Data Mining- Practice 6 Classification- Support Vector machines and Kernel Trick

Rajesh Kalakoti[§], Sven Nomm[*]
[*] Taltech, Estonia, 12616
[§] Email: rajesh.kalakoti@outlook.com

***Abstract*—In today's practical session, we delve into the Support Vector Machines (SVMs) and the ingenious technique known as the kernel trick. SVMs, a class of supervised machine learning algorithms, are proficient in both classification and regression tasks.**

I. Support Vector Machine

Support Vector Machines, commonly referred to as SVMs, are a class of supervised machine learning algorithms designed for both classification and regression tasks. They have gained widespread popularity due to their ability to handle complex and high-dimensional data with remarkable efficiency. SVMs are particularly adept at solving binary classification problems. At the heart of SVMs is the concept of finding the optimal hyperplane that best separates two classes within a dataset. This hyperplane is strategically positioned to maximize the margin, which is the distance between the hyperplane and the nearest data points of each class. The data points closest to the hyperplane are known as "support vectors."

SVMs are attractive for their ability to work well in scenarios where data is not linearly separable. To address non-linear separability, SVMs leverage a remarkable technique called the "kernel trick." This method allows SVMs to implicitly map data into a higher-dimensional space, where a linear separation becomes feasible. Various kernel functions, such as polynomial kernels and radial basis function (RBF) kernels, are employed to perform this transformation efficiently. The kernel trick enhances the SVM's capacity to model complex, non-linear relationships in the data. if you want to implement the SVM from scratch in python, you can check here [1].

```
library(klaR)
```

```
## Loading required package: MASS
```

```
library(caret)
```

```
## Loading required package: ggplot2
```

```
## Loading required package: lattice
```

```
library(stringr)
library(here)
```

```
## here() starts at /home/rajeshkalakoti/Document
```

```
bc_file_path = here("data",
                          "breast_cancer_data.csv
db_file_path = here("data",
                          "diabetes_data.csv")

heart_fp= here("data",
                     "heart.csv")
bc_data = read.csv(bc_file_path, header = TRUE)
db_data = read.csv(db_file_path,header = TRUE)
heart_data = read.csv(heart_fp,header = TRUE)
```

II. Gradient descent implementation

A. Overview

In this part we will be creating our own support vector machine stochastic gradient descent with Barzilai-Borwein step size (SVM SGD-BB). As there is little chance our data is separable, we will be using a soft-margin classifier. Furthermore, hard-margin classifier (also called maximal margin classifier) is much more sensitive to observations close to the choice boundary, meaning it can more easily overfit our data.

We will first explain the optimization problem, then go through the method of finding an optimum, and walk you through the code step by step. Before we run the our code on the heart dataset we will test our implementation on a few simulated datasets. Then we will finally run our code on the data set and see how well it compares to other models.

B. Support Vector Optimization Problem

In this implementation we will focus on linear svm, therefore we can represent our function as:

$$\hat{f}(X) = \hat{\beta}_0 + \hat{\beta_1} x_1 + \hat{\beta}_2 x_2 + ... + \hat{\beta}_n x_p$$

Where we classify an object as:

$$\hat{y} = -1 for \hat{f}(X) < 0 \hat{y} = 1 for \hat{f}(X) >= 0$$

To greatly simply our calculations when running an algorithm we will present our calculations in a matrix notation:

$$\hat{f}(X) = X\hat{\beta}$$

Many sources represent bias (also called constant term) as a separate variable b but throughout our calculations we will include the term as a first entry in the vector of coefficients B, and instead add a column of 1's to any data we pass to the algorithm. This should allow our algorithm to be more scalable when we use our algorithm for higher dimensional problems later on.

