# Code

```
# Packes required for subsequent analysis. P_load ensures these will be installed and loaded.
if (!require("pacman")) install.packages("pacman")
pacman::p_load(fastDummies,
               SVMMaj,
               dplyr,
               caret,
               mclust,
               VeryLargeIntegers,
               matlib,
               latex2exp,
               reshape2,
               kernlab,
               plot3D,
               stargazer,
               rdetools)

##############################################################################################
# Functions to compute the SVM and it cross validation
##############################################################################################

# function that creates parameters for different hinge errors
create_hinge_param <- function(mY, vQ, vZ, hinge = "absolute", epsilon = 1e-08, k_huber = 1){

  # when the hinge is absolute
  if(hinge == "absolute"){

    # add epsilon threshold to make sure its invertible
    m <- abs(1 - vZ)
    m <- m * (m > epsilon) + epsilon * (m <= epsilon)

    # calculate a and b using m
    a <- .25 * m^-1
    b <- mY * (a + 0.25)

    return(list(a = a,
                b = b))

  }else if (hinge == "quadratic"){

    # for quadratic hinge, no need to calculate a (always identity)
    m <- vZ* (vZ > 1) + (1 >= vZ)
    b <- m * mY

    return(list(b = b))

  }else if(hinge == "huber"){
```

```
    # a is the same for both cases
    a <- 0.5 * (k_huber + 1)^-1

    # use if statements to define b and c
    b <- (mY == -1) * ((vQ <= -1) * (a * vQ) + # if the Y == -1
            (vQ >= k_huber) * (a * vQ - 0.5) +
            (vQ > -1 & vQ < k_huber) * (-a)) +
         (mY == 1) * ((vQ <= -k_huber) * (0.5 + a * vQ) + # if the Y == 1
            (vQ >= 1) * (a * vQ) +
            (vQ > -k_huber & vQ < 1) * (a))

    c <- (mY == -1) * ((vQ <= -1) * (a * vQ^2) +
            (vQ >= k_huber) * (1 - (k_huber + 1)/2 + a * vQ^2) + # if the Y == -1
            (vQ > -1 & vQ < k_huber) * (a)) +
         (mY == 1) * ((vQ <= -k_huber) * (1 - (k_huber + 1)/2  + a * vQ^2) + # if the Y == 1
            (vQ >= 1) * (a * vQ^2) +
            (vQ > -k_huber & vQ < 1) * (a))

    return(list(a = a,
                b = b,
                c = c))
  }
}

# calculate the loss of a particular prediction, based on the type of hinge error
calc_loss <- function(mW, lambda, vZ, mY = NA, hinge = "absolute", vQ = NA, k_huber = NA ){

  # create penalty variable
  mWtW = t(mW) %*% mW
  penalty = lambda * mWtW

  # use z for fater computation
  if(hinge == "absolute"){

    vloss <- (1 > vZ) * (1 - vZ)

  }else if(hinge == "quadratic"){

    vloss <- (1 > vZ) * (1 - vZ)^2

  # for huber, use if statements to categorize each case
  }else if(hinge == "huber"){

    vloss <- (mY == 1) * (vQ <= -k_huber) * (1 - vQ -(k_huber + 1)/2) + # if Y == 1
      (mY == 1) * (vQ > -k_huber) * (0.5 * (k_huber+1)^-1 * pmax(0,1 - vQ) ^2) +
      (mY == -1) * (vQ <= k_huber) * (0.5 * (k_huber+1)^-1 * pmax(0,1 + vQ) ^2) + # if Y == -1
      (mY == -1) * (vQ > k_huber) * (vQ +1 -(k_huber + 1)/2)

  }else{
    stop("Not given an known hinge error")
  }

  # return loss + penalty
```

```r
  return(sum(vloss) + penalty)
}


# implement support vector machine
svm_mm <- function(mY, mX, hinge = "absolute", lambda = 10, epsilon = 1e-08, k_huber = 3){

  # set initial weights and constant. From these, create initial v = c + w
  vW = runif(ncol(mX)-1, -1, 1)
  fConstant = 0.0
  vV_current = as.matrix(c(fConstant, vW))

  # get n of observations in the sample, and p variables
  n = nrow(mX)
  p = ncol(mX)

  # define matrix p
  P = diag(p)
