Lab3_Block1

Stefano Toffol (steto820), Andreas Stasinakis (andst745), Mim Kemal Tekin (mimte666)

12/18/2018

Contributions:

- Assignment 1 - Andreas Stasinakis(Analysis), Mim Kemal Tekin(Code);
- Assignment 2 - Stefano Toffol;

Assignment 1

Assignment 1: Kernel Methods

Task 1.1

In this task we implement a kernel method in order to predict the hourly temperatures for a data and plane in Sweden. We use 3 different kernels and we combine them with 2 ways. Firstly we take the sum of them and after that the product. The most important thing in this procedure is the selection of the smoothing coefficient or width(h) for every different kernel. A sensible choice of smoothing coefficients would be the one below : i) $h_{distance} = 0.3$ ii) $h_{dates} = 40$ iii) $h_{time} = 3$

Those choices mean that we weight more the observations which are less than 300 kilometers away from the place of interest, less than 40 days after or before of the input's day and less than 3 hours difference from the hour interval selected. In general we should use cross validation, from 5 to 10 folds, in order to choose the optimal coefficients. In this case though, we have 50000 observations so it would be really difficult to do it. In order to choose the optimal hs we use *grid search* for a test dataset of 100 observations. We choose some different values for every smoothing coefficient. In all cases we choose some small values and some high values as boundaries. More specific, for the distance between a station and our place of interest we take a vector of 4 different values (0.1, 0.2 , 0.3, 0.5). The logic behind this choice is that we want to give more weight only for stations which their distances with the point of interest is 100 klm, 200 klm, 300 klm until 500klm (the distances are in meters so we divide by 1000 to convert them to kilometers. It does not make sense to pay attention in stations which their distance is more than 500. For the dates' smoothing coefficients we choose 5 different values (20,30,40,50,100). We are only interesting in observations which their date are not more than 100 days and that is the reason why our highest value of smoothing parameter is 100. We try to split the year in seasons so choosing a value more than 100 does not make that sense. For example, if we want to predict the weather for June and we choose $h = 150$ we will take all observations until November, but in this season the weather is totally different. Finally for the time smoothing parameter, we choose 3 different values (3,5,8). As we explain before, we are only interesting in period in a specific time period of the day, so the highest day interval is 8 hours. So we use the run a *grid search* algorithm for calculating the *MSE* for all h combinations in order to choose the optimal one.

The data frame of the *grid algorithm* we can verify the hypothesis above. The minimum error for the test data is for $h_distance = 0.1 - 0.3$, so we can choose the last one. The estimation of h_date is 40 which also make sense as mentioned before. Finally h_time is estimated 3. As mentioned before, this test data set was really small but we use it only to take a general idea of our choice. The output is the expected one so the h's above seem to be the optimals.

Table 1: MSE ordered by MSE of sum kernel (25 rows)

Var1	Var2	Var3	mse_sumK	mse_prodK
0.5	40	3	54.33083	39.72583
0.3	40	3	54.33135	54.49998
0.2	40	3	54.33211	54.51279
0.1	40	3	54.33363	54.55207
0.5	50	3	54.92026	52.78648
0.3	50	3	54.92075	55.92969
0.2	50	3	54.92148	55.95102
0.1	50	3	54.92297	56.02465
0.5	30	3	55.83737	38.39372
0.3	30	3	55.83795	54.61836
0.2	30	3	55.83884	54.61847
0.1	30	3	55.84069	54.61197
0.5	40	4	58.35383	49.36553
0.3	40	4	58.35434	52.21114
0.2	40	4	58.35514	52.22256
0.1	40	4	58.35695	52.26015
0.5	50	4	58.52393	53.13715
0.3	50	4	58.52441	53.28560
0.2	50	4	58.52516	53.30503
0.1	50	4	58.52687	53.37324
0.5	30	4	60.18373	36.45313
0.3	30	4	60.18432	52.65335
0.2	30	4	60.18524	52.65283
0.1	30	4	60.18744	52.64748
0.5	20	3	60.49365	40.43125

Assignment 2

To accomplish the first part of the assignment, it was decided to split the data in train/validation/set (holdout method) using the following code:

```
data(spam)
n <- nrow(spam)
data <- spam

# Split data in train/validation/test in 50/25/25
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
id1=setdiff(1:n, id)

set.seed(12345)
id2=sample(id1, floor(n*0.25))
valid=data[id2,]
id3=setdiff(id1,id2)
test=data[id3,]
```

After training and validating the various models, their misclassification rates have been computed and reported

in Table @ref(tab:A2-training). It is evident how the model with $C = 1$ is actually the best among the model trained, having the lowest misclassification rate in the validation and the smallest difference between the two datasets.