Support vector classifier is a solution to a minimization problem [2]:

$$\text{max: } M \text{subject to: } \|\beta\|_2 = 1 y_i(x_i^T\beta) \geq M(1-\epsilon_i),$$
$$\text{for all } i \sum_{i=1}^{n} \epsilon_i \leq K, \epsilon_i \geq 0, \text{ for all } i$$

M represents one-sided distance of the decision boundry to the margin[2]. As this distance can be represented as second norm B^-1, instead of maximizing the margin M (like in the previous formulation) we can rop our condition on 2nd beta norm squared equal to one, and replacing M with it.

$$M = 1/\|\beta\|$$

Epsilon represents the relative distance to our decision boundary, where $1 >= e_i >= 0$ implies observation i has been correctly classified but lies inside the margin M, whereas $e_i >=$ implies our observation has been miss classified. By putting an upper bound on the sum of e_i's let's say K, we limit the amount of possible missclasifications.

$$\sum_{i=1}^{n} \epsilon_i <= K$$

For K = 0 we do not allow any missclasifications, which is equivalent to a the maximum margin classifier. Instead of putting an upper bound of K we again reformulate it as a sum of epsilons scaled by a factor C. Our, factor C will be inversely proportional to K, meaning for a separable case C -> inf.

We can now re-arrange our optimization problem as [3]

$$min \frac{1}{2}\|\beta\|_2^2 + C\sum_{i=1}^{n} \epsilon_i$$
$$subjectto: \epsilon_i \geq 0, y_i({x_i}^T\beta) \geq 1-\epsilon_i, foralli$$

This is a more computationally convenient representation which we will rely on, but it needs a few more modifications before we can apply it:

$$\epsilon_i \geq 1 - y_i({x_i}^T\beta)\epsilon_i \geq 0, foralli$$

We can now put those 2 conditions to get:

$$\epsilon_i \geq max(0, 1 - y_i({x_i}^T\beta))$$

This will represent our hinge SVM hinge loss [2][3] which we can put into our final minimization problem.

We can finally write up the function we will be minimizing, let's denote it as h(B):

$$h(\beta) = \frac{1}{2}\|\beta\|_2^2 + C\sum_{i=1}^{n} max(0, 1 - y_i({x_i}^T\beta))$$

This is the typical form in which our optimization problem is presented [4]. The left side represents the regularization parameter, that maximizes the margin M. The right side is the aforementioned loss function, in our case the hinge loss, that sums up the severity of classifications.

One interesting thing to note is that some resources[1], re-write this equation in the following form:

$$h(\beta) = \frac{1}{2}\ \lambda\|\beta\|_2^2 + \sum_{i=1}^{n} max(0, 1 - y_i({x_i}^T\beta))$$

This formulation resembles the ridge regression with parameter lambda penalizing the coefficients. This formulation is essentially the same function where lambda is inversely proportional to C parameter.

In this case we will be using the first version of the formulation.

C. *Gradient Descent method and derivations*

The aforementioned optimization has solution using Lagrange multipliers with KKT (Karush–Kuhn–Tucker) conditions [2][6]. This can be solved using quadratic programming, but such algorithms are very time consuming[5].

Instead we will be applying a gradient descent algorithm. This algorithm uses a simple trick that the gradient of our optimization problem will, at a given point x_i, point in the direction of steepest ascent. Conversely, taking a step in the opposite direction will lead us towards the minimum. In the simplest form, the gradient works in the following way.

1. Initialize a set of values values Beta (can be all 0 or a set of random values)
2. Calculate the gradient of our minimization function h(Beta) with respect to Beta
3. Update Beta, subtracting Gradient times learning rate
4. Repeat, until some criterion is reached.

Before we explain the learning rate and stochastic gradient descent let us first calculate the gradient we will be using. Sadly, the function is not differential due to the max() condition, but we can find gradients for both outcomes of the condition

$$G(\beta, X, Y) = \nabla_\beta h(\beta, X, Y) = \begin{cases} \beta - Cy_ix_i & \text{if } 1 - y_i(x_i^T\beta) > 0 \\ \beta & \text{else} \end{cases}$$

Gradient descent requires calculating the gradient for all observations X. As this can be time consuming we should use stochastic gradient descent (SGD). SGD at each new iteration of GD algorithm takes a sample of size S and calculates the gradient for that particular sample. SGD also has one more advantage over SG. We know that both at the local extremum and a saddle point, the gradient of our function will be equal to 0. This means our algorithm can terminate after getting stuck at a local saddle point, without finding an actual minimum we're looking for.

