STAT 542: Homework 6

Shashi Roshan (NetID : sroshan2)

Contents

Question 1 [40 Points] Sovling SVM using Quadratic Programming

Install the `quadprog` package (there are similar ones in Python too) and utilize the function `solve.QP` to solve SVM. The `solve.QP` function is trying to perform the minimization problem:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\boldsymbol{\beta}^T\mathbf{D}\boldsymbol{\beta} - d^T\boldsymbol{\beta} \\ \text{subject to} \quad & \mathbf{A}^T\boldsymbol{\beta} \geq a \end{aligned}$$

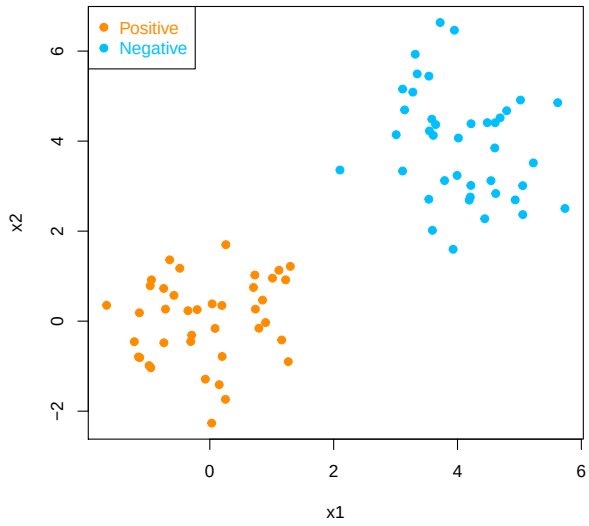
For more details, read the document file of the `quadprog` package on CRAN. You should generate the data using the following code (or write a similar code in Python). For each sub-question, perform the following:

- Propoerly define $\mathbf{D}$, d, A and a.
- Convert the solution into β and β_0, which can be used to define the classification rule
- Plot the decision line, the two margin lines and the support vectors

Note: The package requires $\mathbf{D}$ to be positive definite, while it may not be true in our case. A workaround is to add a "ridge," e.g., $10^{-5}\mathbf{I}$, to the $\mathbf{D}$ matrix, making it invertible. This may affect your results slightly. So be careful when plotting your support vectors.

```
set.seed(3)
n = 40
p = 2
xpos = matrix(rnorm(n * p, mean = 0, sd = 1), n, p)
xneg = matrix(rnorm(n * p, mean = 4, sd = 1), n, p)
x = rbind(xpos, xneg)
y = matrix(c(rep(1, n), rep(-1, n)))

plot(
  x,
  col = ifelse(y > 0, "darkorange", "deepskyblue"),
  pch = 19,
  xlab = "x1",
  ylab = "x2"
)
```

```
legend(
  "topleft",
  c("Positive", "Negative"),
  col = c("darkorange", "deepskyblue"),
  pch = c(19, 19),
  text.col = c("darkorange", "deepskyblue")
)
```



- [20 points] Formulate the SVM **primal** problem of the separable SVM formulation and obtain the solution to the above question.

