# Assignment 1, Task 2
# ------------------------------------------------------------------------


myMSE = function(lambda, pars = params){

  cnt <<- cnt + 1 # to increase global counter

  res = loess(Y ~ ., data = cbind(Y = pars$Y, pars$X),
              enp.target = lambda)
  pred = predict(res, newdata = cbind(Y = pars$Ytest, pars$Xtest))

  SSE = sum((pars$Ytest - pred)^2)
  MSE = SSE / length(pars$Ytest)
  return(MSE)
}


# ------------------------------------------------------------------------
# Assignment 1, Task 3
# ------------------------------------------------------------------------


cnt = 0
lambdas = as.list(seq(0.1, 40, 0.1))

params = list(Y = train$LMR, X = as.data.frame(as.matrix(train$Day)),
              Ytest = test$LMR, Xtest = as.data.frame(as.matrix(test$Day)))

res = unlist(lapply(lambdas, function(x, params) myMSE(x, params),
                    params = params))


# ------------------------------------------------------------------------
# Assignment 1, Task 4
# ------------------------------------------------------------------------

df_plot = data.frame(lambda = unlist(lambdas), myMSE = res)
min.lambda = round(df_plot$lambda[which.min(df_plot$myMSE)], 4)

ggplot(df_plot, aes(x = lambda, y = myMSE)) +
  geom_point(size = 1, color = "steelblue3") +
  geom_vline(xintercept = min.lambda, color = "orange") +
  geom_line() + theme_bw() + theme(plot.title = element_text(hjust = 0.5)) +
  labs(title = "Plot of myMSE by lambda (minimum lambda marked in orange")
```

```r
cat("Best lambda: ", min.lambda, "\n\n")


cat("Number of myMSE() calls: ", cnt, "\n")


# ------------------------------------------------------------------------------
# Assignment 1, Task 5
# ------------------------------------------------------------------------------

cnt = 0


res = optimize(myMSE, interval = c(0.1, 40), tol = 0.01)


res
cat("Number of myMSE() calls: ", cnt, "\n")


# ------------------------------------------------------------------------------
# Assignment 1, Task 6
# ------------------------------------------------------------------------------

res = optim(par = 35, fn = myMSE, method = "BFGS", lower = 0.1, upper = 40
            #control = list(maxit = 20000, temp = 20, parscale = 20)
            )

res


# ------------------------------------------------------------------------------
# Assignment 2, Task 1
# ------------------------------------------------------------------------------


load("data.RData")
x = data


# ------------------------------------------------------------------------------
# Assignment 2, Task 2
# ------------------------------------------------------------------------------


mle_mu = function(x = x){
```

```r
  n = length(x)
  return(sum(x)/n)

}

mle_sigma = function(x = x){

  n = length(x)
  mu = mean(x)
  return(sqrt(sum((x - mu)^2)/n))

}


cat("mle_mu(x): ", mle_mu(x), "\n")
cat("mle_sigma(x): ", mle_sigma(x), "\n")


# ------------------------------------------------------------------------------
# Assignment 2, Task 3
# ------------------------------------------------------------------------------


neg_llh = function(params, x = data){

  mu = params[1]
  sigma = params[2]
  n = length(x)

  # return(0.5 * n * log(2 * pi * (sigma^2)) + sum(((x - mu)^2 / (2 * sigma^2))))
  return(1/(2*sigma^2)*sum((x - mu)^2) + n*log(sigma) + n*log(sqrt(2*pi)))

}


# Conjugate Gradient method ----------------------------------------------------

cat("Conjugate Gradient (gradient not specified)-------------------------\n\n")
CG_NoGradient = optim(par = c(0, 1), fn = neg_llh, method = "CG")
CG_NoGradient


# BFGS -------------------------------------------------------------------------

cat("BFGS (gradient not specified)---------------------------------------\n\n")
BFGS_NoGradient = optim(par = c(0, 1), fn = neg_llh, method = "BFGS")
```

```r
BFGS_NoGradient



gradient = function(params, x = data){

  mu = params[1]
  sigma = params[2]
  n = length(x)

  partial_mu = -1/(sigma^2) * sum(x - mu)
  partial_sigma = -1/(sigma^3) * sum((x - mu)^2) + n / sigma
  gradient = c(partial_mu, partial_sigma)

  return(gradient)

}



# Conjugate Gradient method ----------------------------------------------------

cat("Conjugate Gradient (WITH gradient)-------------------------\n\n")
CG_Gradient = optim(par = c(0, 1), fn = neg_llh, method = "CG", gr = gradient)



# BFGS -------------------------------------------------------------------------

cat("BFGS (WITH gradient)---------------------------------------\n\n")
BFGS_Gradient=optim(par = c(0, 1), fn = neg_llh, method = "BFGS", gr = gradient)
BFGS_Gradient


# Overview ---------------------------------------------------------------------

# Overview of number mu, sigma, log-likelihood, number of calls
df = data.frame(Solver = c("CG", "BFGS", "CG", "BFGS"),
                Gradient = c(FALSE, FALSE, TRUE, TRUE),
                Convergence = c(CG_NoGradient$convergence,
                                BFGS_NoGradient$convergence,
                                CG_Gradient$convergence,
                                BFGS_Gradient$convergence),
                Mu = c(CG_NoGradient$par[1],
                                BFGS_NoGradient$par[1],
                                CG_Gradient$par[1],
```

```
                                BFGS_Gradient$par[1]),
                Sigma = c(CG_NoGradient$par[2],
                                BFGS_NoGradient$par[2],
                                CG_Gradient$par[2],
                                BFGS_Gradient$par[2]),
                Lllh = c(CG_NoGradient$value,
                                BFGS_NoGradient$value,
                                CG_Gradient$value,
                                BFGS_Gradient$value),
                Function_Calls = c(CG_NoGradient$counts[1],
                                BFGS_NoGradient$counts[1],
                                CG_Gradient$counts[1],
                                BFGS_Gradient$counts[1]),
                Gradient_Calls = c(CG_NoGradient$counts[2],
                                BFGS_NoGradient$counts[2],
                                CG_Gradient$counts[2],
                                BFGS_Gradient$counts[2]))
knitr::kable(df)


# CG_NoGradient
# BFGS_NoGradient
# CG_Gradient
# BFGS_Gradient
```