Therefore we shall transform our initial GD algorithm to take a sample S of size s at each iteration and calculate the gradient according to:

$$G(\beta, X, Y) = \beta - C * \frac{1}{s}[\sum_{i=1}^{S} y_i x_i] for 1 - y_i(x_i^T \beta) > 0, i \in S$$

Finally, we need to consider the learning rate, which will dictates how large the steps we will take towards the direction of our minimum[8]. Although, it seems as it could be set to an absolute value, this would require optimizing the parameter. Too large steps could mean our algorithm will jump around the minimum not really ever reaching it. Too small steps and it might take large amount of iterations to converge to a minimum[9]. To solve this we shall be using a Barzilai–Borwein (BB) method[8]. Although, it might seem problematic to use BB method for stochastic setting as our algorithm never calculates the entire gradient, paper proposed by K.Sopyla and P.Drozda[10], finds SGD-BB performs equally well as other similar learning rates, and has a much lower sensitivity to the choice of our intitial parameters.

The Barzilai–Borwein method for learning rate (variable nn in code) is:

$$\eta_n = \frac{|(\beta_t - \beta_{t-1})^T (G_t - G_{t-1})|}{\|G_t - G_{t-1}\|_2^2}$$

Where t is our current iteration of SGD algorithm.

D. *Code explained*

```
svm_gradient_descent <- function (
    x,y,C,S = 10,t = 1000,bias = TRUE) {
    #B <- coeffiient matrix
    #t <- number of interations
    #nn <- learning rate
    #H <- vector of h hinge loss
  #value for each iteration
    #G <- gradient
    #b <- whether to include bias
  #term (also called constant)
    #V <- vector of saved v
  #alues y or 0 for max(0,1-Yi(BXi)
    #S <- Sample size for SGD
    #XS, YS <- Sample of XS,YS of size S

    #Adds a row of 1's if constant is true
    if (bias == TRUE) {
      Dummy1 <- matrix(1,nrow(x),1)
      x <- cbind(Dummy1,x)
    }

    #Changes TRUE / FALSE to -1 and 1
  #and turns y into a data type matrix
    Y <- data.matrix(ifelse(y == TRUE,1,-1))

    #Makes sure x is of data type matrix
    X <- data.matrix(x)

    #Initializes coefficient & gradient matrices
    B <- matrix(0,ncol(x),1)
    G <- matrix(0,ncol(x),1)

    for (i in 1:t) {
      #Sets previous
      #gradient to be current gradient
      Gp <- G

      #Sampling without replacement
      Sample <- sample(nrow(X),S)
      XS <- as.matrix(X[Sample,])
      YS <- as.matrix(Y[Sample,])
      V <- matrix(0,nrow(YS),1)

      #Part(1)
      #max(0,1-yi(xiB)) part of the algorithm,
      #saving output in V
      for (n in c(1:nrow(XS))) {
        #Main part of the max(0,1-yi(Bxi) equati
        V[n] <- ifelse((1 - (XS[n,] %*% B)*YS[n])
                       YS[n],0)
      }

      #Part(2)
      #updating the gradient using V,B and C
      for (j in c(1:ncol(XS))) {
        G[j] <- B[j]- (C*((t(V)%*%XS[,j])/nrow(XS
      }

      #Part(3)
      #Barzilai-Borwein Step Size
      # for i> iterations (fixed for i==1)
      if (i== 1) {
        nn = 1/1000
      } else {
        numerator = abs(t(B-Bp)%*%(G-Gp))
        denominator = (t(G-Gp)%*%(G-Gp))
        nn = as.vector(numerator/denominator)
      }

      #Part(4)
      #Checks whether the value of B is NaN
      # and prevents further iterations
      #if TRUE
      if (is.na(nn)) {
        warning(c("Terminated after ",
                  i," iterations"))
        predicted <- ifelse(X%*%Bp<0,-1,1)
        error_rate <- sum(ifelse(
          predicted ==Y,0,1))/nrow(Y)
        M <- 1/(sqrt(t(Bp)%*%Bp))

        svm_output <- list(coef = Bp,
                  error_rate = error_rate,
                  margin = M)
        class(svm_output) <- "svm_sgd_bb"
        return(svm_output)
      }
      #Sets previous coefficients
      # to the current
      # coefficients, before updating
      Bp <- B

      #Takes the step in
      #the direction of minimum
      B <- B-nn*G
    }
    predicted <- ifelse(X%*%B<0,-1,1)
    error_rate <- sum(ifelse(
      predicted ==Y,0,1))/nrow(Y)
    M <- 1/(sqrt(t(B)%*%B))

    svm_output <- list(coef = B,
                  error_rate = error_rate,
                  margin = M,
                  gradient=G)
    class(svm_output) <- "svm_sgd_bb"
    return(svm_output)
  }
```

