# 590hw1

Xiansi

2023/9/24

## ————————————————-HW1————————————————

```
library(keras)
library(reticulate)
library(tensorflow)
library(e1071)
```

## ———————— The functions we need for hw1————————————

```
square_loss <- function(true, pred) mean((true - pred) ^ 2)

crossentropy_loss <- function(true, pred) -rowSums(true * log(pred))

model <- function(inputs, W, b) (tf$matmul(inputs, W) + b)

training_step <- function(inputss, targetss, W, b, lr, loss_fun = square_loss) {
  with(tf$GradientTape() %as% tape, {
    ## Forward pass, inside a gradient tape scope
    loss <- loss_fun(true = targetss, pred = model(inputss, W, b))
  })
  grad_loss_wrt <- tape$gradient(loss, list(W = W, b = b))
  ## Retrieve the gradient of the loss with regard to weights.
  ## Now update the weights:
  W$assign_sub(grad_loss_wrt$W * lr)
  b$assign_sub(grad_loss_wrt$b * lr)
  loss
}
```

##————import data for Q1————————##

```
data = read.csv("D:/iowa state/STAT590s3/hw1/sdss-all-c.csv", header = FALSE)
names(data) = c("ID","Class", "B","S","T", "M1", "M2")
data$Class = ifelse(data$Class == 3, 0, ifelse(data$Class == 6, 1, data$Class))
cdata = subset(data, !grepl("\\?", M1) & !grepl("\\?", M2))
set.seed(123)
n = nrow(cdata)
train_indices = sort(sample(1:n, 0.75 * n))
training_dataQ1 = cdata[train_indices, ]
test_dataQ1     = cdata[-train_indices, ]

inputsQ1      = as.matrix(data.frame(lapply(training_dataQ1[,c(3:7)], as.numeric)))
targetsQ1     = training_dataQ1$Class
```

```
inputsT       = as.matrix(data.frame(lapply(test_dataQ1[,c(3:7)], as.numeric)))
targetsT      = test_dataQ1$Class
```

**———-1(a)(i)evaluate what sorts of learning rates do well on the test set—**

```
input_dim = 5
output_dim = 1
inputs  = tensorflow::as_tensor(scale(inputsQ1), dtype = "float32")
inputsT = tensorflow::as_tensor(scale(inputsT), dtype = "float32")
learningrate = c(0.01, 0.05, 0.1, 0.15, 0.2, 0.5)
accuracy_results = data.frame(matrix(nrow = length(learningrate), ncol = 4))
colnames(accuracy_results) <- c("learningrate", "accuracy", "step","square_loss")


myfun <- function(train_x, train_y, val_x, val_y, lr, max_iter = 1e3, eps = 1e-3) {
  p_input <- ncol(train_x)
  p_output <- ncol(train_y)
  W <- tf$Variable(initial_value = tf$random$uniform(shape(p_input, p_output)))
  b <- tf$Variable(initial_value = tf$zeros(shape(p_output)))
  curr <- Inf
  for (step in seq_len(max_iter)) {
    loss_tensor <- training_step(inputss = train_x, targetss = train_y, W = W, b = b, lr = lr)
    loss <- tf$get_static_value(loss_tensor)
    if (abs(loss - curr) <= eps) break
    curr <- loss
  }
  pred <- model(val_x, W = W, b = b)
  pred <- if (p_output == 1) as.numeric(pred >= 0.5) else apply(pred, 1, which.max) - 1
  c(lr = lr, acc = mean(pred == c(val_y)), step = step, loss = loss)
}

for (ll in 1:length(learningrate)) {
  accuracy_results[ll, ] <- myfun(train_x = inputs, train_y = as.matrix(targetsQ1),
                                  val_x = inputsT, val_y = targetsT, lr = learningrate[ll])
}
accuracy_results
```

```
##   learningrate  accuracy step square_loss
## 1         0.01 0.9209809  106  0.07764914
## 2         0.05 0.9891008   34  0.04598758
## 3         0.10 0.9945504   23  0.04191615
## 4         0.15 0.9945504   20  0.04052493
## 5         0.20 1.0000000   14  0.03954885
## 6         0.50 1.0000000    8  0.03849263
```

