Homework_1

Claire Kraft

2024-08-21

Question 2.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a classification model would be appropriate. List some (up to 5) predictors that you might use.

- I got into rock climbing less than a year ago. I think a classification model would be very useful for detecting climbing routes. In climbing there are route setter who configure the climbing “problems” by strategically screwing various climbing holds on the wall. It’d be neat if a classification model can scan the walls and determine the climbing style by looking at both the climbing holds and configuration of the holds. Some climbs are set up to force more dynamic moves and other climbs (called “slabs”) which force technical, precise, and static moves. The prediction model can label the climbing routes or problems as either a dynamic climb or slab climb.
- When not climbing I enjoy speed cubing (solving Rubik’s cube quickly). I just started competing this past year. A classification model could be beneficial to my training session. The model can detect my turns per second and case recognition & prediction. For the classic 3x3 Rubik’s cube there are a few methods to getting the puzzle to a solved state. In this case we’ll just consider the CFOP method which is: cross + first two layers + orientation of the last layer + permutation of the last layer. Simply put this method is much like baking a cake (layer by layer). As I’m solving i do not have to watch my hands as i manipulate the pieces instead i commit almost everything to muscle memory and visually scan the cube for patterns. Upon recognizing the patterns, I have to execute the most optimal algorithm to reach another another pattern repeatedly until the whole cube is solved. A classification model can learn the patterns, predict the best algorithms, and clock my turns per second. Essentially a classification model could be doing what I’m doing in parallel and just compare my performance with itself, much like a chess engine that is computing alongside the chess player. A really top notch cuber will turn the cubes so fluidly and controllably to be able to scan patterns, predict, and execute without seeming to stop.

Question 2.2

The files credit_card_data.txt (without headers) and credit_card_data-headers.txt (with headers) contain a dataset with 654 data points, 6 continuous and 4 binary predictor variables. It has anonymized credit card applications with a binary response variable (last column) indicating if the application was positive or negative. The dataset is the “Credit Approval Data Set” from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Credit+Approval (https://archive.ics.uci.edu/ml/datasets/Credit+Approval)) without the categorical variables and without data points that have missing values.

Question 2.2.1

Using the support vector machine function ksvm contained in the R package kernlab, find a good classifier for this data. Show the equation of your classifier, and how well it classifies the data points in the full data set. (Don’t worry about test/validation data yet; we’ll cover that topic soon.)

Using the linear kernel (vanilladot) to test various c values to find the “best” model. C controls the margin of error in classification. In the module 2 lectures Dr. Sokol explains how he increases the margins or threshold for classifying mushrooms. Despite wild mushrooms looking and smelling like the ones in the grocery store, all mushrooms are assumed to be inedible. The high threshold ensures no one is hurt from consuming the wrong mushroom. The bigger the C the less risk of misclassifications. The smaller the C the more chance of misclassification.