1) *Part-1:* This part will be checking whether a given variable is a support vector. Instead of saving the variables as 1's and 0's, in order to simplify calculations of Part(2), we will instead save y_i's and 0's. We save our support vectors as y_i's, as in part(2) we will be multiplying matrices by the entire set XS, giving us output of either 0 (if not a support vector) or $x_i * y_i$ if it is.

2) *Part 2:* We simply update our gradient according to the derivations from 2.1.

3) *part 3:* In this part we set our learning rate. In order to apply BB, we of course need previous set of coefficients as well as gradients, hence for 1st iteration of algorithm we will set a fixed value of nn=0.001

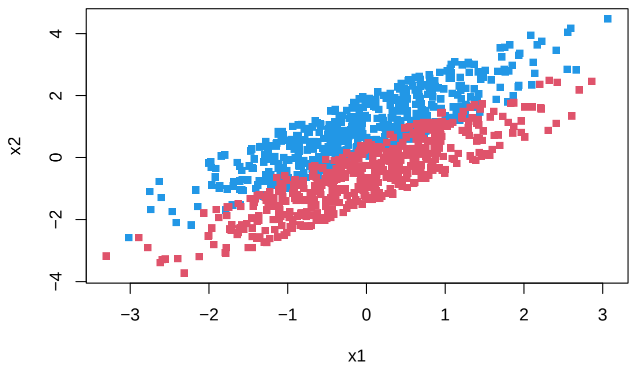
4) *part 4:* There might be a situation were our algorithm runs into a saddle point or happens to reach minimum faster then we initially expected (or for some other unexpected reason Gp == G). The former can especially happen for very low values of S. In either case if G == Gp our BB algorithm will return a value NaN, as it will be trying to solve 0/0 and conversely update our B vector for the rest of the iterations. To prevent this we will terminate the loop faster, display a warning to the user that we have not reached all of our desired iterations, and give an output we would otherwise do, but for the previous loop.

E. Test our Code.

Let us create two data sets,1st linearly separable (X1), the other not (X2) [11]:

```
set.seed(80)
n = 500
a1 = rnorm(n)
a2 = 1 + a1 + 2* runif(n)
b1 = rnorm(n)
b2 = -1 + b1 - 2*runif(n)
X1 = rbind(matrix(cbind(a1,a2),,2),
           matrix(cbind(b1,b2),,2))

a1 = rnorm(n)
a2 =  - a1 + 2* runif(n)
b1 = rnorm(n)
b2 = -0.5 - b1 - 2*runif(n)
X2 = rbind(matrix(cbind(a1,a2),,2),
           matrix(cbind(b1,b2),,2))
Y <- matrix(c(rep(1,n),rep(-1,n)))

remove(a1,a2,b1,b2,n)
```

```
set.seed(NULL)
n = 500
a1 = rnorm(n)
a2 = a1 + 2* runif(n)
b1 = rnorm(n)
b2 = 0.5 + b1 - 2*runif(n)
XXX = rbind(matrix(cbind(a1,a2),,2),
            matrix(cbind(b1,b2),,2))
YYY <- matrix(c(rep(1,n),rep(-1,n)))
plot(XXX,col=ifelse(YYY>0,4,2),pch=".",
     cex=7,xlab = "x1",ylab = "x2")
```



