This dataset contains 654 data points, with the first 10 columns being the predictor variables and the last column being a binary response variable indicating if the application was positive (1) or negative (0).

```
# Load the kernlab package
library('kernlab')

# Read the credit card data from a text file into a data frame
data <- read.table('data/credit_card_data.txt', header = FALSE)
```

The support vector machine function '**ksvm**' is used to find a classifier for the data. The argument '**kernel = "vanilladot"**' is used to classify the data with a linear kernel function.

The parameter '**C**' is the regularization term that determines the trade-off between having a low error on the training data and maintaining a large margin that separates the classes. Larger values of '**C**' gives a smaller-margin hyperplane that emphasize correct classification of training data. While this can reduce training error, it may lead to overfitting, meaning the model becomes too specific to the training data and performs poorly on unseen data.

Conversely, smaller values of '**C**' will result in a larger-margin hyper-plane. A larger margin provides a more conservative decision boundary, but at the cost of misclassifying more points in the training set, as more points fall within the margin. The relationship between training error and margin size is described by the objective function, where '**C**' is inversely proportional to λ:

$$\underset{a_0,\cdots,a_n}{\text{Minimize}} \sum_{i=1}^{m} \max\left\{0,1-\left(\sum_{j=1}^{n} a_j\, x_{ij} + a_0\right) y_i\right\} + \lambda \sum_{j=1}^{n} (a_j)^2$$

Using a for-loop, different orders of magnitude of '**C**' is tested to tune the trade off between correctness and margin. Finally, each value of '**C**' and their accuracy is stored into a data frame named '**df1**'.

```
# Create an empty data frame to store the values of C and corresponding
accuracy
df1 <- data.frame(C = numeric(), Accuracy = numeric())

# Loop over a range of C values from 10^-10 to 10^10
for (i in 10^(-10:10)){

  #training a model with the ksvm function
  invisible(capture.output({
  model <- ksvm(as.matrix(data[,1:10]), # the feature matrix
                as.factor(data[,11]), # the response variable
                type="C-svc", # specifies to use C-classification
                kernel = "vanilladot", #specifies to use linear kernel
                C=i, # the regularization parameter
```

```
                scaled=TRUE)  #scales the data
  }))


  #Predict the response variable for the input data using the trained model
  pred <- predict(model,data[,1:10])

  # Calculate the accuracy of the predictions
  pred_accuracy <- sum(pred == data[,11]) / nrow(data)

  # Append the current value of C and the corresponding accuracy to the data
frame
  df1 <- rbind(df1, data.frame(C=i, Accuracy = pred_accuracy))

  }
# Print the results data frame
print(df1, row.names = FALSE)
```

```
##       C  Accuracy
##  1e-10 0.5474006
##  1e-09 0.5474006
##  1e-08 0.5474006
##  1e-07 0.5474006
##  1e-06 0.5474006
##  1e-05 0.5474006
##  1e-04 0.5474006
##  1e-03 0.8379205
##  1e-02 0.8639144
##  1e-01 0.8639144
##  1e+00 0.8639144
##  1e+01 0.8639144
##  1e+02 0.8639144
##  1e+03 0.8623853
##  1e+04 0.8623853
##  1e+05 0.8639144
##  1e+06 0.6253823
##  1e+07 0.5458716
##  1e+08 0.6636086
##  1e+09 0.8027523
##  1e+10 0.4923547
```

The results indicate that within the '**C**' parameter range of 1e-10 to 1e-04, the model's accuracy is 0.5474006. This is because the margin is too large that all training points fall within it, leading to a prediction of zero for the response variable across all data points. This accuracy matches exactly to the proportion of zero in the response variable of the original dataset, which is 358 out of 654 data points, indicating a baseline accuracy is 54.7%.

When the '**C**' parameter is set between 0.01 and 100, the prediction accuracy peaks at 0.8639144, which corresponds to 565 correct predictions. In the context of credit card approval, approving a risky client (false positives) is more costly than rejecting a good client (false negatives). A larger margin with a more conservative decision boundary reduces the risk of approving borderline cases that might default. Therefore, '**C = 0.01**' is selected for the classifier, as it has the highest prediction accuracy while having a low '**C**' value, thereby ensuring the largest margin.