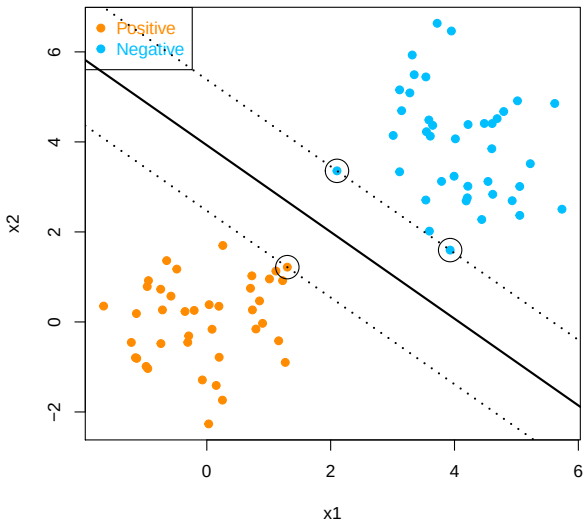
```
library(quadprog)

Dmat = diag(1, 3) + diag(10 ^ -5, 3)

dvec = rep(0, 3)

A = cbind(x, b = rep(1, nrow(x)))
A = sweep(A, MARGIN = 1, y, FUN = '*')
Amat = t(A)

bvec =  rep(1, nrow(x))

sol = solve.QP(
  Dmat = Dmat,
  dvec = dvec,
  Amat = Amat,
  bvec = bvec
)

beta = c(sol$solution[1], sol$solution[2])
beta = matrix(beta, 1, 2)
```

```r
beta_0 = sol$solution[3]

cat('beta_0 : ', beta_0, '\nbeta : ', beta)
```

```
## beta_0 :  2.696011
## beta :  -0.6616875 -0.6865516
```

```r
# beta_0 :  2.696011
# beta :  -0.6616875 -0.6865516
```

```r
# plot data
plot(
  x,
  col = ifelse(y > 0, "darkorange", "deepskyblue"),
  pch = 19,
  xlab = "x1",
  ylab = "x2"
)
legend(
  "topleft",
  c("Positive", "Negative"),
  col = c("darkorange", "deepskyblue"),
  pch = c(19, 19),
  text.col = c("darkorange", "deepskyblue")
)

# decision boundary
abline(
  a = -beta_0 / beta[1, 2],
  b = -beta[1, 1] / beta[1, 2],
  col = "black",
  lty = 1,
  lwd = 2
)

# margins
abline(
  a = (-beta_0 - 1) / beta[1, 2],
  b = -beta[1, 1] / beta[1, 2],
  col = "black",
  lty = 3,
  lwd = 2
)
abline(
  a = (-beta_0 + 1) / beta[1, 2],
  b = -beta[1, 1] / beta[1, 2],
  col = "black",
  lty = 3,
  lwd = 2
)

# support vectors
points(x[abs(sol$Lagrangian) > 5 * 10 ^ -5,], col = "black", cex = 3)
```



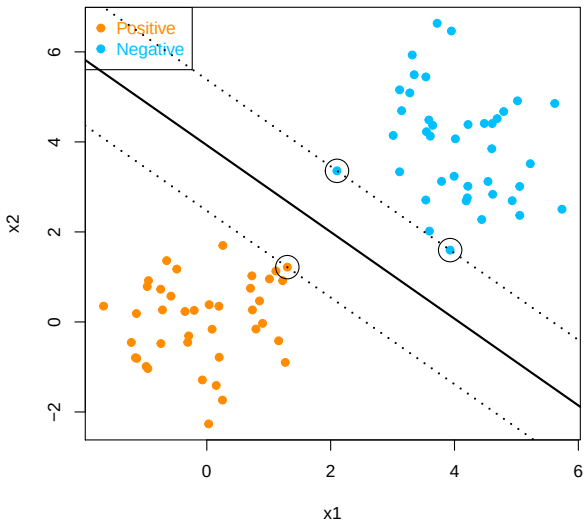
- [20 points] Formulate the SVM **dual** problem of the separable SVM formulation and obtain the solution to the above question.

```
library(quadprog)

n = nrow(x)
XY = t(sweep(x, MARGIN = 1, y, FUN = '*'))

Dmat = t(XY) %*% XY + diag(10 ^ -5, n)

dvec = matrix(1, nrow = n, ncol = 1)

Amat = cbind(matrix(y, nrow = n, ncol = 1), diag(n))

bvec = rbind(matrix(0, nrow = 1, ncol = 1) , matrix(0, nrow = n, ncol =
                                                      1))

sol = solve.QP(
  Dmat = Dmat,
  dvec = dvec,
  Amat = Amat,
  bvec = bvec,
  meq = 1
)

beta = matrix(0, ncol = 2, nrow = 1)
for (i in 1:n) {
  beta = beta + (sol$solution[i] * y[i] * x[i, ])
}
beta_0 = -(max(x[y == -1,] %*% t(beta)) + min(x[y == 1,] %*% t(beta))) / 2

cat('beta_0 : ', beta_0, '\nbeta : ', beta)
```

```
## beta_0 :  2.696001
## beta :  -0.6616855 -0.6865488
```

```r
# beta_0 :  2.696001
# beta :  -0.6616855 -0.6865488
```

```r
# plot data
plot(
  x,
  col = ifelse(y > 0, "darkorange", "deepskyblue"),
  pch = 19,
  xlab = "x1",
  ylab = "x2"
)
legend(
  "topleft",
  c("Positive", "Negative"),
  col = c("darkorange", "deepskyblue"),
  pch = c(19, 19),
  text.col = c("darkorange", "deepskyblue")
)

# decision boundary
abline(
  a = -beta_0 / beta[1, 2],
  b = -beta[1, 1] / beta[1, 2],
  col = "black",
  lty = 1,
  lwd = 2
)

# margins
abline(
  a = (-beta_0 - 1) / beta[1, 2],
  b = -beta[1, 1] / beta[1, 2],
  col = "black",
  lty = 3,
  lwd = 2
)
abline(
  a = (-beta_0 + 1) / beta[1, 2],
  b = -beta[1, 1] / beta[1, 2],
  col = "black",
  lty = 3,
  lwd = 2
)

# support vectors
points(x[abs(sol$solution) > 5 * 10 ^ -5, ], col = "black", cex = 3)
```