  P[1,1]<-0

  # set initial values; k, stepscore
  k = 1
  stepscore = 0

  if(hinge == "quadratic"){
    mZ = inv(t(mX) %*% mX + lambda * P) %*% t(mX)
  }

  while(k ==1 || stepscore > epsilon){

    # get current prediction (q) and z
    vCurrent_q = mX %*% vV_current
    vCurrent_z = vCurrent_q * mY

    # get parameters given the z (absolute and quadratic) or q and y (huber)
    lHinge_param_current = create_hinge_param(mY = mY,
                                              vQ = vCurrent_q,
                                              vZ = vCurrent_z,
                                              hinge = hinge,
                                              epsilon = epsilon,
                                              k_huber = k_huber)
    # all hinges need a b, so can define here
    b = lHinge_param_current$b

    # quick update if quadratic
    if(hinge == "quadratic"){

      vV_update = mZ %*% b

    }else {

      # define A and b
      A = diag(as.vector(lHinge_param_current$a),n)
```

```
    # get updated v
    vV_update = solve(t(mX) %*% A %*% mX + lambda * P, t(mX) %*% b)

  }

  # get weights in previous and next
  mW_current = tail(vV_current,-1)
  mW_update = tail(vV_update, -1)

  # get new prediction (q) and z
  vNew_q = mX %*% vV_update
  vNew_z = vNew_q * mY


  # calculate new, and previous loss
  fCurrent_loss = calc_loss(mW = mW_current,
                            vQ = vCurrent_q,
                            vZ = vCurrent_z,
                            mY = mY,
                            lambda = lambda,
                            hinge = hinge,
                            k_huber = k_huber)


  fNew_loss = calc_loss(mW = mW_update,
                        vQ = vNew_q,
                        vZ = vNew_z,
                        mY = mY,
                        lambda = lambda,
                        hinge = hinge,
                        k_huber = k_huber)

  # calculate improvement
  stepscore <- (fCurrent_loss - fNew_loss)/fCurrent_loss

  # check: if all predicted correctly, turns to NaN since divided by zero
  if (is.na(stepscore)){
    stepscore = 0
  }

  # move to next iteration
  k = k + 1
  vV_current = vV_update
}

# get predicted category
mY_hat = sign(vNew_q)


# gather results
mConfusionTable <- table(mY, mY_hat)

# create confusion matrix and calculate accuracy
```

```
  fAccuracy <- sum(mY_hat == mY)/length(mY)

  fARI <- adjustedRandIndex(mY, mY_hat)

  # return results object
  lResults = list(v = vV_update,
                  mY = mY,
                  mX = mX,
                  loss = fCurrent_loss,
                  q = vNew_q,
                  yhat = mY_hat,
                  Accuracy = fAccuracy,
                  ConfusionTable = mConfusionTable,
                  ARI = fARI)

  return(lResults)

}

# cross validation for svm
svm_mm_cv <- function(mX, mY, lambda, folds, hinge = "absolute", k_huber = NA, epsilon = 1e-08, metric =

  # save total score on metric here
  fTest_metric <- 0

  #Perform k fold cross validation
  for(i in 1:length(folds)){

    #Split the data according to the folds
    vTest_id = folds[[i]]
    vTrain_id = -folds[[i]]

    # define train and test set for y and x
    mY_train <- mY[vTrain_id]
    mX_train <- mX[vTrain_id,]
    mY_test <- mY[vTest_id]
    mX_test <- mX[vTest_id,]


    # if summaj bool selected, then implement with one of the kernels kernel selected
    # get result from training set
    if(svmmaj_bool & kernel_type == "linear"){

      lResult <- svmmaj(y = mY_train, X = mX_train, hinge = hinge, lambda = lambda, scale = "none", kern
