BSAN750 Homework 4

Name: John Kingsley

Due: 2023-09-25

**Instruction:** This is an individual assignment. Please directly work on the attached Rmarkdown file and generate a clean and well-organized word/pdf document using the given layout. However, if you can’t successfully compile the rmarkdown file, you should use the attached word document, and manually organize and format your work as if the rmarkdown file is successfully compiled.

# 1 Briefly explain how k-nearest neighbor algorithm works.

**Answer:**

The KNN algorithm works by initially finding the euclidean distances between each value in a testing set from every value in a corresponding training set. We can imagine these distances are stored in a list of length(testing set), where each list index contains a vector of euclidean distances nrow(training set) long.

After these distances are found, the classification values (in the case of our in-class example, Y.train) corresponding to the top k smallest distances are selected as the K-nearest neighbors. By a majority vote, the most popular classification value is selected as the predicted cluster of which the “i”th value in the testing set belongs to.

# 2 Apply the k-nearest neighbor algorithm to the “seed” dataset by the following steps:

## 2.1 Use the following code to load the data

**R code:**

seed <- read.table('http://archive.ics.uci.edu/ml/machine-learning-databases/00236/seeds\_dataset.txt', header=F)  
names(seed) <- c("area", "perimeter","campactness", "length", "width", "asymmetry", "groovelength", "type")

The variable “type” is the type of seed which is the response variable (the class label). The rest variables are measures of seed (features) that are used as predictors.

## 2.2 Standardize the data except for the last column, which is the label of type.

**R code:**

seed1 <- data.frame(scale(seed[,-8]))  
seed1$type <- seed$type

## 2.3 Randomly split the data to 80% training and 20% testing samples.

**R code:**

n <- nrow(seed1)  
index <- sample(n, 0.8\*n)  
training <- seed1[index,]  
testing <- seed1[-index,]

## 2.4 Use 5-nearest neighbor to predict the wheat type for the testing sample. Report the confusion matrix and the overall missclassification error.

**R code and results:**

library(class)  
fit\_k5 <- knn(training[,-8], testing[,-8], training[,8], k=5)  
  
# Confusion Matrix  
cm <- table(Actual = testing[, 8], Predicted = fit\_k5)  
cm

## Predicted  
## Actual 1 2 3  
## 1 9 0 0  
## 2 1 19 0  
## 3 0 0 13

# Misclassification Error  
mean(fit\_k5!=testing[,8])

## [1] 0.02380952

**Describe and comment the results:**

**Confusion Matrix**

The confusion matrix shows the count of true positives or negatives (diagonal elements), false positives, and false negatives for each class. The rows of the matrix represent the actual class labels, and the columns represent the predicted class labels.

If class 1 is the “positive” class:

The model correctly made 11 predictions of class 1 when the actual class was 1 ([1,1]). These are true positives.

The model incorrectly made 2 predictions of class 2 when the actual class was 1 ([1,2]). These are false negatives.

The model incorrectly made 1 predictions of class 3 when the actual class was 1 ([1,3]). This is also false negative.

The model correctly made 13 predictions of class 2 when the actual class was 2 ([2,2]). These are true negatives if we are considering class 1 as our positive class.

The model correctly made 14 predictions of class 3 when the actual class was 3 ([2,2]). Again, these are true negatives if we are considering class 1 as our positive class.

The model incorrectly made 1 prediction of our “positive” class 1 when the actual class was 3 ([3,1]). This is the only false positive.

**Misclassification Error**

This calculates the proportion of incorrectly classified instances and is a measure of model performance.

In our case, this was approximately ~9.5%. The model is misclassifying this proportion of instances in the data set.

## 2.5 Re-do the previous question (only calculate the missclassification error) with 5-fold cross-validation. (Hint: you may follow the steps below.)

* Instead of a random split, you would randomly divide the data to 5 portions with each 20%.
* Then for each 20% data, use it as testing and the rest 80% as training sample, so that you can repeat previous two questions.
* Do this 5 times for the 5 portions and you should obtain 5 missclassification errors.
* Finally, calculate the average of these 5 missclassification errors.