Implement linear classification to distinguish between the two classes. Conduct experiments using various learning rates, specifically 0.01, 0.05, 0.1, 0.15, and 0.2, with the second column representing the accuracy of predictions. In this scenario, make sure to standardize the original five features. It's noteworthy that in this case, the performance of the learning rates tends to improve as they become larger. However, beyond a certain threshold, accuracy starts to decline.

**———-1(a)(ii)Use support vector machines to separate the two data—**

```
inputsQII      = data.frame(lapply(training_dataQ1[,c(3:7)], as.numeric))
names(inputsQII) = NULL

testdata =  data.frame(lapply(test_dataQ1[,c(3:7)], as.numeric))
testdata = data.frame(scale(testdata))
names(testdata) = NULL
testdata = data.frame(x=testdata,class=as.factor(test_dataQ1$Class))

kernelX = c("linear","radial","polynomial","sigmoid")
result1alinear = data.frame(matrix(nrow = length(kernelX), ncol = 2))
colnames(result1alinear) <- c("kernel", "accuracy")
for (kk in 1:4) {
  svmfitlinear =  svm(targets~., data = data.frame(x=inputsQII,targets=as.factor(targetsQ1)),
                      kernel = kernelX[kk])
  pred_test = table(true = testdata$class, pred = predict(svmfitlinear, newdata = testdata))
  result1alinear[kk,] = c(kernelX[kk], sum(diag(pred_test))/sum(pred_test))
}

result1alinear
```

```
##       kernel          accuracy
## 1     linear 0.188010899182561
## 2     radial 0.811989100817439
## 3 polynomial 0.188010899182561
## 4    sigmoid 0.188010899182561
```

In this scenario, I used the svm() function to separate the two sets of data using different kernels, while keeping the default cost parameter settings. Based on the results, it appears that the other kernels tend to predict all the classes as one, whereas the radial kernel provides a more accurate and meaningful classification.

**———-1(a)(iii)Use cross-validation to decide on which of these kernels is the best, and the cost parameter—**

```
kernelX = c("linear","radial","polynomial","sigmoid")
accuracy_rel = data.frame(matrix(nrow = length(kernelX), ncol = 3))
colnames(accuracy_rel) <- c("kernel", "accuracy", "cost")

for (ll in 1:4) {
  tune_out    = tune(svm,
                     targets~.,
                     data = data.frame(x=inputsQII,targets=as.factor(targetsQ1)),
                     kernel = kernelX[ll],
                     ranges = list(cost = c(0.001, 0.01, 0.1, 1,5,10,100)),
                     cachesize = 4096, tolerance = 0.01, tunecontrol = tune.control(cross = 5))
  summary(tune_out)
  bestmod  <- tune_out$best.model
  summary(bestmod)

  class_pred = predict(bestmod, testdata)
  Res = table(predicted = class_pred, true = as.factor(test_dataQ1$Class))
  accuracy_rel[ll,] = c(kernelX[ll], sum(diag(Res))/ sum(Res),bestmod$cost)
}
```

```
accuracy_rel
```

```
##       kernel          accuracy cost
## 1     linear 0.188010899182561    5
## 2     radial 0.811989100817439    5
## 3 polynomial 0.188010899182561    5
## 4    sigmoid 0.188010899182561  0.1
```

In this context, for each decision boundary, I tuned the parameters to compare the best cost associated with them. The results indicate that the nonlinear decision boundary with a radial kernel performs the best when the cost parameter is set to 10.