      vQ_test <- mX_test %*% lResult$beta

      # if chosen, implement rbf kernel
    }else if(svmmaj_bool & kernel_type == "rbf"){

      lResult <- svmmaj(y = mY_train, X = mX_train, hinge = hinge, lambda = lambda, scale = "none", kern


      #if chosen, implement polynomial kernel
```

```
    } else if(svmmaj_bool & kernel_type == "polynom"){

      lResult <- svmmaj(y = mY_train, X = mX_train, hinge = hinge, lambda = lambda, scale = "none", kern
      vQ_test <- mX_test %*% lResult$beta

    } # if chosen, implement non-kernel version
      else {
      lResult = svm_mm(mY = mY_train, mX = mX_train, hinge = hinge, lambda = lambda, epsilon = epsilon,
      vQ_test <- mX_test %*%  lResult$v

    }

    # calculate the predicted y for this fold
    mY_hat <- sign(vQ_test)

    if(metric == "ARI"){

      # get the confusion matrix
      fAdjRand <- adjustedRandIndex(mY_test, mY_hat)
      fTest_metric_fold <- fAdjRand

    }else if (metric == "misclassification"){

      # calculate misclassification
      tab <- table(mY_test, mY_hat)
      fMisclassification <- 1-sum(diag(tab))/sum(tab)
      fTest_metric_fold <- fMisclassification

    }

    # add to calculate average
    fTest_metric <- fTest_metric + fTest_metric_fold

  }

  # calculate average
  avg_test_metric = fTest_metric/length(folds)

  lResult_cv <- list(lambda = lambda,
                     metric = avg_test_metric)

  return(lResult_cv)


}

# search best combination of parameters with gridsearch
svm_mm_gridsearch <- function(mX, mY, lambda, hinge = "absolute", k, k_huber = NA, epsilon = 1e-08, meti

  # change parameter grid based on type of hinge error
  if (hinge == "huber"){

    mParamgrid = expand.grid(lambda, k_huber)
```

```
  }else {

    mParamgrid = expand.grid(lambda)
  }

  # create k equally size folds
  folds = createFolds(mY, k = k, list = TRUE, returnTrain = FALSE)

  mParamgrid$metric <- NA

  # iterate over the grid
  for(i in 1:nrow(mParamgrid)){

    # select parameters from the grid
    param <- mParamgrid[i,]

    print(param[1,c(1,2)])

    # test these with k-fold
    cv_result <- svm_mm_cv(mX = mX,mY= mY, lambda =param[1,1], k_huber = param[1,2],folds = folds, hinge

    print(cv_result$metric)

    #save the result
    mParamgrid$metric[i] <- cv_result$metric
  }

  return(mParamgrid)
}


#############################################################################################
# Helper functions for subsequent analysis
#############################################################################################

# function to analyse results of svm_mm
analyse_svm_result <- function(mY, vQ, plot_title = NaN){

  # get predicted category
  mY_hat <- sign(vQ)

  # get confusion matrix
  mConfusionMatrix <- table(mY,mY_hat)

  # get ARI
  adjRand <- adjustedRandIndex(mY_test, mY_hat)

  # make plot to show which ones were wrongly predicted
  dfComparePlot  <- data.frame(q = vQ, mY = mY_test)
  ComparePlot <- ggplot(data = dfComparePlot, aes(x = q, fill = as.factor(mY))) +
    geom_histogram(bins = 50,alpha = 0.7) +
    labs(
      title = plot_title,
```

```
      y = "Count",
      x = TeX("$q$")
    ) +
    scale_fill_manual(values=c("red", "blue"),
                      name = "Result",
                      labels = c("Did not take subscription (-1)", "Took subscription (1)")) +
    theme_bw() +
    theme(plot.title =element_text(size=20, face = "plain",hjust = 0.5),
          axis.title=element_text(size=15, face = "plain"))

  return(list(mY_hat = mY_hat,
              "Table" = mConfusionMatrix,
              ARI = adjRand,
              ComparePlot = ComparePlot
  ))