Table 2: Misclassification Rates for the train and validation datasets according to the different values of C

C	Train	Validation
NA	0.056087	0.086087
NA	0.040000	0.069565
NA	0.021304	0.074783

Using the data left out for the testing of our model, it's then computed the misclassification rate of the optimal model, which results in 0.085143, meaning that 9 out of 10 observations were correctly classified. The performances of the model are summarized in Table @ref(tab:A2-misclass-table)

Table 3: Confusion matrix of the test dataset (C = 1).

	Predicted values		
Real values	Non-spam	Spam	Frequencies
Non-spam	643	31	674
Spam	67	410	477
Frequencies	710	441	1370

The best *SVM* found is summarized in the following:

```
print(best_fit)
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.05
##
## Number of Support Vectors : 1007
##
## Objective Function Value : -467.7423
## Training error : 0.04
```

The C parameter defines the cost of wrongly classify an observation and is used when we drop the assumption of linear separability in the feature space, canonical hypothesis for the regular *SVM*. The parameter penalize the misclassified values, in other words the data standing inside one margin in the feature space. It allows the algorithm to avoid overfitting: in fact if no penalization is given to those observations the linear classifier of the model will closely follow the misclassified observations, subjected to random noise. Penalizing them instead, the linear classifier will be less influenced by them and almost insensible to border line observations.