###–1(b)Re-evaluate performance of the above, but with datasets having complete fields###

```
data1b = data[,c(2:5)]
set.seed(123)
m = nrow(data1b)
train_indices1b = sample(1:m, 0.75 * m)
training_data1b = data1b[train_indices1b, ]
test_data1b     = data1b[-train_indices1b, ]

inputs1b      = as.matrix(data.frame(lapply(training_data1b[,c(2:4)], as.numeric)))
targets1b     = training_data1b$Class
inputsTb       = as.matrix(data.frame(lapply(test_data1b[,c(2:4)], as.numeric)))
targetsTb      = test_data1b$Class

##-----use linear classification to separate the two classes----##
input_dim1b = 3
output_dim  = 1
inputs1B  = tensorflow::as_tensor(scale(inputs1b), dtype = "float32")
inputsTB = tensorflow::as_tensor(scale(inputsTb), dtype = "float32")
learningrate = c(0.01, 0.05, 0.1, 0.15, 0.2)
accuracy_results1B = data.frame(matrix(nrow = length(learningrate), ncol = 4))
colnames(accuracy_results1B) <- c("learningrate", "accuracy", "step","square_loss")

for (ll in 1:length(learningrate)) {
  accuracy_results1B[ll, ] <- myfun(train_x = inputs1B, train_y = as.matrix(targets1b),
                                    val_x = inputsTB, val_y = targetsTb, lr = learningrate[ll])
}
accuracy_results1B
```

```
##   learningrate  accuracy step square_loss
## 1         0.01 0.8700265  128  0.08044979
## 2         0.05 0.9416446   36  0.04776814
## 3         0.10 0.9602122   19  0.04339608
## 4         0.15 0.9761273   15  0.04172541
## 5         0.20 0.9761273   13  0.04118111
```

the performance of the learning rates also tends to improve as learning rate become larger such as 1(a)i, but the accuracy is not good as 1(a)i.

```
### -------Use support vector machines to separate the two data---
inputsQIB      = data.frame(lapply(training_data1b[,c(2:4)], as.numeric))
names(inputsQIB) = NULL

testdata1b =  data.frame(lapply(test_data1b[,c(2:4)], as.numeric))
```

```
testdata1b = data.frame(scale(testdata1b))
names(testdata1b) = NULL
testdata1b = data.frame(x=testdata1b,class=as.factor(test_data1b$Class))

kernelX = c("linear","radial","polynomial","sigmoid")
result1linear = data.frame(matrix(nrow = length(kernelX), ncol = 2))
colnames(result1linear) <- c("kernel", "accuracy")
for (kk in 1:4) {
  svmfitlinear =  svm(targets~., data = data.frame(x=inputsQIB,targets=as.factor(targets1b)),
                      kernel = kernelX[kk])
  pred_test = table(true = testdata1b$class, pred = predict(svmfitlinear, newdata = testdata1b))
  result1linear[kk,] = c(kernelX[kk], sum(diag(pred_test))/sum(pred_test))
}

result1linear
```

```
##       kernel         accuracy
## 1     linear 0.19893899204244
## 2     radial 0.19893899204244
## 3 polynomial 0.19893899204244
## 4    sigmoid 0.80106100795756
```

```
### -------Use cross-validation to decide on which of these kernels is the best,
## and the cost parameter
kernelX = c("linear","radial","polynomial","sigmoid")
accuracy_rel1b = data.frame(matrix(nrow = length(kernelX), ncol = 3))
colnames(accuracy_rel1b) <- c("kernel", "accuracy", "cost")

for (ll in 1:4) {
  tune_out    = tune(svm,
                     targets~.,
                     data = data.frame(x=inputsQIB,targets=as.factor(targets1b)),
                     kernel = kernelX[ll],
                     ranges = list(cost = c(0.001, 0.01, 0.1, 1,5,10,100)),
                     cachesize = 4096, tolerance = 0.01, tunecontrol = tune.control(cross = 5))
  summary(tune_out)
  bestmod  <- tune_out$best.model
  summary(bestmod)

  class_pred = predict(bestmod, testdata1b)
  Res = table(predicted = class_pred, true = as.factor(testdata1b$class))
  accuracy_rel1b[ll,] = c(kernelX[ll], sum(diag(Res))/ sum(Res),bestmod$cost)
}
accuracy_rel1b
```

```
##       kernel          accuracy cost
## 1     linear  0.19893899204244  0.1
## 2     radial  0.19893899204244    1
## 3 polynomial  0.19893899204244    1
## 4    sigmoid 0.782493368700265  0.1
```