}

# function to create plot that shows difference between types of errors
create_show_df <- function(vk_huber_show){ # specify as an argument which errors one wants to show

  # define dataframe, get q
  df_show <- data.frame( q =seq(-3,3,by=0.2) )
  vQ_show = df_show$q

  # count to change column names appropriately
  col_count = 1

  # iterate over each type of
  for(ik_huber_show in vk_huber_show){

    # define plus one and minus one loss
    vloss_plusOne <-(vQ_show <= -ik_huber_show) * (1 - vQ_show -(ik_huber_show + 1)/2) +
      (vQ_show > -ik_huber_show) * (0.5 * (ik_huber_show+1)^-1 * pmax(0,1 - vQ_show) ^2)

    vloss_minusOne <-  (vQ_show <= ik_huber_show) * (0.5 * (ik_huber_show+1)^-1 * pmax(0,1 + vQ_show) ^
      (vQ_show > ik_huber_show) * (vQ_show +1 -(ik_huber_show + 1)/2)

    # change column names in dataframe
    col_names_huber = c(paste0("plusOne_", ik_huber_show),paste0("minusOne_", ik_huber_show) )
    df_show$plusOne <- vloss_plusOne
    df_show$minusOne <- vloss_minusOne
    colnames(df_show)[-(1:col_count)] <- col_names_huber
    col_count = col_count + 2


  }

  # add other types of losses: absolute and quadratic
  vAbsError_plusOne <- (1 > vQ_show) * (1 - vQ_show)
  vAbsError_minusOne <- (1 > -vQ_show) * (1 - -vQ_show)
  vQuadError_plusOne <- (1 > vQ_show) * (1 - vQ_show)^2
```

```
  vQuadError_minusOne <- (1 > -vQ_show) * (1 - -vQ_show)^2

  # add all to same dataframe
  df_show$plusOne_abserror <- vAbsError_plusOne
  df_show$minusOne_abserror <- vAbsError_minusOne
  df_show$plusOne_quaderror <- vQuadError_plusOne
  df_show$minusOne_quaderror <- vQuadError_minusOne


  return(df_show)

}

# quick function to get the optimal parameters summajcrossval, based on ARI
create_dfCompare_cv <- function(svmmaj_cv_obj){

  # dataframe where all ARI will be added
  AdjRand_compare <- data.frame(svmmaj_cv_obj$param.grid[,-ncol(svmmaj_cv_obj$param.grid)])
  AdjRand_compare$ARI <- 0

  vQ_cv <-svmmaj_cv_obj$qhat

  for(i in 1:k){

    mY_hat <- sign(vQ_cv[i,])

    print(sum(mY_hat == sample_y)/length(sample_y))

    adjRand_fold <- adjustedRandIndex(sample_y, mY_hat)

    AdjRand_compare$ARI[i] = adjRand_fold

  }

  return(AdjRand_compare)


}



##############################################################################################
# Analysis of bank data
##############################################################################################


#### section 1: data pre-processing

# ensure that we can reproduce the code
set.seed(333)

# load bank data
bank <- read.csv("bank-additional.csv", sep = ";")
```

```r
#load("bank.Rdata")

# get dependent variable and transform to 1, -1
mY <- bank$y
mY_num <- as.matrix(ifelse(mY == "yes", 1, -1))

# get all possible independent variables
mX = bank[,-ncol(bank)]

# create dummy variables
df_toDummy = as.matrix(data.frame(
  mX$job,
  mX$marital,
  mX$education,
  mX$default,
  mX$housing,
  mX$loan,
  mX$contact,
  mX$month,
  mX$day_of_week,
  mX$poutcome,
  mX$pdays))

dummy_vars <-  dummy_columns(df_toDummy)
dummy_vars <- dummy_vars[,(1+ncol(df_toDummy)):ncol(dummy_vars)]

# select numeric variables, scale these. Leave out duration for realistic predictions
numeric_vars <- as.matrix(data.frame(
  mX$age,
  mX$campaign,
  mX$cons.price.idx,
  mX$cons.conf.idx,
  mX$nr.employed,
  mX$emp.var.rate,
  mX$euribor3m))

# combine numeric and dummy variables
mX_var <- as.matrix(cbind(1, scale(numeric_vars),dummy_vars))

# pick a random sample of 1000
sample_id <- sample(4000, 1000)
sample_x = mX_var[sample_id,]
sample_y = mY_num[sample_id]


### section 2: compare a single svm_mm with svm_maj