The classifier equation is in the form of:

$$a_1 x_1 + \cdots + a_m x_m + a_0 = 0$$

Multiplying the support vectors ('**xmatrix**') by their corresponding coefficients ('**coef**'), then summing these products across all support vectors using '**colSums**', yields a_1 to a_m of the decision boundary. And the intercept a_0 is calculated as the negative of the stored value '**b**'.

```
#training a model with the ksvm function
  invisible(capture.output({
  model001 <- ksvm(as.matrix(data[,1:10]), # the feature matrix
                as.factor(data[,11]), # the response variable
                type="C-svc", # specifies to use C-classification
                kernel = "vanilladot", #specifies to use linear kernel
                C=0.01, # the regularization parameter set to 0.01
                scaled=TRUE)}))  #scales the data

  # Calculate the coefficients of the model
  a <- colSums(model001@xmatrix[[1]] * model001@coef[[1]])

  # Calculate the intercept of the model
  a0 <- -model001@b


#Print the coefficients 'a' with a label
cat("Coefficients (a):\n", a, "\n\n")
```

```
## Coefficients (a):
##  -0.0001500738 -0.001481829 0.001408313 0.007286389 0.991647 -0.004466124
0.00714829 -0.0005468386 -0.001693058 0.1054824
```

```
# Print the intercept 'a0' with a label
cat("Intercept (a0):\n", a0, "\n")
```

```
## Intercept (a0):
##  0.08198854
```

Therefore, the equation of the classifier with C = 0.01, for scaled data x is:

$$-0.0001500738x_1 - 0.001481829x_2 + 0.001408313x_3 + 0.0072863886x_4 + 0.991647x_5 - 0.004466124x_6 + 0.00714829x_7 - 0.0005468386x_8 - 0.001693058x_9 + 0.1054824x_{10} + 0.08198854 = 0$$

Testing Other Kernel Function

Other kernel function can be used to train and predict the dataset by changing the '**kernel**' argument. For instance, '**kernel = "rbfdot"**' uses the radial basis kernel to train the data.

```
#creating an empty data frame
df_rbf <- data.frame(C = numeric(), Accuracy = numeric())
# Loop over a range of C values from 10^-5 to 10^10
for (i in 10^(-5:10)){
  model <- ksvm(as.matrix(data[,1:10]), # the feature matrix
                as.factor(data[,11]), # the response variable
                type="C-svc", # specifies to use C-classification
                kernel = "rbfdot", #specifies to use Radial Basis kernel
                C=i, # the regularization parameter set to 0.01
                scaled=TRUE)  #scales the data

  #Predict the response variable for the input data using the trained model
  pred <- predict(model,data[,1:10])

  # Calculate the accuracy of the predictions
  pred_accuracy <- sum(pred == data[,11]) / nrow(data)

  # Append the current value of C and the corresponding accuracy to the data
frame
  df_rbf <- rbind(df_rbf, data.frame(C=i, Accuracy = pred_accuracy))

}
# Print the results data frame
print(df_rbf, row.names = FALSE)
```

```
##      C  Accuracy
##  1e-05 0.5474006
##  1e-04 0.5474006
##  1e-03 0.5474006
##  1e-02 0.5703364
##  1e-01 0.8593272
##  1e+00 0.8715596
##  1e+01 0.9204893
##  1e+02 0.9541284
##  1e+03 0.9847095
##  1e+04 0.9954128
##  1e+05 0.9969419
##  1e+06 0.9984709
##  1e+07 1.0000000
##  1e+08 1.0000000
```

```
##   1e+09 1.0000000
##   1e+10 1.0000000
```

The radial basis kernel achieve prediction accuracy of 100% when C is over 1e+07. However, this result does not reflect the model's performance on new, unseen dataset. This is because it is an overfitted model that would likely fail to generalize to new data.