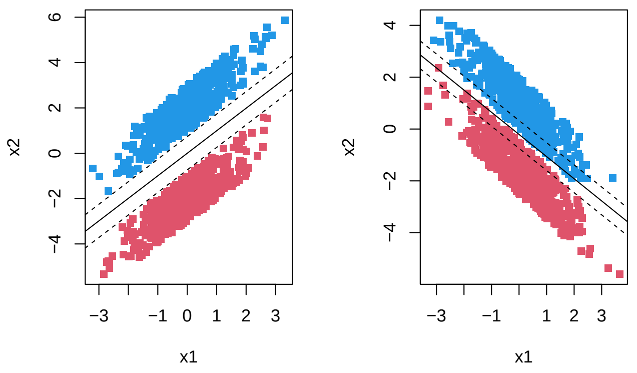
Now let us check our models for 2 levels of C (high and low). We will also increase the amount of iterations to make sure our model reaches a minimum, and set the sample to be about 1/3 of our dataset.Before running our function we will unset the seed for our random sampling to work correctly:

```
set.seed(NULL)
Test1 <- svm_gradient_descent(X1,Y,
                               C=100,
                               t=2000,
                               S=300)
Test2 <- svm_gradient_descent(X1,Y,
                               C=0.1,
                               t=2000,S=300)
Test3 <- svm_gradient_descent(X2,Y,
                               C=100,t=2000,S=300)
Test4 <- svm_gradient_descent(X2,Y,
                               C=0.1,t=2000,S=300)
set.seed(80)
```

Plotting the predictions:

```
par(mfrow=c(1,2))
plot(X1,col=ifelse(Y>0,4,2),
     pch=".",cex=7,
     xlab = "x1",
     ylab = "x2")
abline(-Test1$coef[1]/Test1$coef[3],
       -Test1$coef[2]/Test1$coef[3])
abline(Test1$margin-(Test1$coef[1]/Test1$coe
       -Test1$coef[2]/Test1$coef[3],lty=2)
abline(-Test1$margin-(Test1$coef[1]/Test1$co
       -Test1$coef[2]/Test1$coef[3],lty=2)

plot(X2,col=ifelse(Y>0,4,2),
     pch=".",cex=7,xlab = "x1",
     ylab = "x2")
abline(-Test3$coef[1]/Test3$coef[3],
       -Test3$coef[2]/Test3$coef[3])
abline(Test3$margin-(Test3$coef[1]/Test3$coe
       -Test3$coef[2]/Test3$coef[3],lty=2)
```

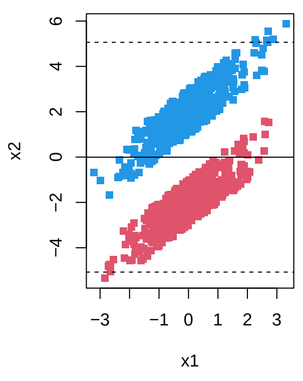
```
abline(-Test3$margin-(Test3$coef[1]/Test3$coef[3]
       -Test3$coef[2]/Test3$coef[3],lty=2)
```

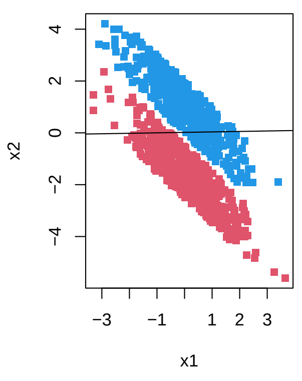


The above we can see the models run with C=100. This is proportional to setting our upper boundary on miss classifications as very high. As expected our model classifies the data almost perfectly with a very low margin.