Appendix

```
knitr::opts_chunk$set(echo = FALSE, fig.width = 4.5, fig.height = 3,
                      fig.align = "center",
                      warning = F, error = F, message = F,
                      fig.pos = 'h')

# Load libraries
library(readxl)
library(kableExtra)
library(ggplot2)
library(grid)
library(gridExtra)
library(viridis)
library(dplyr)
library(geosphere)
library(kernlab)

conf_matrix = function(real_data, predicted_data, levels){
  # make the values factor
  real_data = factor(real_data, levels=levels)
  predicted_data = factor(predicted_data, levels = levels(real_data))
  # we can have "not predicted" values in predicted data
  # make equal the levels of predicted values with real data
  levels(predicted_data) = levels(real_data)

  ct = table(real_data, predicted_data)
  df = data.frame(c(as.vector(ct[,1]), sum(ct[,1])),
                  c(as.vector(ct[,2]), sum(ct[,2])),
                  c(sum(ct[1,]), sum(ct[2,]), sum(ct)))
  rownames(df) = c(levels, "Frequencies")
  colnames(df) = c(levels, "Frequencies")
  return(df)
}

calculate_rate = function(conf_matrix){
  conf_matrix = as.matrix(conf_matrix)
  return(1 - sum(diag(conf_matrix[1:2,1:2]))/sum(conf_matrix[1:2,1:2]))
}

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

gaussian_kernel = function(norm, weight){
  return(exp(-norm^2/(2*weight^2)))
}

sum_kernel = function(...){
  args = list(...)
  for(i in 2:length(args))
    args[[1]] = args[[1]] + args[[i]]
  return(args[[1]])
}
```

```
prod_kernel = function(...){
  args = list(...)
  for(i in 2:length(args))
    args[[1]] = args[[1]] * args[[i]]
  return(args[[1]])
}

# coordinate distance function
# a = c(lon_1, lat_1)
# b = c(lon_2, lat_2)
# returns: coordinate distance between 2 points as kilometers
coor_dist = function(a,b){
  if(!is.numeric(a) || !is.numeric(b))
    stop("a and b inputs should be numeric type!")
  if(length(a)!=2 || length(b)!=2)
    stop("a and b inputs should have size 2!")
  dis = geosphere::distHaversine(a,b)
  return(dis/1000)
}

# day counter function between 2 date
# date1 and date2: yyyy-mm-dd
# returns positive day count between 2 date
days_2_date = function(date1, date2){
  y = 2000
  date1 = sub("\\d{4}", y, as.character(date1))
  date2 = sub("\\d{4}", y, as.character(date2))
  day_count = abs(as.numeric(difftime(date1, date2)))
  return(min(day_count, 365-day_count))
}

# hour counter function between 2 time
# time1 and time2: hh:mm:ss
# returns positive hours between 2 time
hours_2_time = function(time1, time2){
  time1 = as.character(time1)
  time2 = as.character(time2)
  hours = as.difftime(c(time1, time2), units = "hours")
  diff = abs(hours[1]-hours[2])
  return(as.numeric(min(diff, 24-diff)))
}

predict_kernel = function(kernel, real_data){
  pred = sum(kernel*real_data) / sum(kernel)
  return(pred)
}

# predict forecest function
# predicts air tempreture for a day by using sum of 3 kernel
# df_st: train dataset
# input: is a list which is for the day to predict
#     list( loc = c(lat, lon),
#           date = "yyyy-mm-dd",
```

```
#           time = "hh:mm:ss")
# weights: is a vector which has weights for 3 kernels. 0.5 is default for each.
#     c(h_distance, h_date, h_time)
# method: is a parameter for final kernel calculation.
#     "S" for sum of all 3 kernels (default)
#     "P" for product of all 3 kernels
forecast = function(df_st, input, weights=c(0.5,0.5,0.5), method="S"){
  # eliminate the days after the input day
  indexes = which(as.Date(input$date) > as.Date(df_st$date))
  df_st = df_st[indexes, ]
  # get real target values
  y_real = df_st$air_temperature
  # set h values ( weights of kernels )
  h_distance = weights[1]
  h_date = weights[2]
  h_time = weights[3]
  # the point to predict
  x_location = input$loc
  # the date to predict
  x_date = input$date
  # the times to predict
  x_times = input$time

  # calculate location distances
  message(paste(x_times,"- location calculations."))
  loc_dist = apply(df_st[,c("longitude", "latitude")], MARGIN = 1,
                   FUN = coor_dist, b = x_location)
  # calculate date distances
  message(paste(x_times,"- date calculations."))
  date_dist = apply(as.data.frame(df_st[,c("date")]), MARGIN = 1,
                    FUN = days_2_date, date2 = x_date)
  # calculate time distances
  message(paste(x_times,"- time calculations."))
  time_dist = apply(as.data.frame(df_st[,c("time")]), MARGIN = 1,
                    FUN = hours_2_time, time2 = x_times)

  # calculate kernels
  loc_kernel = gaussian_kernel(norm = loc_dist, h_distance)