Comparison with KNN model

For k-nearest neighbors classification, I have implemented two different approaches. The first one involves splitting the dataset into training set and test set, while the second approach uses the leave-one-out cross validation method.

The '**kknn**' package is loaded into R. Then the same dataset is read and stored as a data frame named '**credit_card_data**'. The '**set.seed**' function is used to ensure reproducibility when randomly sampling the data.

```
# Load the 'kknn' library for k-nearest neighbors classification
library('kknn')

# Read the credit card data from a text file into a data frame
credit_card_data <- read.table('data/credit_card_data.txt', header = FALSE)

# Set the seed for random number generation to ensure reproducibility
set.seed(1)
```

Method 1:

The dataset is splitted into 75% of training data (490 data points) and 25% of test data (164 data points) using the 'sample' function, where each data point has an equal probability of being sampled as the test set.

```
# Get the number of rows in the credit_card_data dataset (654)
m <- dim(credit_card_data)[1]

# Randomly sample 1/4 of the rows without replacement
# i: indices of the sampled rows
# m: total number of rows
i = sample(1:m, # sampling from 1 to m (654)
           size = round(m/4), # number of rows to sample (1/4 of total rows)
           replace = FALSE, # sampling without replacement
           prob = rep(1/m, m))# prob: equal probability for each row to be
selected

# Create the training dataset by excluding the sampled rows
train_data <- credit_card_data[-i, ]

# Create the test dataset using the sampled rows
test_data <- credit_card_data[i, ]
```

To perform k-nearest neighbor classification, the '**kknn**' function is used to train the data. The argument **'formula = V11~**' specifies that the response variable is in the 11th column of the dataset.The '**k**' argument determines how many neighboring data points are used to classify the data. To determine the optimal value, a for-loop is set up to test '**k**' values ranging from **'k=1**' through **'k=30**'.

The '**fitted**' function outputs the predicted probability that the response variable is 1 for each data point in the test dataset. To convert these probabilities into binary predictions, the '**ifelse**()' function is used, which assigns a value of 1 to predictions with a probability of 0.5 or higher, and a value of 0 to those with a probability below 0.5. Finally the prediction accuracy is calculated by dividing the correct prediction over the total number of data point in the testing set. The '**k**' values and their corresponding accuracy are stored into a data frame named '**df**'.

```
# Create an empty data frame to store K values and corresponding accuracies
df <- data.frame(K =numeric(),Accuracy =  numeric())

#Loop over a range of K values from 1 to 30
for (k in 1:30){

  # Apply the kknn with the current K value
  credit_card_kknn <- kknn(formula = V11~., #V11 is the response variable
                           train_data, # the training dataset
                           test_data, # the testing dataset
                           k=k, # the number of neighbors to consider
                           scale=TRUE) # scale the data

  # Get the predicted values
  prediction = fitted(credit_card_kknn)

  # Convert predictions to binary (0 or 1) based on a threshold of 0.5
  Binary.Prediction <- ifelse(prediction >= 0.5, 1, 0)

  # Create a confusion matrix to compare actual vs predicted values
  # see next part for example of confusion matrix
  confusion_matrix <- table(test_data$V11, Binary.Prediction)

  # Calculate accuracy as the sum of the diagonal elements (correct
predictions)
  # divided by the total number of predictions
  accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)

  # Append the current K value and its corresponding accuracy to the data
frame
  df <- rbind(df, data.frame(K = k, Accuracy = accuracy))
  }
# print the data frame
df <- df[order(df$Accuracy, decreasing = TRUE),]
```

```r
print(df, row.names = FALSE)
```

```
##   K  Accuracy
##  12 0.8963415
##  10 0.8902439
##  11 0.8902439
##  13 0.8902439
##  14 0.8902439
##  15 0.8902439
##  16 0.8902439
##  17 0.8841463
##   9 0.8780488
##  18 0.8780488
##  19 0.8780488
##  20 0.8780488
##   5 0.8719512
##   7 0.8719512
##   8 0.8719512
##  21 0.8719512
##  22 0.8719512
##   6 0.8597561
##  23 0.8597561
##  24 0.8597561
##  28 0.8597561
##  29 0.8597561
##  30 0.8597561
##  25 0.8536585
##  26 0.8536585
##  27 0.8536585
##   1 0.8292683
##   2 0.8292683
##   3 0.8292683
##   4 0.8292683
```