**R code and results:**

fold\_index <- sample(5, n, replace=TRUE)  
me <- rep(NA, 5)  
  
for(i in 1:5){  
 train\_fold <- seed1[fold\_index!=i,]  
 test\_fold <- seed1[fold\_index==i,]  
 fit\_k5 <- knn(train\_fold[,-8], test\_fold[,-8], train\_fold[,8], k=5)  
 me[i] <- mean(fit\_k5!=test\_fold[,8])  
}  
  
cv\_score <- mean(me)  
cv\_score

## [1] 0.07064047

**Describe and comment the results:**

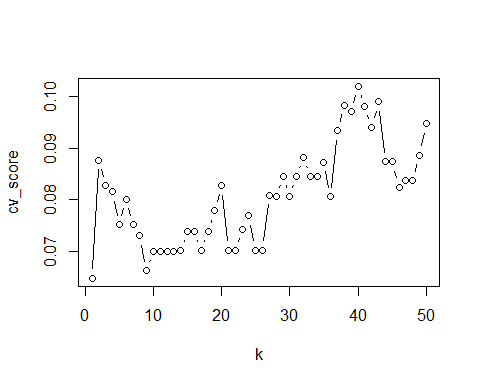
Our CV score of ~0.0649 indicates that the model’s performance is pretty good. A lower value like this suggests that the model has a low misclassification rate on average when making predictions on unseen data.

On average, the model’s predictions are incorrect for only ~6.49% of the instances in the test data. This is a positive sign and suggests that the model is effective in making accurate predictions.

## 2.6 Now, repeat previous question (5-fold cross-validation) by varying k (parameter for knn) from 1 to 50. How does the missclassification error changes across k? You may draw a figure to show this.

**R code and results:**

fold\_index <- sample(5, n, replace=TRUE)  
cv\_score <- rep(NA, 50)  
  
for(k in 1:50){  
 me <- rep(NA, 5)  
 for(i in 1:5){  
 train\_fold <- seed1[fold\_index!=i,]  
 test\_fold <- seed1[fold\_index==i,]  
 fit\_k50 <- knn(train\_fold[,-8], test\_fold[,-8], train\_fold[,8], k=k)  
 me[i] <- mean(fit\_k50!=test\_fold[,8])  
 }  
 cv\_score[k] <- mean(me)  
}  
  
plot(cv\_score, type='b', xlab="k")



best\_k <- which.min(cv\_score)  
best\_k

## [1] 1

**Describe and comment the results:**

Looking at the plot of CV scores from k=1:50, we can see that increasing k beyond 5 did nothing to improve our model. In fact, our CV score got progressively worse as we increased k beyond 5.

# 3 Complete Q5 and Q6 in Exercise 4.

For Q6 (writing your own function of knn), as previous homework, it is okay that your function may not work, but you are expected to put effort and try your best to write this function.

**In-class example:**

iris1 <- data.frame(scale(iris[,-5])) ## scale  
iris1$Species <- iris$Species  
  
## split data to training and testing  
n <- nrow(iris1)  
index <- sample(n, 0.7\*n)  
training <- iris1[index,]  
testing <- iris1[-index,]  
  
library(class)  
fit\_k3 <- knn(training[,-5], testing[,-5], training[,5], k=3)

**myknn():**

myknn <- function(X.train, X.test, Y.train, k){  
   
 Eudist <- function (x, y) sqrt(sum((x - y) ^2))  
  
 d <- vector("list", length = nrow(X.test))  
  
 for(i in 1:nrow(X.test)){  
 d[[i]] <- numeric(nrow(X.train))  
 for(j in 1:nrow(X.train)){  
 d[[i]][j] <- Eudist(X.test[i,], X.train[j,])  
 }  
 }  
  
 neighbors <- vector("list", length=length(d))  
  
 for (i in 1:length(d)) {  
 sorted\_indices <- order(d[[i]])  
 top\_k\_indices <- sorted\_indices[1:k]  
 neighbors[[i]] <- Y.train[top\_k\_indices]  
 }  
  
 classes <- vector("numeric", length = length(neighbors))  
  
 for (i in 1:length(neighbors)) {  
 neighbor\_values <- unlist(neighbors[[i]])  
 counts <- table(neighbor\_values)  
 majority\_vote <- names(counts)[which.max(counts)]  
 classes[i] <- majority\_vote  
 }  
   
 return(classes)  
}

**Does it work?**

classes <- myknn(training[,-5], testing[,-5], training[,5], 3)  
  
classes==fit\_k3 # it works!

## [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  
## [16] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  
## [31] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

# 4 Briefly explain bias-variance tradeoff.

**Answer:**

A high-bias model makes strong assumptions about the data and is too simplistic to capture complex relationships. This kind of model tends to underfit the data and won’t perform well on training data or unseen/test data.

High-variance models are too sensitive to small fluctuations or noise in the training data. Such models tend to overfit the training data, performing well on training data but poorly on unseen/test data.

Increasing model complexity tends to decrease bias but increase variance, and vice versa.

Models that are highly flexible (high variance, low bias) can fit their training data very closely but tend to make poor generalizations. Simpler models (low variance, high bias) make strong assumptions about their data and may not fit their data closely, tending to generalize better.

Finding a balance is crucial for building models that generalize well to unseen data, and techniques such as cross validation are good for accomplishing this.