  date_kernel = gaussian_kernel(norm = date_dist, h_date)
  time_kernel = gaussian_kernel(norm = time_dist, h_time)
  forecast_kernel = NA
  # calculate forecast kernel
  if(method=="S")
    forecast_kernel = loc_kernel + date_kernel + time_kernel
  else if(method=="P")
    forecast_kernel = loc_kernel * date_kernel * time_kernel
  # predict
  prediction = predict_kernel(forecast_kernel, y_real)
  head(forecast_kernel)
  return(prediction)
}

# function that tests different width parameters for kernels
```

```
# df_st: dataset
# test_count: count of observations which will test
# widths: list which has the different width values for kernels
#     list( w_dis = c(numeric()),
#           w_date = c(numeric()),
#           w_time = "c(numeric()))
# returns a dataframe which has all combinations of widths and mse for tests
test_widths = function(df_st, test_count=100, widths){
  # create grid to search
  grids = expand.grid(list(widths$w_dis, widths$w_date, widths$w_time))
  # create some test cases from the data
  set.seed(12345)
  id = sample(1:n, test_count)
  test_data = df_st[id,]
  df_st = df_st[-id, ]
  # create another grid to store mse values and set them 0
  # skipped for skipped observations
  g = grids
  g$mse_sumK = 0
  g$mse_prodK = 0
  skipped = 0
  # main iteration to traverse test observations
  for(j in 1:nrow(test_data)){
    message(paste("################################",j,"test"))
    # get test data and create input object
    test = test_data[j, ]
    input = list(loc = c(test$longitude, test$latitude),
             date = test$date,
             time = test$time)

    # eliminate the days after the input day
    tmp_df = df_st[which(as.Date(input$date) > as.Date(df_st$date)), ]
    y_real = tmp_df$air_temperature

    # calculate location distances
    message(paste(j,"- location calculations."))
    loc_dist = apply(tmp_df[,c("longitude", "latitude")], MARGIN = 1,
                     FUN = coor_dist, b = input$loc)
    # calculate date distances
    message(paste(j,"- date calculations."))
    date_dist = apply(as.data.frame(tmp_df[,c("date")]), MARGIN = 1,
                      FUN = days_2_date, date2 = input$date)
    # calculate time distances
    message(paste(j,"- time calculations."))
    time_dist = apply(as.data.frame(tmp_df[,c("time")]), MARGIN = 1,
                      FUN = hours_2_time, time2 = input$time)
    # htest prediction temp variables
    h_test_preds_sum = c()
    h_test_preds_prod = c()
    for(i in 1:nrow(grids)){
      # print(paste(i,"grid"))
      # get current widths from grid
      h_vector = as.numeric(grids[i,])
```

```
      # calculate kernels
      loc_kernel = gaussian_kernel(norm = loc_dist, h_vector[1])
      date_kernel = gaussian_kernel(norm = date_dist, h_vector[2])
      time_kernel = gaussian_kernel(norm = time_dist, h_vector[3])
      # create sum kernel and product kernel
      sum_kernel = loc_kernel + date_kernel + time_kernel
      prod_kernel = loc_kernel * date_kernel * time_kernel
      # predictions
      pred_sum = predict_kernel(sum_kernel, y_real)
      pred_prod = predict_kernel(prod_kernel, y_real)
      # store predictions
      h_test_preds_sum[i] = pred_sum
      h_test_preds_prod[i] = pred_prod
    }
    # check if predictions has some NaN values, if they have skip this observation
    # because when you try integer(10)+NaN operation, it will be NaN
    if(NaN %in% h_test_preds_prod || NaN %in% h_test_preds_sum){
      message(paste("!!!!",j,"skipped !!!"))
      skipped = skipped + 1
      next
    }
    # get real temperature from test
    yyy = rep(test$air_temperature, length(pred_sum))
    # calculate squared residuals and add the vector to our grid
    res_sum = (yyy - h_test_preds_sum)^2
    res_prod = (yyy - h_test_preds_prod)^2
    g$mse_sumK = g$mse_sumK + res_sum
    g$mse_prodK = g$mse_prodK + res_prod
  }
  # get mean of the squared residuals
  g$mse_sumK = g$mse_sumK/(nrow(test_data)-skipped)
  g$mse_prodK = g$mse_prodK/(nrow(test_data)-skipped)

  return(g)
}



set.seed(1234567890)
df_stations = read.csv("../dataset/stations.csv", fileEncoding="ISO-8859-1")
df_temps = read.csv("../dataset/temps50k.csv")
df_st = merge(df_stations,df_temps,by="station_number")
n = dim(df_st)[1]
##### sample input
# the point to predict c(long,lat)
x_location = c(14.826, 58.4274)
# the date to predict
x_date = "2013-11-04"
# the times to predict
x_times = paste(02:12*2, "00","00", sep = ":")


############## WIDTH TEST ############################################################