When comparing with 1(a)(ii) and (iii), it appears that a set of five features performs well. Specifically, the radial kernel performs well with these five features, while the sigmoid kernel also performs well in this context.

In conclusion, considering five features performs better than ignoring the two shape variables.

###————————Q2————————###

```
##-----------import data for Q2--------------------##
digit = read.table("D:/iowa state/STAT590s3/hw1/zipdigit.dat", col.names = "class")
zipimg = read.table("D:/iowa state/STAT590s3/hw1/ziptrain.dat") %>% as.matrix()
```

###—-2(a)i—Determine the number of components for 80% variance explained

```
###-----2(a)i--use principal components to reduce dimensionality of the dataset---
# remove mean effect
summary_image_by = function(func){
  summary_img = array(dim = c(10, 256))
  for (i in 0:9){
    summary_img[i+1,] <- apply(zipimg[which(digit == i),], 2, func)
  }
  summary_img
}
mean_img = summary_image_by(mean)
# remove mean effect
mean_effect = array(dim = c(2000, 256))
for (i in 0:9){
  nrows = length(which(digit == i))
  mean_effect[which(digit == i),] <- matrix(rep(mean_img[i+1,], nrows), nrow = nrows, byrow = T)
}

cleaned_zip_img = zipimg - mean_effect
# data = cbind(zipimg, digit) %>% mutate(class = as.character(class))
# data %>% group_by(class) %>% mutate(across(everything(), ~ . - mean(.)))

#PCA
pca_result = prcomp(cleaned_zip_img, center = F)
#calculate total variance explained by each principal component
cumulative_variance = cumsum(pca_result$sdev^2) / sum(pca_result$sdev^2)
match(T, cumulative_variance >= 0.8)
```