After the brute force method the most accurate rate is 86.39%. It’s interesting that the accuracy is highest when the margins are lower and the accuracy decreases as the margin raises. This observation seems to go against the theory. Based on working with data scientists industry, I know that accuracy isn’t the only metric for determining good or bad model. There are other metrics such as confusion matrix, F1 score, precision, etc to consider as well.

```
#------------------------ getting a sense of the data
# Read in credit_card_data data (source: https://teacherscollege.screenstepslive.com/a/1126998-i
mport-data-into-r-txt-files-in-r-studio)
credit <- read.table("C://Users//Clair//OneDrive//Documents//Fall 2024//IYSE 6501//hw1//data 2.
2//credit_card_data.txt", sep="\t", header=FALSE)
# View a summary of the dataset including basic statistics for each column
summary(credit)
```

```
##        V1               V2              V3               V4
##  Min.   :0.0000   Min.   :13.75   Min.   : 0.000   Min.   : 0.000
##  1st Qu.:0.0000   1st Qu.:22.58   1st Qu.: 1.040   1st Qu.: 0.165
##  Median :1.0000   Median :28.46   Median : 2.855   Median : 1.000
##  Mean   :0.6896   Mean   :31.58   Mean   : 4.831   Mean   : 2.242
##  3rd Qu.:1.0000   3rd Qu.:38.25   3rd Qu.: 7.438   3rd Qu.: 2.615
##  Max.   :1.0000   Max.   :80.25   Max.   :28.000   Max.   :28.500
##        V5               V6               V7               V8
##  Min.   :0.0000   Min.   :0.0000   Min.   : 0.000   Min.   :0.0000
##  1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.: 0.000   1st Qu.:0.0000
##  Median :1.0000   Median :1.0000   Median : 0.000   Median :1.0000
##  Mean   :0.5352   Mean   :0.5612   Mean   : 2.498   Mean   :0.5382
##  3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.: 3.000   3rd Qu.:1.0000
##  Max.   :1.0000   Max.   :1.0000   Max.   :67.000   Max.   :1.0000
##        V9               V10              V11
##  Min.   :   0.00   Min.   :     0   Min.   :0.0000
##  1st Qu.:  70.75   1st Qu.:     0   1st Qu.:0.0000
##  Median : 160.00   Median :     5   Median :0.0000
##  Mean   : 180.08   Mean   :  1013   Mean   :0.4526
##  3rd Qu.: 271.00   3rd Qu.:   399   3rd Qu.:1.0000
##  Max.   :2000.00   Max.   :100000   Max.   :1.0000
```

```
# Get dimension of df, row x column
print("Size of df: ")
```

```
## [1] "Size of df: "
```

```
dim(credit)
```

```
## [1] 654  11
```

```
# Look for null values
print("Count of total missing values: ")
```

```
## [1] "Count of total missing values: "
```

```
sum(is.na(credit))
```

```
## [1] 0
```

```
#------------------------- manipulating the data
# convert .txt > df > matrix (source: https://stackoverflow.com/questions/46518838/how-to-convert-table-to-matrix-in-r)
credit_matrix <- data.matrix(credit)
head(credit_matrix)
```

```
##      V1    V2    V3   V4 V5 V6 V7 V8  V9 V10 V11
## [1,]  1 30.83 0.000 1.25  1  0  1  1 202   0   1
## [2,]  0 58.67 4.460 3.04  1  0  6  1  43 560   1
## [3,]  0 24.50 0.500 1.50  1  1  0  1 280 824   1
## [4,]  1 27.83 1.540 3.75  1  0  5  0 100   3   1
## [5,]  1 20.17 5.625 1.71  1  1  0  1 120   0   1
## [6,]  1 32.08 4.000 2.50  1  1  0  0 360   0   1
```

```
#------------------------- helper
#install.packages("kernlab")
library(kernlab)

#------------------------- training the data
# Call ksvm(), Vanilladot is a simple linear kernel
# Train the model using the first 10 columns as features, 11th column is the target
# Parameter is c=x
model <- ksvm(credit_matrix[,1:10],as.factor(credit_matrix[,11]),type="C-svc",kernel="vanilladot",C=1,scaled=TRUE)
```

```
##  Setting default kernel parameters
```

```
# calculate a1…am
a <- colSums(model@xmatrix[[1]] * model@coef[[1]])
print("Equation: ")
```

```
## [1] "Equation: "
```

```
print(a)
```

```
##            V1            V2            V3            V4            V5 
## -0.0011026642 -0.0008980539 -0.0016074557  0.0029041700  1.0047363456 
##            V6            V7            V8            V9           V10 
## -0.0029852110 -0.0002035179 -0.0005504803 -0.0012519187  0.1064404601
```

```
# calculate a0
a0 <- -model@b
print("a0 intercept: ")
```

```
## [1] "a0 intercept: "
```

```
print(a0)
```

```
## [1] 0.08148382
```

```
#------------------------- predicting the data
# See what the model predicts
pred <- predict(model,credit_matrix[,1:10])
print("Prediction: ")
```

```
## [1] "Prediction: "
```

```
print(pred)
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
##  [75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [112] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [149] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [186] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
## [223] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
## [260] 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
## [297] 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
## [334] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [371] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [408] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [445] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [482] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [519] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
## [556] 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
## [593] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
## [630] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## Levels: 0 1
```

```
# See what fraction of the model’s predictions match the actual classification
accuracy <- sum(pred == credit_matrix[,11]) / nrow(credit_matrix)
accuracy
```

```
## [1] 0.8639144
```

2.2.2

You are welcome, but not required, to try other (nonlinear) kernels as well; we’re not covering them in this course, but they can sometimes be useful and might provide better predictions than vanilladot.