# We get the same predictions (see confusion matrix), and exactly the same loss, for each type of error
# but still different beta's? most likely something to do with the transformation of the data...

# our implementation
result_abs_compare = svm_mm(sample_y, sample_x, lambda = 0.5, hinge = "absolute")
result_quad_compare = svm_mm(sample_y, sample_x, lambda = 0.5, hinge = "quadratic")
```

```r
result_huber_compare = svm_mm(sample_y, sample_x, lambda = 0.5, hinge = "huber", k_huber = 3)


#svmmaj implementation
result_svmmaj_abs_compare <- svmmaj(sample_x,sample_y, lambda = 0.5, scale = "none",hinge = "absolute")
result_svmmaj_quad_compare <- svmmaj(sample_x,sample_y, lambda = 0.5, scale = "none",hinge = "quadratic
result_svmmaj_huber_compare  <- svmmaj(sample_x,sample_y, lambda = 0.5, scale = "none",hinge = "huber",


result_abs_compare$ConfusionTable
result_quad_compare$ConfusionTable
result_huber_compare$ConfusionTable

result_svmmaj_abs_compare$q
result_svmmaj_huber_compare$q
result_svmmaj_quad_compare$q


### section 3: compare cross validation results, and create cv plots

# check out the following parameters
vLambda =10^seq(5, -3, length.out= 10)
vk_huber = seq(0,3, by = 1)

# k-fold of 10 to have enough info, but not make it computationally too intensive
k = 10

# cv comparison - example is quadratic since faster
result_svm_mm_cv <- svm_mm_gridsearch(sample_x, sample_y, k = k, lambda = vLambda, hinge = "quadratic",
result_svmmaj_cv <- svmmajcrossval(sample_x,sample_y, search.grid = list(lambda = vLambda) , k = k,scal

# data cleaning of our cv results
colnames(result_svm_mm_cv) <- c("lambda", "misclassification")

# finds same lambda for quadratic
ggplot(data = result_svm_mm_cv, aes(x = log(lambda), y = misclassification, col = "red"))+
  geom_line(size = 1.5)+
  geom_line(data = result_svmmaj_cv$param.grid, aes(y = loss, col = "blue"), size = 1.5) +
  scale_color_manual(name = "Type of CV",
                       labels = c("svmmaj",
                                  "our implementation"),
                     values = c("red","blue")
  ) +
  labs(title = "Comparing the CV implementations (quadratic error)",
       y = "Misclassification") +
  theme_bw()  +
  theme(plot.title =element_text(size=20, face = "plain",hjust = 0.5),
                    axis.title=element_text(size=15, face = "plain"))



## now create plots per type of error for optimal lambda
result_cv_absolute <- svm_mm_gridsearch(sample_x, sample_y, k = k, lambda = vLambda, hinge = "absolute"
```

```r
result_cv_quadratic <- svm_mm_gridsearch(sample_x, sample_y, k = k, lambda = vLambda, hinge = "quadrati
result_cv_huber <- svm_mm_gridsearch(sample_x, sample_y, k = k, lambda = vLambda, k_huber = vk_huber, hi

# change column names
colnames(result_cv_quadratic) <- c("lambda", "ARI")
colnames(result_cv_absolute) <- c("lambda", "ARI")
colnames(result_cv_huber) <- c("lambda", "k_huber", "ARI")

# get optimal parameters
optimal_lambda_quadratic <- round(result_cv_quadratic[which.max(result_cv_quadratic$ARI),]$lambda,1)
optimal_lambda_absolute <- round(result_cv_absolute[which.max(result_cv_absolute$ARI),]$lambda,1)
optimal_lambda_huber <- result_cv_huber[which.max(result_cv_huber$ARI),]$lambda
optimal_k_huber <- result_cv_huber[which.max(result_cv_huber$ARI),]$k_huber