Question 2 [40 Points] Sovling SVM using Penalized Loss Optimization

Consider the logistic loss function,

$$L(y, f(x)) = \log(1 + e^{-yf(x)}).$$

The rest of the job is to solve this optimization problem if given the functional form of $f(x)$. To do this, we will utilize a general-purpose optimization package/function. For example, in `R`, you can use the `optim` function. Read the documentation of this function (or equivalent ones in Python) and set up the objective function properly to solve for the parameters. If you need an example of how to use the `optim` function, read the corresponding part in the example file provided on our course website here (Section 10).

- [20 points] Linear SVM. When $f(x)$ is a linear function, SVM can be solved by optimizing the penalized loss:

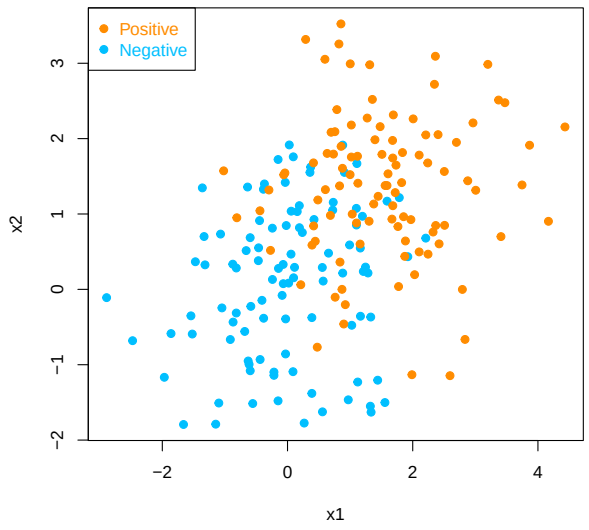
$$\arg\min_{\beta_0,\beta} \sum_{i=1}^{n} L(y_i, \beta_0 + x_i^T\beta) + \lambda\|\beta\|^2$$

  You should generate the data using the code provided below code (or write similar code in Python), and answer these questions:

  - Write a function to define the penalized loss objective function. The algorithm can run faster if you further define the gradient function. Hence, you should also define the gradient function properly and implement it in the optimization.
  - Choose a reasonable λ value so that your optimization can run properly. In addition, I recommend using the `BFGS` method in optimization.
  - After solving the optimization problem, plot all data and the decision line
  - If needed, modify your λ so that the model fits reasonably well and re-plot. You don't have to tune λ as long as you obtain a reasonable decision line.

```
set.seed(20)
n = 100 # number of data points for each class
p = 2 # dimension
```

```r
# Generate the positive and negative examples
xpos = matrix(rnorm(n * p, mean = 0, sd = 1), n, p)
xneg = matrix(rnorm(n * p, mean = 1.5, sd = 1), n, p)
x = rbind(xpos, xneg)
y = c(rep(-1, n), rep(1, n))

plot(
  x,
  col = ifelse(y > 0, "darkorange", "deepskyblue"),
  pch = 19,
  xlab = "x1",
  ylab = "x2"
)
legend(
  "topleft",
  c("Positive", "Negative"),
  col = c("darkorange", "deepskyblue"),
  pch = c(19, 19),
  text.col = c("darkorange", "deepskyblue")
)
```



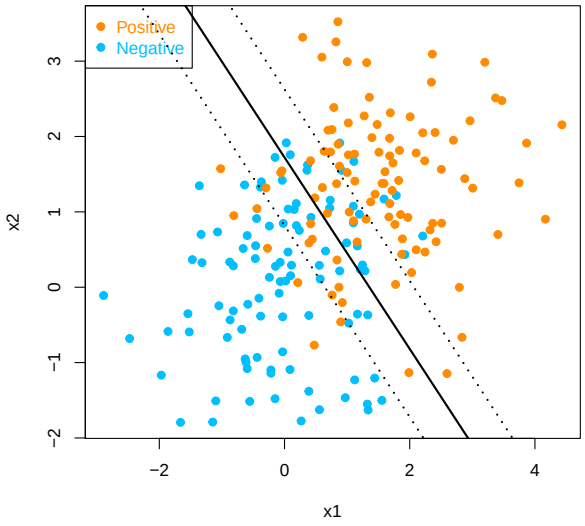

```r
loss <- function(x, y, beta, lambda)
{
  b0 = beta[1]
  b = beta[-1]
  x_beta = b0 + (x %*% b)
  sum = 0
  for (i in 1:length(y)) {
    sum = sum + log(1 + exp(-y[i] * (x_beta[i])))
  }
  beta_norm = sqrt(sum(b ^ 2))
  return(sum + lambda * beta_norm)
```

```
}

gradient <- function(beta, x, y, lambda) {
  b0 = beta[1]
  b = beta[-1]
  fx = apply(x, MARGIN = 1, function(x) b0 + (x %*% b))
  p = - (x * y)
  return(c(sum(
    (1 + exp(-y * fx)) ^ -1 * exp(- y * fx ) * -(y)),
  (2 * lambda * b) + colSums((1 + exp(-y * fx)) ^ -1 * exp(- y * fx ) * (p))))
}

solution = optim(
  matrix(rep(0, 3)),
  gr=gradient,
  loss,
  x = x,
  y = y,
  lambda = 0.5,
  method = "BFGS"
)
solution$par
```

```
##           [,1]
## [1,] -1.907585
## [2,]  1.407150
## [3,]  1.107509
```

```
# plot the results
plot(
  x,
  col = ifelse(y > 0, "darkorange", "deepskyblue"),
  pch = 19,
  xlab = "x1",
  ylab = "x2"
)
legend(
  "topleft",
  c("Positive", "Negative"),
  col = c("darkorange", "deepskyblue"),
  pch = c(19, 19),
  text.col = c("darkorange", "deepskyblue")
)

b = matrix(solution$par[2:3], ncol = 2)
b0 = solution$par[1]

# decision boundary
abline(
  a = -b0 / b[1, 2],
  b = -b[1, 1] / b[1, 2],
  col = "black",
  lty = 1,
```

```
  lwd = 2
)

# plotting the margin
abline(
  a = (-b0 - 1) / b[1, 2],
  b = -b[1, 1] / b[1, 2],
  col = "black",
  lty = 3,
  lwd = 2
)

abline(
  a = (-b0 + 1) / b[1, 2],
  b = -b[1, 1] / b[1, 2],
  col = "black",
  lty = 3,
  lwd = 2
)
```