Looking at the R documentation page (https://search.r-project.org/CRAN/refmans/kernlab/html/ksvm.html), polynomial `polydot` and Radial basis function `rbfdot` are nonlinear kernels. The polynomial model seems to be the most consistent in high accuracy scores with sub 1% difference. Radial basis function model seems to behave as expected, the bigger the C the less risk of misclassifications. The smaller the C the more chance of misclassification. The accuracy metric seems more appropriate as evaluation for the nonlinear classification models compared to the linear classification model.

```r
#------------------------ getting a sense of the data
# Read in credit_card_data data (source: https://teacherscollege.screenstepslive.com/a/1126998-import-data-into-r-txt-files-in-r-studio)
credit <- read.table("C://Users//Clair//OneDrive//Documents//Fall 2024//IYSE 6501//hw1//data 2.2//credit_card_data.txt", sep="\t", header=FALSE)

# View a summary of the dataset including basic statistics for each column
# summary(credit) didn't change my dataset so don't need to reprint summary

# Look for null values
#print("Count of total missing values: ")
#sum(is.na(credit)) didn't change my dataset so the missing values count have not changed

#------------------------ manipulating the data
# Convert .txt > df > matrix (source: https://stackoverflow.com/questions/46518838/how-to-convert-table-to-matrix-in-r)
credit_matrix <- data.matrix(credit)
#head(credit_matrix) didn't change my dataset so don't need to reprint summary

#------------------------ helper
#install.packages("kernlab")
library(kernlab)

#------------------------ loop through 10 models with varying margins
# Define a vector to store accuracy for each model
accuracies <- c()

# Loop through values of C from 100 to 1000 in steps of 100
for (C_value in seq(100, 1000, by=100)) {

  # Train an SVM model using the polydot kernel with varying C
  model <- ksvm(credit_matrix[,1:10], as.factor(credit_matrix[,11]),
                type="C-svc", kernel="polydot",
                kpar=list(degree=3), C=C_value, scaled=TRUE)

  # Predict the data using the trained model
  pred <- predict(model, credit_matrix[,1:10])

  # Calculate accuracy of the model by comparing predictions to actual class labels
  accuracy <- sum(pred == credit_matrix[,11]) / nrow(credit_matrix)

  # Print the C value and corresponding accuracy
  print(paste("C =", C_value, "-> Accuracy:", accuracy))

  # Store the accuracy for each model
  accuracies <- c(accuracies, accuracy)
}
```

```
## [1] "C = 100 -> Accuracy: 0.990825688073395"
## [1] "C = 200 -> Accuracy: 0.99388379204893"
## [1] "C = 300 -> Accuracy: 0.99388379204893"
## [1] "C = 400 -> Accuracy: 0.995412844036697"
## [1] "C = 500 -> Accuracy: 0.995412844036697"
## [1] "C = 600 -> Accuracy: 0.995412844036697"
## [1] "C = 700 -> Accuracy: 0.99388379204893"
## [1] "C = 800 -> Accuracy: 0.995412844036697"
## [1] "C = 900 -> Accuracy: 0.995412844036697"
## [1] "C = 1000 -> Accuracy: 0.995412844036697"
```

```
#------------------------- getting a sense of the data
# Read in credit_card_data data (source: https://teacherscollege.screenstepslive.com/a/1126998-import-data-into-r-txt-files-in-r-studio)
credit <- read.table("C://Users//Clair//OneDrive//Documents//Fall 2024//IYSE 6501//hw1//data 2.2//credit_card_data.txt", sep="\t", header=FALSE)

# View a summary of the dataset including basic statistics for each column
# summary(credit) didn't change my dataset so don't need to reprint summary

# Look for null values
#print("Count of total missing values: ")
#sum(is.na(credit)) didn't change my dataset so the missing values count have not changed

#------------------------- manipulating the data
# Convert .txt > df > matrix (source: https://stackoverflow.com/questions/46518838/how-to-convert-table-to-matrix-in-r)
credit_matrix <- data.matrix(credit)
#head(credit_matrix) didn't change my dataset so don't need to reprint summary

#------------------------- helper
#install.packages("kernlab")
library(kernlab)

#------------------------- loop through 10 models with varying margins
# Define a vector to store accuracy for each model
accuracies <- c()

# Loop through values of C from 100 to 1000 by 100
for (C_value in seq(100, 1000, by=100)) {

  # Train an SVM model using the RBF kernel with varying C
  model <- ksvm(credit_matrix[,1:10], as.factor(credit_matrix[,11]),
                type="C-svc", kernel="rbfdot",
                C=C_value, scaled=TRUE)

  # Predict the data using the trained model
  pred <- predict(model, credit_matrix[,1:10])

  # Calculate accuracy of the model by comparing predictions to actual class labels
  accuracy <- sum(pred == credit_matrix[,11]) / nrow(credit_matrix)

  # Print the C value and corresponding accuracy
  print(paste("C =", C_value, "-> Accuracy:", accuracy))

  # Store the accuracy for each model
  accuracies <- c(accuracies, accuracy)
}
```

```
## [1] "C = 100 -> Accuracy: 0.952599388379205"
## [1] "C = 200 -> Accuracy: 0.960244648318043"
## [1] "C = 300 -> Accuracy: 0.969418960244648"
## [1] "C = 400 -> Accuracy: 0.969418960244648"
## [1] "C = 500 -> Accuracy: 0.969418960244648"
## [1] "C = 600 -> Accuracy: 0.977064220183486"
## [1] "C = 700 -> Accuracy: 0.977064220183486"
## [1] "C = 800 -> Accuracy: 0.980122324159021"
## [1] "C = 900 -> Accuracy: 0.984709480122324"
## [1] "C = 1000 -> Accuracy: 0.980122324159021"
```

Question 2.2.3

Using the k-nearest-neighbors classification function kknn contained in the R kknn package, suggest a good value of k, and show how well it classifies that data points in the full data set. Don't forget to scale the data (scale=TRUE in kknn).