# create dataframe for plot that compares quadratic and absolute error
df_cv_compare <- data.frame(lambda = result_cv_absolute$lambda, abs_ARI = result_cv_absolute$ARI, quad_
df_cv_compare <- melt(df_cv_compare, id.vars = "lambda")

# finds same lambda for quadratic
ggplot(data = df_cv_compare, aes(x = log(lambda), y = value, col = variable))+
  geom_line(size = 1.5) +
  labs(y = "Average ARI",
       title = "Result of CV"

  ) +
  scale_color_manual(name = "Type of error",
                     labels = c("Absolute", "Quadratic", "Huber (at k = 1)"),
                     values = c("red","blue", "green")
  )  +
  theme_bw()  +
  theme(plot.title =element_text(size=20, face = "plain",hjust = 0.5),
        axis.title=element_text(size=15, face = "plain"))


### section 4: test the optimal parameters from cv on a train and test set

## 75% of the sample size
fTrain_size <- floor(0.75 * nrow(sample_x))

# get train indexes for the dataset
vTrain_id <- sample(nrow(sample_x), size = fTrain_size)

# train and test split
mX_train <- sample_x[vTrain_id, ]
mX_test <- sample_x[-vTrain_id, ]
mY_train <- sample_y[vTrain_id]
mY_test <- sample_y[-vTrain_id]


# train for the beta's
result_absolute <- svm_mm(mY = mY_train,mX= mX_train, lambda = optimal_lambda_absolute, hinge = "absolu
result_quadratic <- svm_mm(mY = mY_train,mX= mX_train, lambda = optimal_lambda_quadratic, hinge = "quadi
```

```r
result_huber <- svm_mm(mY = mY_train, mX= mX_train, lambda = optimal_lambda_huber,k_huber = optimal_k_hu

# calculate the q for test set, based on weights from training
vQ_test_quadratic <- mX_test %*% result_quadratic$v
vQ_test_absolute <- mX_test %*% result_absolute$v
vQ_test_huber <- mX_test %*% result_huber$v

# get results per type
analysis_quadratic <- analyse_svm_result(mY_test, vQ_test_quadratic, plot_title = "Results for quadrati
analysis_absolute <- analyse_svm_result(mY_test, vQ_test_absolute, plot_title = "Results for the absolu
analysis_huber <- analyse_svm_result(mY_test, vQ_test_huber, plot_title = "Results for the Huber error"

### section 5: get hyperparameters for the RBF and polynomial kernels

# define sigma as to get a gamme of 1/m
sigma_rbf <- ncol(mX_train)/2

# define the lambdas and k_huber over which to iterate
vLambda_kernel <-  10^seq(5, -3, length.out= 10)
k_huber_kernel <- seq(0,3, by = 1)

# grid of erros to test
grid_absQuad <- list(lambda = vLambda_kernel)
grid_huber <- list(lambda = vLambda_kernel, hinge.delta = k_huber_kernel)

# RBF
result_cv_quadratic_rbf <- svmmajcrossval(X = sample_x, y = sample_y, hinge = "quadratic", ngroup = k,
                                          search.grid = grid_absQuad, scale = "none",kernel = rbfdot, s

result_cv_huber_rbf <- svmmajcrossval(X = sample_x, y = sample_y, hinge = "huber", ngroup = k,
                                      search.grid = grid_huber, scale = "none",kernel = rbfdot, sigma =


# create dfs to compare
df_Compare_rbf_quad_cv <- create_dfCompare_cv(result_cv_quadratic_rbf)
df_Compare_rbf_huber_cv <- create_dfCompare_cv(result_cv_huber_rbf)
colnames(df_Compare_rbf_quad_cv) <- c("lambda", "ARI")
colnames(df_Compare_rbf_huber_cv) <- c("lambda", "k_huber", "ARI")

# define optimal parameters for rbf
optimal_lambda_quadratic_rbf <- df_Compare_rbf_quad_cv[which.max(df_Compare_rbf_quad_cv$ARI),]$lambda
optimal_lambda_huber_rbf <- df_Compare_rbf_huber_cv[which.max(df_Compare_rbf_huber_cv$ARI),]$lambda