The below we see both models with C=0.1. As explained in 2.1, C is inversely proportional to lambda (a different formulation of our model), hence low C would be equivalent to penalizing our coefficients very hard, just like in ridge regression. As we can see, we have squashed our decision boundry towards 0 all while significantly increasing the margin (in the right bottom graph margin is outside the picture).

```
par(mfrow=c(1,2))
plot(X1,col=ifelse(Y>0,4,2),pch=".",
     cex=7,xlab = "x1",ylab = "x2")
abline(-Test2$coef[1]/Test2$coef[3],
       -Test2$coef[2]/Test2$coef[3])
abline(Test2$margin-(Test2$coef[1]/Test2$coef[3])
       -Test2$coef[2]/Test2$coef[3],lty=2)
abline(-Test2$margin-(Test2$coef[1]/Test2$coef[3]
       -Test2$coef[2]/Test2$coef[3],lty=2)

plot(X2,col=ifelse(Y>0,4,2),
     pch=".",cex=7,xlab = "x1",
     ylab = "x2")
abline(-Test4$coef[1]/Test4$coef[3],
       -Test4$coef[2]/Test4$coef[3])
abline(Test4$margin-(Test4$coef[1]/Test4$coef[3])
       -Test4$coef[2]/Test4$coef[3],lty=2)
abline(-Test4$margin-(Test4$coef[1]/Test4$coef[3]
       -Test4$coef[2]/Test4$coef[3],lty=2)
```





Now, test our algorithm on the Heart Disease dataset to predict the presence or absence of heart disease based on various features. We have split the dataset into training and testing sets using a 70-30 ratio, where 70% of the data is used for training the algorithm, and 30% is kept aside for testing its performance.

```
splitIndex <- createDataPartition(
  heart_data$HeartDisease, p = 0.7,
  list = FALSE, times = 1)
train <- heart_data[splitIndex, ]
test <- heart_data[-splitIndex, ]

Xtrain <-cbind(train$Age,
               train$RestingBP,
               train$Cholesterol,
               train$MaxHR,
               train$Oldpeak)

Dummy1 <- matrix(1,nrow(test),1)
Xtest <-cbind(Dummy1,
              test$Age,
              test$RestingBP,
              test$Cholesterol,
              test$MaxHR,
               test$Oldpeak)
Ytrain <- train$HeartDisease
Ytest <- data.matrix(ifelse(
  test$HeartDisease == TRUE,1,-1))
```

Now let us deploy our model with multiple values of C to see which one performs best. We shall set S to be 100, which is around 1/3 of our train data set, and as our previous tests have shown, this should be enough to gain significant results.

```
set.seed(80)
SVM_Result1 <- svm_gradient_descent(Xtrain,
                                    Ytrain,
                                    C=0.1,
                                    t=3000,
                                    S=100)
SVM_Result2 <- svm_gradient_descent(Xtrain,
                                    Ytrain,
                                    C=1,
                                    t=3000,
                                    S=100)
SVM_Result3 <- svm_gradient_descent(Xtrain,
                                    Ytrain,
                                    C=10,
                                    t=3000,
                                    S=100)
SVM_Result4 <- svm_gradient_descent(Xtrain,
                                    Ytrain,
                                    C=100,
                                    t=3000,
                                    S=100)


head(SVM_Result1$error_rate)
```

```
## [1] 0.4416796
```

```
head(SVM_Result2$error_rate)
```

```
## [1] 0.3219285
```

```
head(SVM_Result3$error_rate)
```

```
## [1] 0.5583204
```

```
head(SVM_Result4$error_rate)
```

```
## [1] 0.2892691
```

We can see that our model with the highest C, has the lowest error rate. This is rather surprising given we wouldn't expect our data to be linearly separable. Thus we can expect there to be a significant level of model overfit.