In the scaled KNN model, the highest accuracy is 81.50% with k=1, which indicates a high bias as it prefers itself. In contrast, the unscaled model achieves 85.32% accuracy with k=12. This suggests that the unscaled model, with a k value greater than 1, provides a more balanced representation of the data and better captures the realistic/organic variability in the data.

scaled

```
#------------------------- getting a sense of the data
# Reading in the credit card data
credit <- read.table("C://Users//Clair//OneDrive//Documents//Fall 2024//IYSE 6501//hw1//data 2.
2//credit_card_data.txt", sep="\t", header=FALSE)

#------------------------- manipulating the data
# Convert to data frame and scale the features
credit_df <- as.data.frame(credit)
scaled_credit_df <- as.data.frame(scale(credit_df[, 1:10])) # scale df
scaled_credit_df$V11 <- as.factor(credit_df$V11)

#------------------------- helper libraries
#install.packages("kknn") (source:https://www.rdocumentation.org/packages/kknn/versions/1.3.1)
library(kknn)
library(ggplot2)
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:kernlab':
##
##     alpha
```

```r
#------------------------ leave-one-out cross-validation using kknn with internal scaling
accuracies <- c()  # Vector to store accuracy for different K values

# Loop through values of K from 1 to 50
for (K_value in 1:50) {
  correct_predictions <- 0  # Count correct predictions

  # Loop through each row for leave-one-out cross-validation
  for (i in 1:nrow(credit_df)) {

    # Exclude the i-th data point for training
    train_data <- credit_df[-i, ]  # All but the i-th row for training
    test_data <- credit_df[i, , drop = FALSE]  # Only the i-th row for testing

    # Fit the kknn model with internal scaling
    model <- kknn(V11 ~ ., train = train_data, test = test_data, k = K_value, scale = TRUE)

    # Get the predicted label
    pred <- fitted(model)

    # Check if the prediction matches the true label
    if (as.character(pred) == as.character(test_data$V11)) {
      correct_predictions <- correct_predictions + 1
    }
  }

  # Calculate accuracy for this K value
  accuracy <- correct_predictions / nrow(credit_df)

  # Print the K value and corresponding accuracy
  print(paste("K =", K_value, "-> Accuracy:", accuracy))

  # Store the accuracy for each model
  accuracies <- c(accuracies, accuracy)
}
```

```
## [1] "K = 1 -> Accuracy: 0.814984709480122"
## [1] "K = 2 -> Accuracy: 0.686544342507645"
## [1] "K = 3 -> Accuracy: 0.625382262996942"
## [1] "K = 4 -> Accuracy: 0.576452599388379"
## [1] "K = 5 -> Accuracy: 0.529051987767584"
## [1] "K = 6 -> Accuracy: 0.492354740061162"
## [1] "K = 7 -> Accuracy: 0.463302752293578"
## [1] "K = 8 -> Accuracy: 0.431192660550459"
## [1] "K = 9 -> Accuracy: 0.40519877675841"
## [1] "K = 10 -> Accuracy: 0.382262996941896"
## [1] "K = 11 -> Accuracy: 0.350152905198777"
## [1] "K = 12 -> Accuracy: 0.321100917431193"
## [1] "K = 13 -> Accuracy: 0.288990825688073"
## [1] "K = 14 -> Accuracy: 0.269113149847095"
## [1] "K = 15 -> Accuracy: 0.253822629969419"
## [1] "K = 16 -> Accuracy: 0.23394495412844"
## [1] "K = 17 -> Accuracy: 0.207951070336391"
## [1] "K = 18 -> Accuracy: 0.18348623853211"
## [1] "K = 19 -> Accuracy: 0.157492354740061"
## [1] "K = 20 -> Accuracy: 0.136085626911315"
## [1] "K = 21 -> Accuracy: 0.120795107033639"
## [1] "K = 22 -> Accuracy: 0.0948012232415902"
## [1] "K = 23 -> Accuracy: 0.0856269113149847"
## [1] "K = 24 -> Accuracy: 0.081039755351682"
## [1] "K = 25 -> Accuracy: 0.073394495412844"
## [1] "K = 26 -> Accuracy: 0.0611620795107034"
## [1] "K = 27 -> Accuracy: 0.0535168195718654"
## [1] "K = 28 -> Accuracy: 0.0489296636085627"
## [1] "K = 29 -> Accuracy: 0.0458715596330275"
## [1] "K = 30 -> Accuracy: 0.0397553516819572"
## [1] "K = 31 -> Accuracy: 0.0336391437308868"
## [1] "K = 32 -> Accuracy: 0.0290519877675841"
## [1] "K = 33 -> Accuracy: 0.0259938837920489"
## [1] "K = 34 -> Accuracy: 0.0244648318042813"
## [1] "K = 35 -> Accuracy: 0.0214067278287462"
## [1] "K = 36 -> Accuracy: 0.018348623853211"
## [1] "K = 37 -> Accuracy: 0.018348623853211"
## [1] "K = 38 -> Accuracy: 0.0168195718654434"
## [1] "K = 39 -> Accuracy: 0.0168195718654434"
## [1] "K = 40 -> Accuracy: 0.0168195718654434"
## [1] "K = 41 -> Accuracy: 0.0137614678899083"
## [1] "K = 42 -> Accuracy: 0.0137614678899083"
## [1] "K = 43 -> Accuracy: 0.0122324159021407"
## [1] "K = 44 -> Accuracy: 0.0122324159021407"
## [1] "K = 45 -> Accuracy: 0.00917431192660551"
## [1] "K = 46 -> Accuracy: 0.00764525993883792"
## [1] "K = 47 -> Accuracy: 0.00764525993883792"
## [1] "K = 48 -> Accuracy: 0.00764525993883792"
## [1] "K = 49 -> Accuracy: 0.00611620795107034"
## [1] "K = 50 -> Accuracy: 0.00458715596330275"
```

```
# Find the best K for the scaled model
best_k_scaled <- which.max(accuracies)
best_accuracy_scaled <- max(accuracies)

print(paste("Best K for scaled data:", best_k_scaled))
```