optimal_k_huber_rbf <- df_Compare_rbf_huber_cv[which.max(df_Compare_rbf_huber_cv$ARI),]$k_huber


## polynomial
degree_polynom = 2

result_cv_quadratic_polynom <- svmmajcrossval(X = sample_x, y = sample_y, hinge = "quadratic", ngroup =
                                              search.grid = grid_absQuad, scale = "none",kernel = rbfdo
result_cv_huber_polynom <- svmmajcrossval(X = sample_x, y = sample_y, hinge = "huber", ngroup = k,
                                          search.grid = grid_huber, scale = "none",kernel = rbfdot, deg
```

```r
df_Compare_polynom_quad_cv <- create_dfCompare_cv(result_cv_quadratic_polynom)
df_Compare_polynom_huber_cv <- create_dfCompare_cv(result_cv_huber_polynom)

colnames(df_Compare_polynom_quad_cv) <- c("lambda", "ARI")
colnames(df_Compare_polynom_quad_cv) <- c("lambda", "k_huber", "ARI")

optimal_lambda_quadratic_polynom <- df_Compare_polynom_quad_cv[which.max(df_Compare_polynom_quad_cv$ARI
optimal_lambda_huber_polynom <- df_Compare_polynom_huber_cv[which.max(df_Compare_polynom_huber_cv$ARI),]
optimal_k_huber_polynom <- df_Compare_polynom_huber_cv[which.max(df_Compare_polynom_huber_cv$ARI),]$k_hu


### section 6: test kernels on test set

# RBF kernels
result_quadratic_rbf <- svmmaj(y = mY_train,X= mX_train, lambda = optimal_lambda_quadratic_rbf, hinge =
result_huber_rbf <- svmmaj(y = mY_train,X= mX_train, lambda = optimal_lambda_huber_rbf, degree = optimal


# get predicted values
vQ_test_quadratic_rbf <-  predict(result_quadratic_rbf, X.new = mX_test, y = mY_test)[1:nrow(mX_test)]
vQ_test_huber_rbf <- predict(result_huber_rbf, X.new = mX_test, y = mY_test)[1:nrow(mX_test)]

# get results
analysis_quadratic_rbf <- analyse_svm_result(mY_test, vQ_test_quadratic_rbf)
analysis_huber_rbf <- analyse_svm_result(mY_test, vQ_test_huber_rbf)


# Polynom
result_quadratic_polynom <- svmmaj(y = mY_train,X= mX_train, lambda = optimal_lambda_quadratic_rbf, hing
result_huber_polynom <- svmmaj(y = mY_train,X= mX_train, lambda = optimal_lambda_huber_rbf, degree = opt

vQ_test_quadratic_polynom <-  predict(result_quadratic_polynom, X.new = mX_test, y = mY_test)[1:nrow(mX
vQ_test_huber_polynom <- predict(result_huber_polynom, X.new = mX_test, y = mY_test)[1:nrow(mX_test)]

analysis_quadratic_polynom <- analyse_svm_result(mY_test, vQ_test_quadratic_polynom)
analysis_huber_polynom <- analyse_svm_result(mY_test, vQ_test_huber_polynom)

### Section 7: miscellaneous plot
vk_huber_show = c(0)
df_show_plot <- create_show_df(vk_huber_show)

df_show_plot_melted <- melt(df_show_plot, measure.vars = colnames(df_show_plot)[-1] )


# plot for appendix to show differences in the errors
ggplot(data = df_show_plot_melted, aes(x = q, y = value, col = variable)) +
  geom_line(size = 2) +
  labs(y = "Loss") +
  lims(y = c(0,5)) +
    scale_color_discrete(name = "Function",
                         labels = c("+1 Huber loss, k_huber = 0",
                                    "-1 Huber loss, k_huber  = 0",
                                    "+1 Absolute loss",
```

```
                                    "-1 Absolute loss",
                                    "+1 Quadratic loss",
                                    "-1 Quadratic loss")
  ) +
labs(title = "Comparing the different types of errors") +
theme_bw()  +
theme(plot.title =element_text(size=20, face = "plain",hjust = 0.5),
      axis.title=element_text(size=15, face = "plain"))
```