When k=12, the prediction accuracy peaks at 0.8963415, which correspond to 147 correct predictions out of 164 data points. Finally, using the '**table**' function, a confusion matrix can be constructed to visualize the binary predictions for when k = 12.

```r
credit_card_kknn <- kknn(formula = V11~., #V11 is the response variable
                         train_data, # the training dataset
                         test_data, # the testing dataset
                         k=12, # 12 neighbors to consider
                         scale=TRUE) # scale the data
# Get the predicted values
prediction = fitted(credit_card_kknn)

# Convert predictions to binary (0 or 1) based on a threshold of 0.5
Binary.Prediction <- ifelse(prediction >= 0.5, 1, 0)
```

```r
# Create a confusion matrix to compare actual vs predicted values
confusion_matrix <- table(test_data$V11, Binary.Prediction)

# print the confusion matrix
print(confusion_matrix)
```

```
##    Binary.Prediction
##      0  1
##   0 84 10
##   1  7 63
```

Method 2:

In this leave-one-out cross validation approach, each data point is used once as a test set, while all other data points are the training set. The outer loop is set up to find the best '**k**' value, while the inner loop allows every data point to be tested. The arguments used in the '**kknn**' function are identical to those in method 1, as are the procedures for obtaining predictions and calculating accuracy. The 'k' values and their corresponding accuracy is stored in a data frame and printed.

```r
# create an empty data frame to store K values and corresponding accuracy
df <- data.frame(K = numeric(), Accuracy = numeric())

# create a vector to store predictions for each observation in the dataset
prediction_record <- rep(0, m)

# Loop over a range of K values from 1 to 30
for (k in 1:30) {

  # Loop over each data point in the dataset
  for (i in 1:m) {

    # use all but the i th observation as training data
    train_data <- credit_card_data[-i, ]

    # Use the i th observation as test data
    test_data <- credit_card_data[i, ]

    # Apply knn with the current value of K
    credit_card_kknn <- kknn(formula = V11 ~ ., train_data, test_data, k = k,
scale = TRUE)

    # Get the prediction for the test data
    prediction <- fitted(credit_card_kknn)

    # Convert the prediction to binary (0 or 1) using a threshold of 0.5
    Binary.Prediction <- ifelse(prediction >= 0.5, 1, 0)

    # Store the binary prediction in the prediction record
```

```r
    prediction_record[i] <- Binary.Prediction

  }

  # Calculate the accuracy as the proportion of correct predictions
  accuracy <- sum((prediction_record) == credit_card_data[, 11]) / m

  # Append the current K value and its corresponding accuracy to the data
frame
  df <- rbind(df, data.frame(K = k, Accuracy = accuracy))

}

# Sort the data frame by Accuracy in ascending order
df <- df[order(df$Accuracy, decreasing = TRUE), ]

# Print the sorted data frame without row names
print(df, row.names = FALSE)
```

```
##   K  Accuracy
##  12 0.8532110
##  15 0.8532110
##   5 0.8516820
##  11 0.8516820
##  13 0.8516820
##  14 0.8516820
##  16 0.8516820
##  17 0.8516820
##  18 0.8516820
##  10 0.8501529
##  19 0.8501529
##  20 0.8501529
##   8 0.8486239
##  21 0.8486239
##   7 0.8470948
##   9 0.8470948
##  22 0.8470948
##   6 0.8455657
##  24 0.8455657
##  25 0.8455657
##  23 0.8440367
##  26 0.8440367
##  27 0.8409786
##  30 0.8409786
##  29 0.8394495
##  28 0.8379205
##   1 0.8149847
##   2 0.8149847
```

```
##   3 0.8149847
##   4 0.8149847
```

In this method, the best '**k**' values are 12 and 15 with accuracy of 0.8532110.