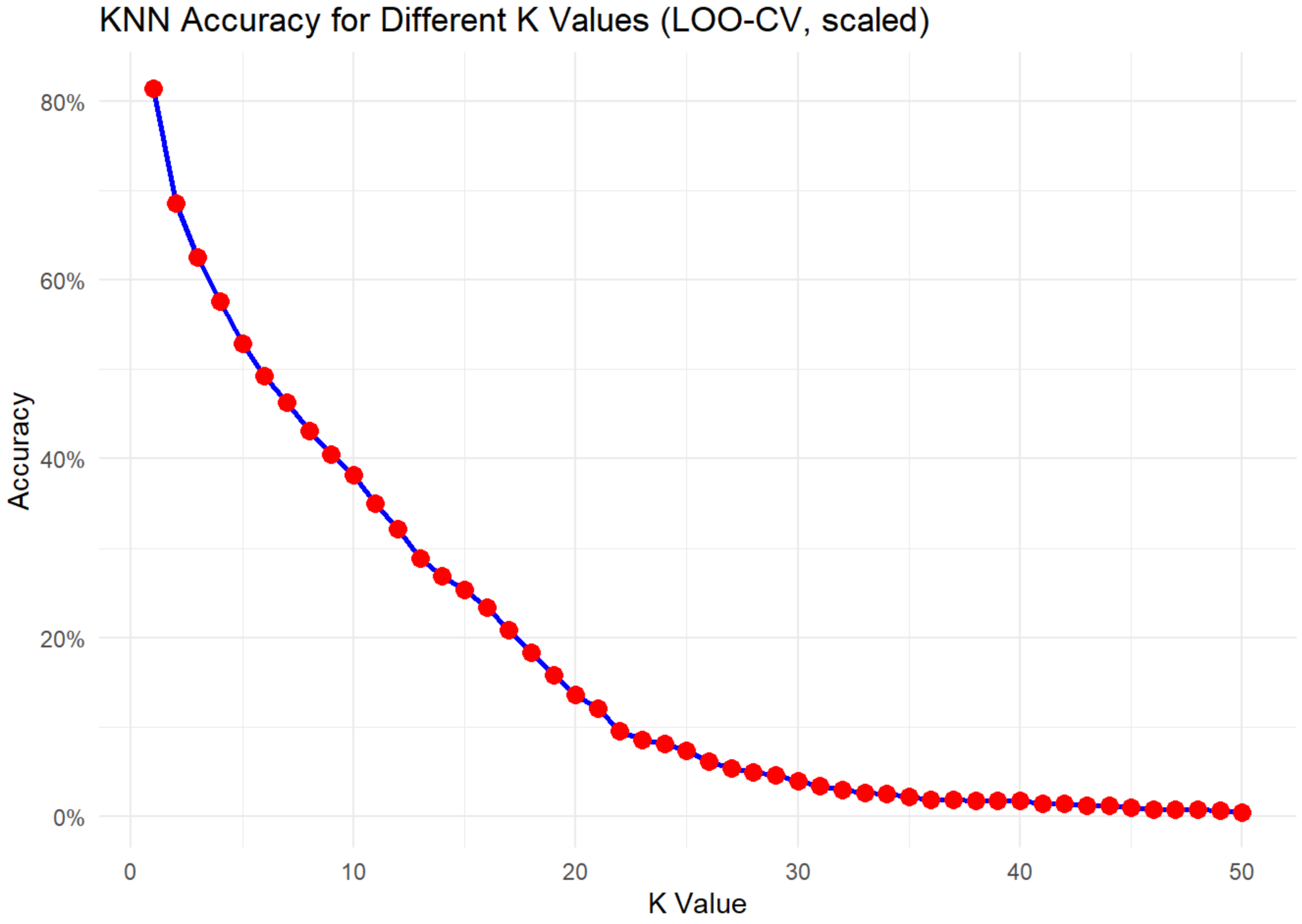
```
## [1] "Best K for scaled data: 1"
```

```
print(paste("Best accuracy for scaled data:", best_accuracy_scaled))
```

```
## [1] "Best accuracy for scaled data: 0.814984709480122"
```

```
# Plotting the accuracy vs K value
ggplot(data.frame(K = 1:50, Accuracy = accuracies), aes(x = K, y = Accuracy)) +
  geom_line(color = "blue", size = 1) +
  geom_point(color = "red", size = 3) +
  labs(title = "KNN Accuracy for Different K Values (LOO-CV, scaled)",
       x = "K Value",
       y = "Accuracy") +
  theme_minimal() +
  scale_y_continuous(labels = scales::percent)
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## ℹ Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```


KNN Accuracy for Different K Values (LOO-CV, scaled)
Accuracy
80%
60%
40%
20%
0%
0
10
20
30
40
50
K Value

unscaled