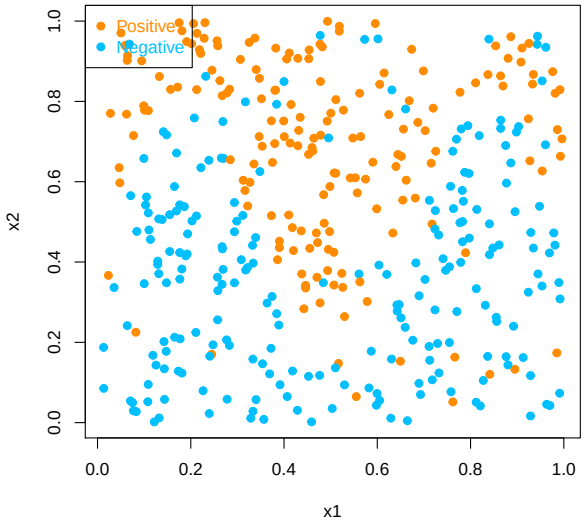
- [20 points] Non-linear SVM. You should generate the data using the code provided below code (or write similar code in Python), and answer these questions:
  - Use the kernel trick to solve for a nonlinear decision rule. The penalized optimization becomes

$$\sum_{i=1}^{n} L(y_i, K_i^T \beta) + \lambda \beta^T K \beta$$

    where K_i is the ith column of the $n \times n$ kernel matrix K. For this problem, we consider the Gaussian kernel. For the bandwidth of the kernel function, you can borrow ideas from our previous homework. You do not have to tune the bandwidth.
  - Pre-calculate the $n \times n$ kernel matrix K of the observed data.
  - Write a function to define the objective function. You should also define the gradient function properly and implement it in the optimization.

- Choose a reasonable λ value so that your optimization can run properly.
- It could be difficult to obtain the decision line itself. However, its relatively easy to obtain the fitted label. Hence, calculate the fitted label (in-sample prediction) of your model and report the classification error?
- Plot the data using the fitted labels. This would allow you to visualize (approximately) the decision line. If needed, modify your λ so that the model fits reasonably well and re-plot. You don't have to tune λ as long as your result is reasonable. You can judge if your model is reasonable based on the true model.

```r
set.seed(1)

n = 400

p = 2

# Generate the positive and negative examples
x = matrix(runif(n * p), n, p)
side = (x[, 2] > 0.5 + 0.3 * sin(3 * pi * x[, 1]))
y_ = sample(c(1,-1), n, TRUE, c(0.9, 0.1)) * (side == 1) + sample(c(1,-1), n, TRUE, c(0.1, 0.9)) *
  (side == 0)

plot(
  x,
  col = ifelse(y_ > 0, "darkorange", "deepskyblue"),
  pch = 19,
  xlab = "x1",
  ylab = "x2"
)
legend(
  "topleft",
  c("Positive", "Negative"),
  col = c("darkorange", "deepskyblue"),
  pch = c(19, 19),
  text.col = c("darkorange", "deepskyblue")
)
```