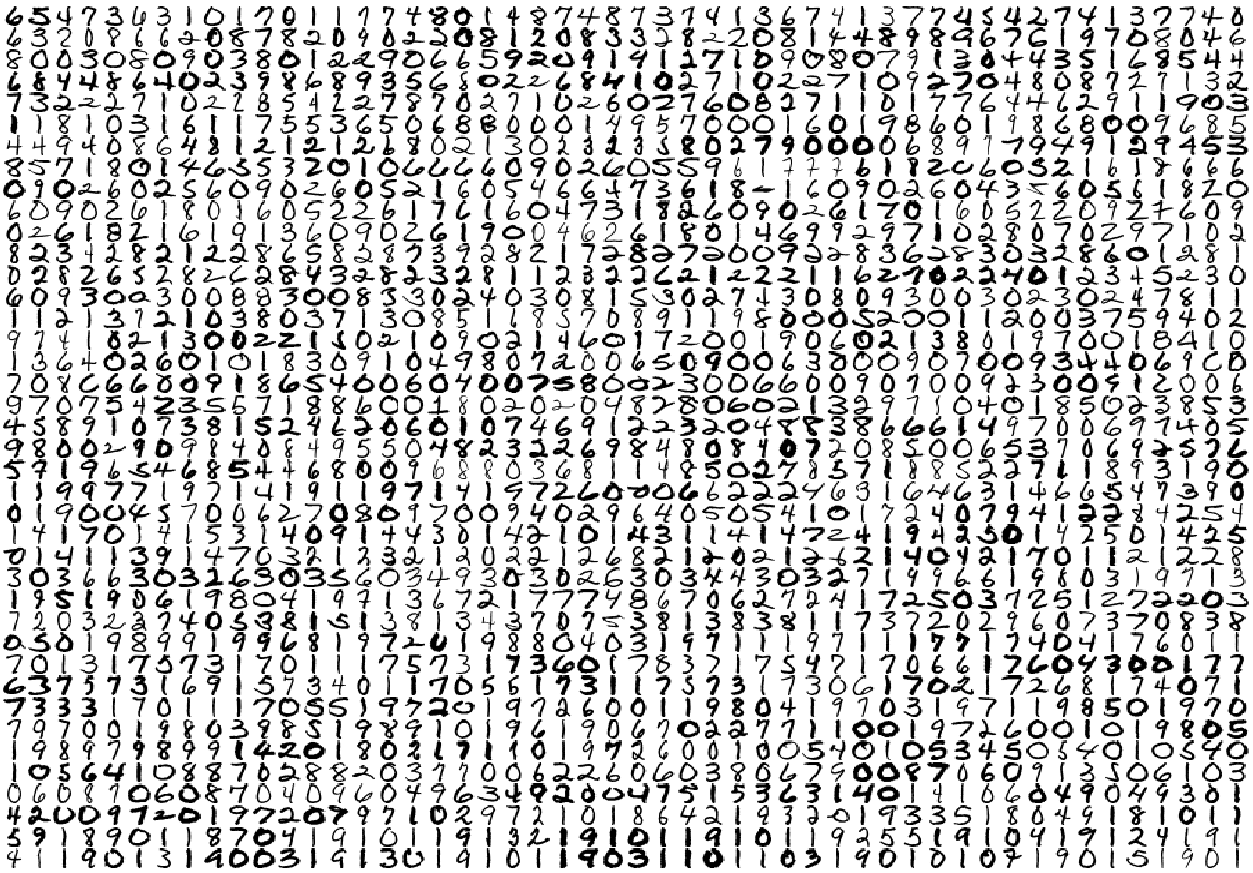
```
## [1] 40
```

```
num_PCs = match(T, summary(pca_result)$importance[3, ] >= 0.8)
#num_PCs
```

the number of components needed to explain at least 80% of the total variance in the data is 40.