```
#------------------------- getting a sense of the data
# Reading in the credit card data
credit <- read.table("C://Users//Clair//OneDrive//Documents//Fall 2024//IYSE 6501//hw1//data 2.
2//credit_card_data.txt", sep="\t", header=FALSE)

#------------------------- manipulating the data
# Convert to data frame without scaling the features
credit_df <- as.data.frame(credit)
credit_df$V11 <- as.factor(credit_df$V11)  # Ensure the labels are treated as factors

#------------------------- helper libraries
# install.packages("kknn")
library(kknn)
library(ggplot2)

#------------------------- leave-one-out cross-validation using unscaled data and kknn
accuracies <- c()  # Vector to store accuracy for different K values

# Loop through values of K from 1 to 50
for (K_value in 1:50) {
  correct_predictions <- 0  # Count correct predictions

  # Loop through each row for leave-one-out cross-validation
  for (i in 1:nrow(credit_df)) {

    # Exclude the i-th data point for training
    train_data <- credit_df[-i, ]  # All but the i-th row for training
    test_data <- credit_df[i, , drop = FALSE]  # Only the i-th row for testing

    # Fit the kknn model for this split using unscaled data
    model <- kknn(V11 ~ ., train = train_data, test = test_data, k = K_value)

    # Get the predicted label
    pred <- fitted(model)

    # Check if the prediction matches the true label
    if (as.character(pred) == as.character(test_data$V11)) {
      correct_predictions <- correct_predictions + 1
    }
  }