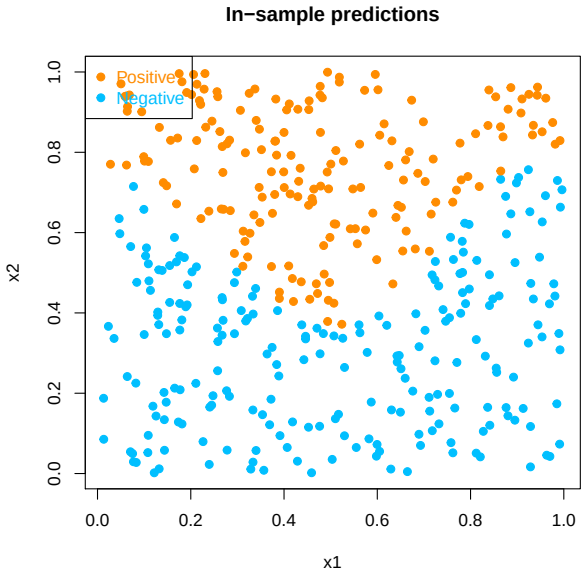

```
n = nrow(x)

gradient <- function(beta, k, y, lambda)
{
  sum = 0
  for (i in 1:length(y)) {
    fx = t(k[, i]) %*% beta
    sum = sum - (as.numeric(exp(-y[i] * fx)) * (as.matrix(k[, i]) %*% as.matrix(y[i,]))) /
      as.numeric((1 + exp(-y[i] * (fx))))
  }
  return(sum + (2 * lambda) * t((t(beta) %*% (t(k) + k))))
}

kernel_loss <- function(k, y, beta, lambda)
{
  sum = 0
  for (i in 1:length(y)) {
    sum = sum + log(1 + exp(-y[i] * (t(k[, i]) %*% beta)))
  }
  beta_norm = t(beta) %*% k %*% beta
  return(sum + lambda * beta_norm)
}

bandwidth = n ^ (-1 / 6)
kernel <- function(x_i, x_j) {
  return(exp((-1 / (2 * bandwidth)) * sum((x_i - x_j) ^ 2)))
}

K = matrix(0, n, n)
for (i in 1:n) {
  for (j in 1:n) {
    K[i, j] = kernel(x[i, ], x[j, ])
  }
```

```
}

y = as.matrix(y_, 400, 1)

solution = optim(
  matrix(rep(0, nrow(x))),
  gr = gradient,
  kernel_loss,
  k = K,
  y = y,
  lambda = 0.004,
  method = "BFGS"
)

predictions = ifelse( (t(K) %*% solution$par) >= 0 , 1, -1)

library(caret)
confusionMatrix(as.factor(predictions), as.factor(y))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  -1   1
##         -1 187  34
##         1   35 144
##
##                Accuracy : 0.8275
##                  95% CI : (0.7868, 0.8632)
##     No Information Rate : 0.555
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.651
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.8423
##             Specificity : 0.8090
##          Pos Pred Value : 0.8462
##          Neg Pred Value : 0.8045
##              Prevalence : 0.5550
##          Detection Rate : 0.4675
##    Detection Prevalence : 0.5525
##       Balanced Accuracy : 0.8257
##
##        'Positive' Class : -1
##
```

```
plot(
  x,
  col = ifelse(predictions == 1, "darkorange", "deepskyblue"),
  pch = 19,
  xlab = "x1",
  ylab = "x2",
```

```
  main = "In-sample predictions"
)
legend(
  "topleft",
  c("Positive", "Negative"),
  col = c("darkorange", "deepskyblue"),
  pch = c(19, 19),
  text.col = c("darkorange", "deepskyblue")
)
```