Now let us check how our models perform on test data:

```
  SVM_predicted1 <- ifelse(
    Xtest%*%SVM_Result1$coef<0,-1,1)
  SVM_test_error_rate1 <- sum(ifelse(
    SVM_predicted1 ==Ytest,0,1))/nrow(Ytest)

  SVM_predicted2 <- ifelse(
    Xtest%*%SVM_Result2$coef<0,-1,1)
  SVM_test_error_rate2 <- sum(
    ifelse(SVM_predicted2 ==Ytest,0,1))/nrow(Ytes

  SVM_predicted3 <- ifelse(
    Xtest%*%SVM_Result3$coef<0,-1,1)
  SVM_test_error_rate3 <- sum(
    ifelse(SVM_predicted3 ==Ytest,0,1))/nrow(Ytes
  SVM_predicted4 <- ifelse(
    Xtest%*%SVM_Result4$coef<0,-1,1)
  SVM_test_error_rate4 <- sum(
    ifelse(SVM_predicted4 ==Ytest,0,1))/nrow




  head(SVM_test_error_rate1)
```

```
## [1] 0.4581818
```

```
  head(SVM_test_error_rate2)
```

```
## [1] 0.4
```

```
  head(SVM_test_error_rate3)
```

```
## [1] 0.5418182
```

```
  head(SVM_test_error_rate4)
```

```
## [1] 0.3490909
```

```

source(here("R","confusion_matrix_.R"))
confusion_matrix_results = ConfusionMatrix(
  as.character(Ytest),as.character(SVM_predi
```

```

print(xtable(
    head(heart_data),
    caption='Example of Heart Disease Dataset',
    label='tbl:xtable.floating'),
  floating.environment='table*')
```

```

 print(xtable(
   confusion_matrix_results,
   caption='confusion matrix results',
   label='tbl:results.xtable',
   align=c(rep('r', 6), 'l')))
```

1) Classification Results::

F. Non linear SVM SHOW CASE:

SVM model with a non-linear kernel (such as a radial basis function, or RBF kernel) on the iris dataset in R, and then plot the accuracy against different values of the kernel parameter (k):

```
# Load required libraries
library(e1071)
library(ggplot2)

# Load iris dataset
data(iris)

# Split the dataset into features (X) and labels
X <- iris[, 1:4]
Y <- iris[, 5]
# Set the seed for reproducibility
set.seed(123)

# Split the data into training
# and testing sets (70% training, 30% testing)
sample_index <- sample(1:nrow(iris),
                       0.7 * nrow(iris))
X_train <- iris[sample_index, 1:4]
Y_train <- iris[sample_index, 5]
X_test <- iris[-sample_index, 1:4]
Y_test <- iris[-sample_index, 5]
```

Here, I have used r CRAN pacakge e1071 to train the SVM.

```
svm_model <- svm(Species ~ .,
                 data = iris[sample_index, ],
                 kernel = "radial")

# Make predictions on
# the test set
predictions <- predict(svm_model,
                       newdata = X_test)
```

III. Cross validataion

A. k-fold Cross-validatation.

K-fold cross-validation is a resampling technique used to assess the performance of a machine learning model. It divides the dataset into ‘k’ subsets or folds. The model is trained on ‘k-1’ folds and tested on the remaining fold. This process is repeated ‘k’ times, and each fold is used exactly once as the validation data.

1) Steps of K-Fold Cross-Validation::

1. **Divide the Data:** Split the dataset into ‘k’ subsets of approximately equal size, often stratified by the target variable for balanced representation in each fold.
2. **Training and Testing:** Train the model ‘k’ times, using ‘k-1’ folds as the training data and the remaining fold as the test set. Evaluate the model’s performance on the test set.
3. **Evaluation:** Calculate performance metrics such as accuracy, precision, recall, or F1-score for each iteration. Average these metrics to obtain an overall performance estimate of the model.

TABLE I: Example of Heart Disease Dataset

Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak	ST_Slope	HeartDisease
40	M	ATA	140	289	0	Normal	172	N	0.00	Up	0
49	F	NAP	160	180	0	Normal	156	N	1.00	Flat	1
37	M	ATA	130	283	0	ST	98	N	0.00	Up	0
48	F	ASY	138	214	0	Normal	108	Y	1.50	Flat	1
54	M	NAP	150	195	0	Normal	122	N	0.00	Up	0
39	M	NAP	120	339	0	Normal	170	N	0.00	Up	0