```

```r
# implement test width (h) values
hs_distance = c(100/1000, 200/1000, 300/1000, 500/1000)
hs_date =     c(20, 30, 40, 50, 100)
hs_time =     c(3, 4, 5, 8)

ws = list(w_dis = hs_distance,
          w_date = hs_date,
          w_time = hs_time)

res = test_widths(df_st, widths = ws, test_count = 50)

# min sum mse order
o_min_sum = res[order(res$mse_sumK),]
row.names(o_min_sum) = 1:nrow(o_min_sum)
# min prod mse order
o_min_prod = res[order(res$mse_prodK),]
# min mean sum and prod mse order
o_mean_sum_prod = res[order((res$mse_prodK + res$mse_sumK)/2),]
################################################################################


opt_w = as.numeric(o_min_sum[1, c(1,2,3)])

kableExtra::kable(o_min_sum[1:25,], "latex", booktabs = T,
                  caption = "MSE ordered by MSE of sum kernel (25 rows)") %>%
                    kable_styling(latex_options = "hold_position")
# kableExtra::kable(o_min_sum, "latex", booktabs = T, align = "c",
#                     caption = "MSE ordered by sum MSE") %>%
#                     kable_styling(latex_options = "hold_position")
################################## NORMAL CODE ###################################
# weights of kernals (optimal)
h_distance = opt_w[1]     #300/1000
h_date = opt_w[2]         #40
h_time = opt_w[3]         #3
#
# h_distance = 0.3     #300/1000
# h_date = 40         #40
# h_time = 3         #3

w = c(h_distance, h_date, h_time)

preds_sum = c()
preds_prod = c()
for(i in 1:length(x_times)){
  # creating input object
  input = list(loc = x_location,
               date = x_date,
               time = x_times[i])
  message(paste("### sum forecast ", i, "###"))
  preds_sum[i] = forecast(df_st, input, w, method = "S")
  message(paste("### product forecast ", i, "###"))
  preds_prod[i] = forecast(df_st, input, w, method = "P")
}
```

```r
ggplot() +
  geom_point(aes(x=2:12*2, y=preds_sum), color="steelblue") +
  geom_line(aes(x=2:12*2, y=preds_sum), color="steelblue") +
  geom_point(aes(x=2:12*2, y=preds_prod), color="red") +
  geom_line(aes(x=2:12*2, y=preds_prod), color="red") +
  scale_x_continuous(breaks = seq(2,24,2)) +
  theme_bw() +
  labs(title = "Forecast Predictions for 2013-11-04",
       subtitle = "Blue: Sum Kernel, Red: Production Kernel",
       x = "Hours of Day", y = "Predictions")


################################################################################

data(spam)
n <- nrow(spam)
data <- spam

# Split data in train/validation/test in 50/25/25
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
id1=setdiff(1:n, id)

set.seed(12345)
id2=sample(id1, floor(n*0.25))
valid=data[id2,]
id3=setdiff(id1,id2)
test=data[id3,]


# C parameter
C = c(0.5, 1, 5)

models <- list()
misclass_train <- rep(NA, 2)
misclass_valid <- rep(NA, 2)

for(i in 1:length(C)) {

  # Fit the model usign a specific C parameter
  temp_fit <- ksvm(type ~ ., data = train, kernel = "rbfdot",
                   kpar = list(sigma = 0.05), C = C[i])
  models[[paste("SVM_",i, sep = "")]] <- temp_fit

  # Predict for both train and validation
  pred_train <- predict(temp_fit, train)
  pred_valid <- predict(temp_fit, valid)

  # Compute the misclassification rates
  misclass_train[i] <- mean(train$type != pred_train)
  misclass_valid[i] <- mean(valid$type != pred_valid)
```

```
}

df_table <- data.frame(C = C,
                       Train = format(misclass_train, digits = 5),
                       Validation = format(misclass_valid, digits = 5))
rownames(df_table) <- NULL

kable(df_table, "latex", booktabs = T, align = "c", digits = c(NA, 6, 6),
      caption = "Misclassification Rates for the train and validation
      datasets according to the different values of C ") %>%
  column_spec(1, border_right = T) %>%
  kable_styling(latex_options = "hold_position")


best_fit = models$SVM_2
best_pred = predict(best_fit, newdata = test)
misrate_best = mean(best_pred != test$type)


# Confusion matrix test
temp <- table(test$type, best_pred)
kable(data.frame(
      c("Non-spam", "Spam", "Frequencies"), c(temp[1:2], sum(temp[1:2])),
      c(temp[3:4], sum(temp[3:4])), c(sum(temp[c(1,3)]), sum(temp[c(2,4)]), 1370)),
    col.names = c("Real values", "Non-spam", "Spam", "Frequencies"),
    "latex", booktabs = T, align = "c",
    caption = "Confusion matrix of the test dataset  (C = 1).") %>%
  add_header_above(c(" " = 1, "Predicted values" = 2, " " = 1)) %>%
  column_spec(c(1,3), border_right = T) %>%
  row_spec(2, hline_after = T) %>%
  kable_styling(latex_options = "hold_position", font_size = 8)


print(best_fit)
```