Question 3 [20 Points] Gradient Boosting using `xgboost`

We will use the handwritten digit recognition data again from the `ElemStatLearn` package. We only consider the train-test split, with the pre-defined `zip.train` and `zip.test`. However, **use `zip.test` as the training data, and `zip.train` as the testing data!** For this question, we again only consider the two digits: 2 and 4. We will use cross-validation to choose the best tuning. For this question, use the `xgboost` package (available in both R and Python), and consider two tuning parameters when you perform cross-validation (using their built-in functions): the base learner (linear or tree), tree depth (for trees only) and the learning rate. Choose a reasonable range of tunings and use the area under the ROC curve (AUC) as the criteria to select the best tuning. Report the prediction accuracy of your final model.

```
# Handwritten Digit Recognition Data
library(ElemStatLearn)

# this is the training data!
dim(zip.test)
```

```
## [1] 2007  257
```

```
train = zip.test

# this is the testing data!
dim(zip.train)
```

```
## [1] 7291  257
```

```
test = zip.train

train = train[(train[,1]==2) | (train[,1]==4), ]

test = test[(test[,1]==2) | (test[,1]==4), ]

# convert response to 0 or 1, since xgboost expects it in that form
train[, 1] = ifelse(train[, 1]==2, 0, 1)
test[, 1] = ifelse(test[, 1]==2, 0, 1)
```

```
library(xgboost)

# variables to store maximum value of test accuracy, and optimal number of rounds
max_test_auc_for_all_parameters = -999
optimal_rounds = -999

# cross validation for linear learner
set.seed(542)
xgbcv = xgb.cv(
  params = list(booster = 'gblinear'),
  data = train[, 2:ncol(train)],
  label = train[, 1],
  metrics = 'auc',
  objective = 'binary:logistic',
  nrounds = 100,
  nfold = 10,
  verbose = FALSE
)

max_test_auc_for_all_parameters = max(xgbcv$evaluation_log$test_auc_mean)
optimal_rounds = which.max(xgbcv$evaluation_log$test_auc_mean)

# cross validation for for tree learner

# grid of hyperparameters
max_depth_values = seq(4, 20, by = 1)
eta_values = seq(0, 0.5, by = 0.05)

optimal_max_depth = -999
optimal_eta = -999

# CV for tree learner
for (max_depth_value in max_depth_values) {
  for (eta_value in eta_values) {

    params = list(booster = 'gbtree',
```

```
                  max_depth = max_depth_value,
                  eta = eta_value)

    set.seed(542)
    xgbcv = xgb.cv(
      params = params,
      data = train[, 2:ncol(train)],
      label = train[, 1],
      metrics = 'auc',
      objective = 'binary:logistic',
      nrounds = 100,
      nfold = 10,
      verbose = FALSE
    )

    max_test_auc = max(xgbcv$evaluation_log$test_auc_mean)

    if (max_test_auc > max_test_auc_for_all_parameters) {
      max_test_auc_for_all_parameters = max_test_auc
      optimal_rounds = which.max(xgbcv$evaluation_log$test_auc_mean)
      optimal_max_depth = max_depth_value
      optimal_eta = eta_value
    }

  }
}

max_test_auc_for_all_parameters
```

```
## [1] 0.993271
```

```
optimal_max_depth
```

```
## [1] 6
```

```
optimal_eta
```

```
## [1] 0.45
```

```
optimal_rounds
```

```
## [1] 28
```

```
# Train xgboost model on train data using optimal learner and set of parameters
set.seed(542)
xgb_model = xgboost(
    params = list(
      booster = 'gbtree',
      max_depth = optimal_max_depth,
      eta = optimal_eta
    ),
```

```
    data = train[, 2:ncol(train)],
    label = train[, 1],
    metrics = 'auc',
    objective = 'binary:logistic',
    nrounds = optimal_rounds,
    nfold = 10,
    verbose = FALSE
  )

# Find AUC on test data
predictions = predict(xgb_model, test[, 2:ncol(train)])
predictions = ifelse (predictions > 0.5, 1, 0)
cat('Test set accuracy : ', mean(predictions == test[, 1]))
```

```
## Test set accuracy :  0.9645698
```

```
library(caret)
confusionMatrix(as.factor(predictions),  as.factor(test[, 1]), positive = '1')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 709  27
##          1  22 625
##
##                Accuracy : 0.9646
##                  95% CI : (0.9534, 0.9737)
##     No Information Rate : 0.5286
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9289
##
##  Mcnemar's Test P-Value : 0.5677
##
##             Sensitivity : 0.9586
##             Specificity : 0.9699
##          Pos Pred Value : 0.9660
##          Neg Pred Value : 0.9633
##              Prevalence : 0.4714
##          Detection Rate : 0.4519
##    Detection Prevalence : 0.4678
##       Balanced Accuracy : 0.9642
##
##        'Positive' Class : 1
##
```

```
# Test set accuracy :  0.9681851
```