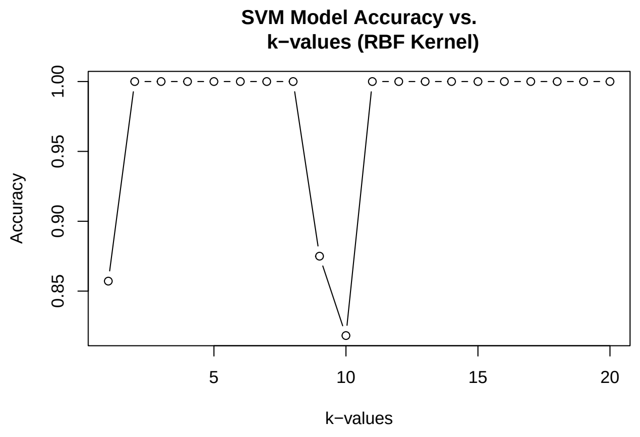
TABLE II: confusion matrix results

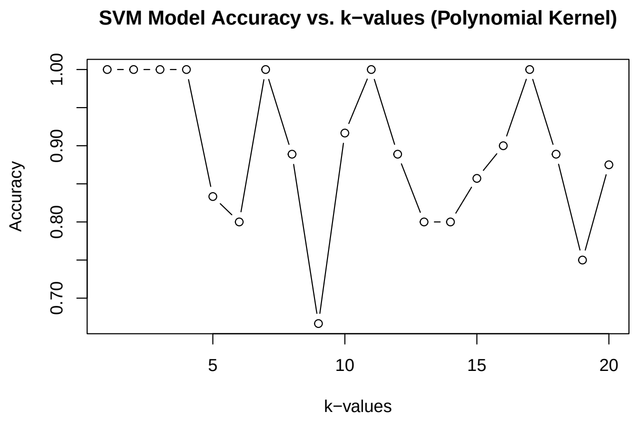
Accuracy	Precision	Sensitivity	F1_Score	Specificity	AUC
0.60	0.64	0.56	0.59	0.65	0.60

```r
data <- iris
k <- 20
data$id <- sample(1:k, nrow(data), replace
accuracies <- numeric(k)

# Function for k-fold cross-validation
for (i in 1:k) {
  trainset <- subset(data,
                     id %in% unique(data$id)
  testset <- subset(data,
                    id %in% c(i))
  svm_model <- svm(Species ~ .,
                   data = trainset, kernel
  predictions <- predict(svm_model,
                         testset)
  accuracy <- sum(
    predictions == testset$Species) / nrow(t
  accuracies[i] <- accuracy
}

# Plot the graph of k-values with accuracies
plot(1:k, accuracies, type = "b",
     xlab = "k-values",
     ylab = "Accuracy",
     main = "SVM Model Accuracy vs.
     k-values (RBF Kernel)")
```

the kernel has been changed to “poly” in the below code. indicating a polynomial kernel. The degree = 2 parameter specifies the degree of the polynomial kernel. you can increase degrees if you want to.

```r
data <- iris
k <- 20
data$id <- sample(1:k, nrow(data), replace = TRUE
accuracies <- numeric(k)

# Function for k-fold cross-validation
for (i in 1:k) {
  trainset <- subset(data,
                     id %in% unique(data$id)[-i])
  testset <- subset(data,
                    id %in% c(i))
  svm_model <- svm(Species ~ .,
                   data = trainset,
                   kernel = "poly", degree = 2)
  predictions <- predict(svm_model,
                         testset)
  accuracy <- sum(
    predictions == testset$Species) / nrow(testse
  accuracies[i] <- accuracy
}

# Plot the graph of k-values with accuracies
plot(1:k, accuracies, type = "b",
     xlab = "k-values",
     ylab = "Accuracy",
     main = "SVM Model Accuracy vs. k-values (Pol
```





References

[1] "Implementing support vector machine from scratch — by marvin lanhenke — towards data science," https://towardsdatascience.com/implementing-svm-from-scratch-784e4ad0bc6a, (Accessed on 10/12/2023).

[2] "An introduction to statistical learning," https://www.statlearning.com/, (Accessed on 10/12/2023).

[3] "Elements of statistical learning: data mining, inference, and prediction. 2nd edition." https://hastie.su.domains/ElemStatLearn/, (Accessed on 10/12/2023).

[4] "A definitive explanation to the hinge loss for support vector machines. — by vagif aliyev — towards data science," https://towardsdatascience.com/a-definitive-explanation-to-hinge-loss-for-support-vector-machines-ab6d8d3178f1, (Accessed on 10/12/2023).