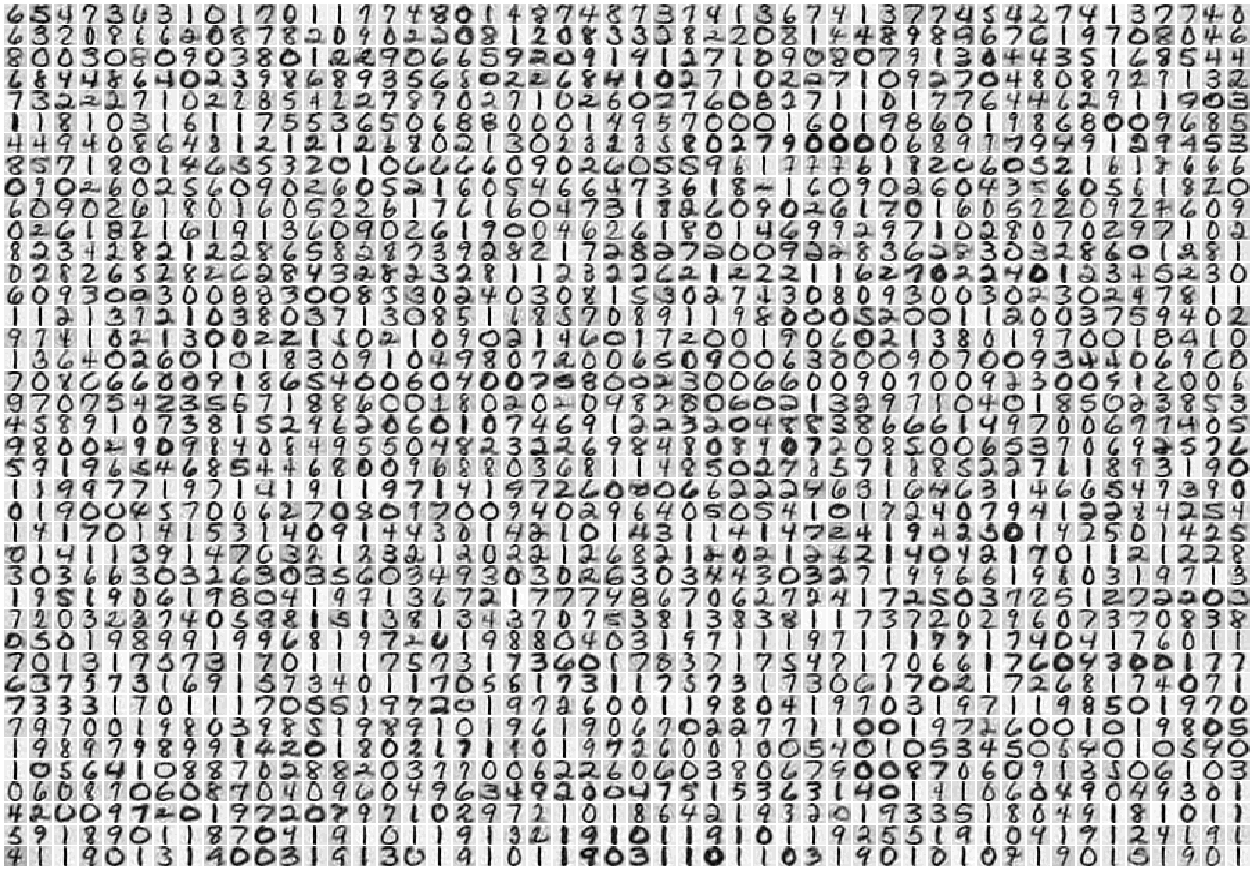
###—2(a)ii—-using PCA is to obtain a lower-rank representation of the images

```
zip_img <- `dim<-`(zipimg, c(2000,16,16))
## image all the 2000 digits in the raw form
par(mar = rep(0.05,4), mfrow = c(40,50))
apply(zip_img, 1, function(x) {image(x[,16:1],
                                     axes = F, col = gray(31:0/31))})
```



```
## NULL
```

```
## image all the 2000 digits in terms of the lower-rank form
newx <- cleaned_zip_img %*% tcrossprod(pca_result$rotation[, 1:num_PCs]) + mean_effect
dim(newx) <- c(2000, 16, 16)
par(mar = rep(0.05,4), mfrow = c(40,50))
apply(newx, 1, function(x) {image(x[,16:1],
                                  axes = F, col = gray(31:0/31))})
```



```
## NULL
```

Digits, when expressed in the lower-rank forma also ppear visually satisfactory.

###————————-2(b)evaluate performance of classification algorithms on the dataset————————##

```
set.seed(123)
mm = nrow(zipimg)
train_indices2b = sample(1:mm, 0.75 * mm)

training_data2b = cleaned_zip_img[train_indices2b, ]
test_data2b    = cleaned_zip_img[-train_indices2b, ]
targets2b      = digit[train_indices2b,]
targetsT2b      = digit[-train_indices2b,]
```

```
##----(a)i---use linear classification to separate the two classes----#

input_dim = 256
output_dim = 10
inputs2b  = tensorflow::as_tensor(training_data2b, dtype = "float32")
inputsT2b = tensorflow::as_tensor(test_data2b, dtype = "float32")
learningrate = c(0.01, 0.05, 0.1, 0.15, 0.2)
# accuracy_results2b <- matrix(NA, length(learningrate), 4,
#                              dimnames = list(NULL, c("learningrate", "accuracy", "step","square_loss"
accuracy_results2b = data.frame(matrix(nrow = length(learningrate), ncol = 4))
colnames(accuracy_results2b) <- c("learningrate", "accuracy", "step","square_loss")

for (ll in 1:length(learningrate)) {
```

```
  accuracy_results2b[ll, ] <- myfun(train_x = inputs2b, train_y = to_categorical(targets2b),
                                    val_x = inputsT2b, val_y = targetsT2b, lr = learningrate[ll])
}
accuracy_results2b
```

```
##   learningrate accuracy step square_loss
## 1         0.01    0.104 1000   1.6107026
## 2         0.05    0.120  539   0.6129438
## 3         0.10    0.092  378   0.4397562
## 4         0.15    0.086  316   0.3818092
## 5         0.20    0.090  271   0.3382872
```