  # Calculate accuracy for this K value
  accuracy <- correct_predictions / nrow(credit_df)

  # Print the K value and corresponding accuracy
  print(paste("K =", K_value, "-> Accuracy:", accuracy))

  # Store the accuracy for each model
  accuracies <- c(accuracies, accuracy)
}
```

```
## [1] "K = 1 -> Accuracy: 0.814984709480122"
## [1] "K = 2 -> Accuracy: 0.814984709480122"
## [1] "K = 3 -> Accuracy: 0.814984709480122"
## [1] "K = 4 -> Accuracy: 0.814984709480122"
## [1] "K = 5 -> Accuracy: 0.851681957186544"
## [1] "K = 6 -> Accuracy: 0.845565749235474"
## [1] "K = 7 -> Accuracy: 0.847094801223242"
## [1] "K = 8 -> Accuracy: 0.848623853211009"
## [1] "K = 9 -> Accuracy: 0.847094801223242"
## [1] "K = 10 -> Accuracy: 0.850152905198777"
## [1] "K = 11 -> Accuracy: 0.851681957186544"
## [1] "K = 12 -> Accuracy: 0.853211009174312"
## [1] "K = 13 -> Accuracy: 0.851681957186544"
## [1] "K = 14 -> Accuracy: 0.851681957186544"
## [1] "K = 15 -> Accuracy: 0.853211009174312"
## [1] "K = 16 -> Accuracy: 0.851681957186544"
## [1] "K = 17 -> Accuracy: 0.851681957186544"
## [1] "K = 18 -> Accuracy: 0.851681957186544"
## [1] "K = 19 -> Accuracy: 0.850152905198777"
## [1] "K = 20 -> Accuracy: 0.850152905198777"
## [1] "K = 21 -> Accuracy: 0.848623853211009"
## [1] "K = 22 -> Accuracy: 0.847094801223242"
## [1] "K = 23 -> Accuracy: 0.844036697247706"
## [1] "K = 24 -> Accuracy: 0.845565749235474"
## [1] "K = 25 -> Accuracy: 0.845565749235474"
## [1] "K = 26 -> Accuracy: 0.844036697247706"
## [1] "K = 27 -> Accuracy: 0.840978593272171"
## [1] "K = 28 -> Accuracy: 0.837920489296636"
## [1] "K = 29 -> Accuracy: 0.839449541284404"
## [1] "K = 30 -> Accuracy: 0.840978593272171"
## [1] "K = 31 -> Accuracy: 0.837920489296636"
## [1] "K = 32 -> Accuracy: 0.836391437308868"
## [1] "K = 33 -> Accuracy: 0.834862385321101"
## [1] "K = 34 -> Accuracy: 0.833333333333333"
## [1] "K = 35 -> Accuracy: 0.831804281345566"
## [1] "K = 36 -> Accuracy: 0.831804281345566"
## [1] "K = 37 -> Accuracy: 0.831804281345566"
## [1] "K = 38 -> Accuracy: 0.831804281345566"
## [1] "K = 39 -> Accuracy: 0.831804281345566"
## [1] "K = 40 -> Accuracy: 0.831804281345566"
## [1] "K = 41 -> Accuracy: 0.831804281345566"
## [1] "K = 42 -> Accuracy: 0.834862385321101"
## [1] "K = 43 -> Accuracy: 0.834862385321101"
## [1] "K = 44 -> Accuracy: 0.836391437308868"
## [1] "K = 45 -> Accuracy: 0.839449541284404"
## [1] "K = 46 -> Accuracy: 0.840978593272171"
## [1] "K = 47 -> Accuracy: 0.837920489296636"
## [1] "K = 48 -> Accuracy: 0.839449541284404"
## [1] "K = 49 -> Accuracy: 0.839449541284404"
## [1] "K = 50 -> Accuracy: 0.837920489296636"
```

```
#------------------------ output final accuracies for each model
print("Accuracies for each KNN model with varying K:")
```

```
## [1] "Accuracies for each KNN model with varying K:"
```

```
print(accuracies)
```

```
##  [1] 0.8149847 0.8149847 0.8149847 0.8149847 0.8516820 0.8455657 0.8470948
##  [8] 0.8486239 0.8470948 0.8501529 0.8516820 0.8532110 0.8516820 0.8516820
## [15] 0.8532110 0.8516820 0.8516820 0.8516820 0.8501529 0.8501529 0.8486239
## [22] 0.8470948 0.8440367 0.8455657 0.8455657 0.8440367 0.8409786 0.8379205
## [29] 0.8394495 0.8409786 0.8379205 0.8363914 0.8348624 0.8333333 0.8318043
## [36] 0.8318043 0.8318043 0.8318043 0.8318043 0.8318043 0.8318043 0.8348624
## [43] 0.8348624 0.8363914 0.8394495 0.8409786 0.8379205 0.8394495 0.8394495
## [50] 0.8379205
```

```
print("Max accuracy for leave-one-out cross-validation:")
```

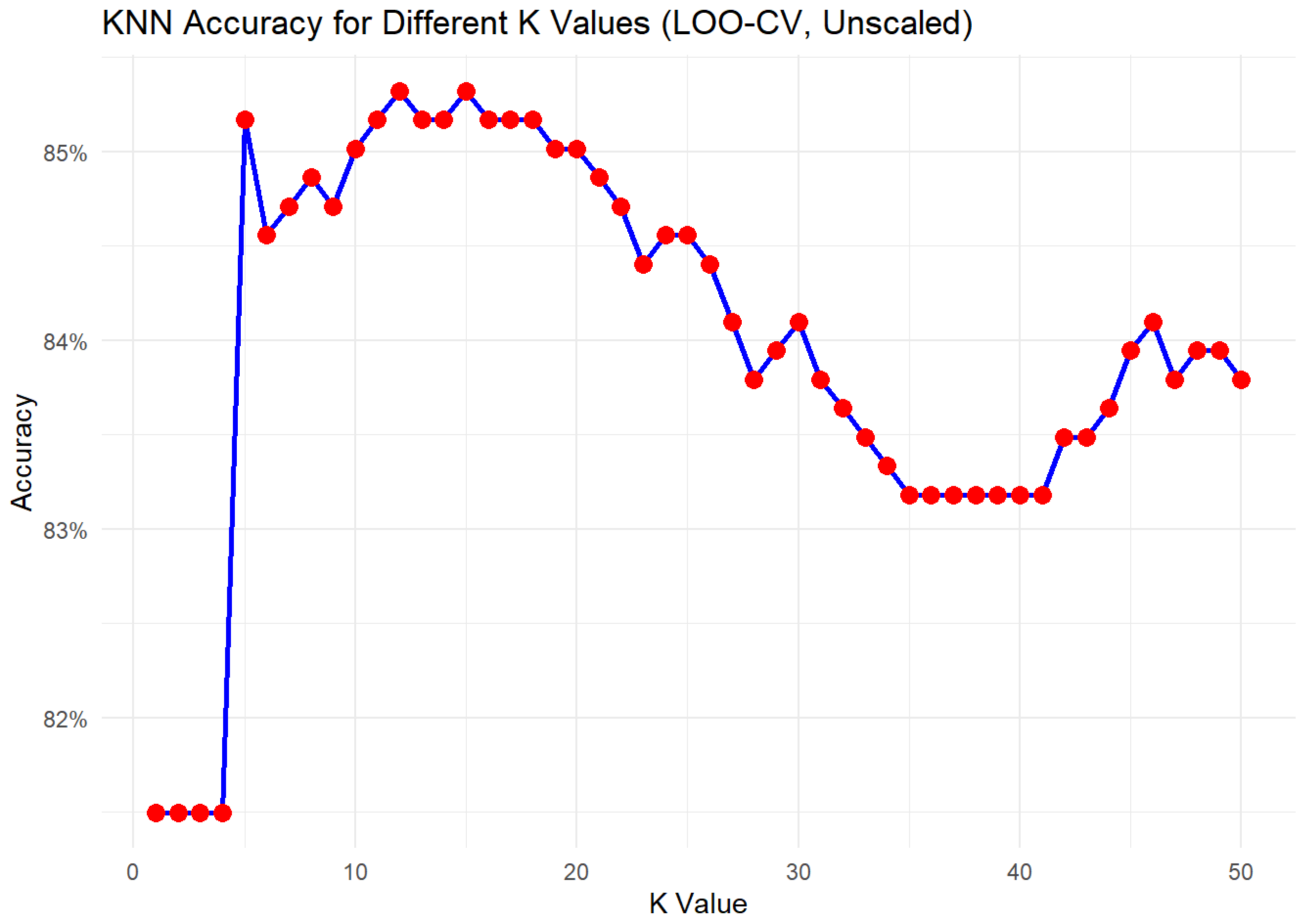
```
## [1] "Max accuracy for leave-one-out cross-validation:"
```

```
print(max(accuracies))
```

```
## [1] 0.853211
```

```
#------------------------ plotting the results using ggplot2
# Create a dataframe with K values and corresponding accuracies
accuracy_data <- data.frame(
  K = 1:50,
  Accuracy = accuracies
)

# Plotting the accuracy vs K value
ggplot(accuracy_data, aes(x = K, y = Accuracy)) +
  geom_line(color = "blue", size = 1) +
  geom_point(color = "red", size = 3) +
  labs(title = "KNN Accuracy for Different K Values (LOO-CV, Unscaled)",
       x = "K Value",
       y = "Accuracy") +
  theme_minimal() +
  scale_y_continuous(labels = scales::percent)
```



```
# Find the best K for the unscaled model
best_k_unscaled <- which.max(accuracies)
best_accuracy_unscaled <- max(accuracies)

print(paste("Best K for unscaled data:", best_k_unscaled))
```

```
## [1] "Best K for unscaled data: 12"
```

```
print(paste("Best accuracy for unscaled data:", best_accuracy_unscaled))
```

```
## [1] "Best accuracy for unscaled data: 0.853211009174312"
```