A learning rate of 0.01, 0.05 or 0.1 (sometime is 0.01, or 0,05, sometime is 0.1) appears to yield the best accuracy, but it's notably on the lower side.

```
### -------1(a)(ii)Use support vector machines to separate the two data---
inputsQ2b         = data.frame(cleaned_zip_img[train_indices2b, ])
names(inputsQ2b)  = NULL
testdata2b        =  data.frame(cleaned_zip_img[-train_indices2b, ])
names(testdata2b) = NULL
testdata2b = data.frame(x=testdata2b,class=as.factor(targetsT2b))

kernelX = c("linear","radial","polynomial","sigmoid")
result2blinear = data.frame(matrix(nrow = length(kernelX), ncol = 2))
colnames(result2blinear) <- c("kernel", "accuracy")
for (kk in 1:4) {
  svmfitlinear =  svm(targets~., data = data.frame(x=inputsQ2b,targets=as.factor(targets2b)),
                      kernel = kernelX[kk])
  pred_test = table(true = testdata2b$class, pred = predict(svmfitlinear, newdata = testdata2b))
  result2blinear[kk,] = c(kernelX[kk], sum(diag(pred_test))/sum(pred_test))
}

result2blinear
```

```
##       kernel accuracy
## 1     linear    0.222
## 2     radial    0.942
## 3 polynomial    0.422
## 4    sigmoid      0.1
```

In this scenario, I used the svm() function to separate data using different kernels, while keeping the default cost parameter settings. Based on the results, the radial kernel provides a more accurate and meaningful classification.

```
### -------(iii)Use cross-validation to decide on which of these kernels is the best,
## and the cost parameter
accuracy_rel2b = data.frame(matrix(nrow = length(kernelX), ncol = 3))
colnames(accuracy_rel2b) <- c("kernel", "accuracy", "cost")
for (ll in 1:4) {
  tune_out    = tune(svm,
                     targets~.,
                     data  = data.frame(x = inputsQ2b,targets=as.factor(targets2b)),
                     kernel = kernelX[ll],
                     ranges = list(cost = c(0.001, 0.01, 0.1, 1,10,100)),
                     cachesize = 4096, tolerance = 0.01, tunecontrol = tune.control(cross = 5))
  # summary(tune_out)
```

```
  bestmod  <- tune_out$best.model
  # summary(bestmod)
  class_pred = predict(bestmod, testdata2b)
  Res = table(predicted = class_pred, true = as.factor(testdata2b$class))
  accuracy_rel2b[ll,] = c(kernelX[ll], sum(diag(Res))/ sum(Res),bestmod$cost)
}
accuracy_rel2b
```

```
##       kernel accuracy  cost
## 1     linear    0.296   0.1
## 2     radial    0.958    10
## 3 polynomial    0.638   100
## 4    sigmoid     0.21 0.001
```

In this context, for each decision boundary, I tuned the parameters to compare the best cost associated with them. The results indicate that the nonlinear decision boundary with a radial kernel performs the best when the cost parameter is set to 5.