ISYE 6501 HOMEWORK # 2

R Markdown

Question 3.1 Using the same data set (credit_card_data.txt or credit_card_data-headers.txt) as in Question 2.2, use the ksvm or kknn function to find a good classifier: (a) using cross-validation (do this for the k-nearest-neighbors model; SVM is optional); and (b) splitting the data into training, validation, and test data sets (pick either KNN or SVM; the other is optional).

The code below does a few things. As always, call the necessary libraries needed for the assignment. I prefer using rio to import datasets because the function is very simple ('import'). Next, I create a random list of numbers, whose values span from 1 to the number of entries in the credit card data set. This randomizes the indices only and is the foundation for creating my training and test sets.

```
#housekeeping
library(pacman)
pacman::p_load(rio, kknn, datasets,factoextra, ggplot2)

#import dataset (without header)
data <- (import("D:/[REDACTED]

#randomize data
set.seed(123)
n <- nrow(data) #number of entries in data set
randindex <- sample(1:n) #randomize order of indices
```

The first part of the next code takes the first 70% of the randomized indices and stores them in a set that I've dubbed aIndexTrain. The remaining 30% go into a set of indices that will be used for a test set. Keep in mind these two new sets/lists only contain randomized indices. The next part of the code is what creates the actual training and testing set by searching the dataset, which I've named 'data', and pulling the values/information at the randomized indices stored in the aIndexTrain and aIndexTest

```
#split randomized indices
aIndexTrain <- randindex[1:round(0.7 * n)] #70% for training
aIndexTest <- randindex[(round(0.7 * n) + 1):n] #30% for test
```

```
#create sets using randomized indices
trainingSet <- data[aIndexTrain,]
testSet <- data[aIndexTest,]
```

Next, I train a kNN model using Leave One Out Cross Validation. It uses the trainingSet. Each datapoint, except V11, is used as training to predict the data in V11, which is our response variable. K=100 means I am training the kNN model for k values 1-100. Then I print the results, which tells me the optimal kernel and k value to use. In this case, the best K=48, and best kernel=optimal.

```
# Leave one out cross validation to find k value
loocv<-train.kknn((V11)~., data=trainingSet, kmax=100, scale=TRUE)
#model results
summary(loocv)

##
## Call:
## train.kknn(formula = (V11) ~ ., data = trainingSet, kmax = 100,
scale = TRUE)
##
## Type of response variable: continuous
## minimal mean absolute error: 0.1921397
## Minimal mean squared error: 0.1059062
## Best kernel: optimal
## Best k: 48
```

Now I need to see how accurate the training set is. First, I initialize a vector filled with 0's. The length of the vector matches the amount of rows in the training set. I create another kNN model, named trainmodel (training model), which, again, uses LOOCV. This time I have k = 48, and kernel = optimal. predTrain (Predictions for the Training set) stores a value for each [i] it iterates to, stopping once it finishes the amount of rows in the data set. The value it stores is it's predictions for row V11. The value is rounded to 0 or 1. Since V11 is the response variable and is made up of 0s and 1s, it was necessary to round so I can find the overrall accuracy of my training set. I do a binary comparison between the predicitons and the actual values in V11 of the trainingSet which is then divded by the total rows in the trainingSet. The accuracy is approx 84%.

```
#training the accuracy
predTrain <- rep(0,(nrow(trainingSet))) #vector of 0s for training set
predictions

#perform loocv via knn on training set
```

```r
for (i in 1:nrow(trainingSet)){ #using loocv best k and kernel
  trainmodel <- kknn((V11)~., trainingSet[-i,],
trainingSet[i,],k=48,kernel="optimal", scale = TRUE)
  predTrain[i]<- as.integer(fitted(trainmodel)+0.5) #rounding to 0 or
1
}
accTrain <- sum(predTrain == trainingSet$V11) / nrow(trainingSet)
print(paste0("Training data accuracy: ", round(accTrain * 100), "%"))

## [1] "Training data accuracy: 84%"
```

The next chunk is similar to the previous. This time, I am running the same code on the test set to evaluate the performance and accuracy of my trainingset. The process is the same: initalize a vector, make predictions at each i, round and store each prediction, and finally compare. The accuracy of the testSet is approx 85%. Note: Originally, I had split the data 80/20. The accuracy of my training set was 83% and the accuracy of my test set was 80%. The drop in accuracy was likely due to overfitting of the training set. It is also important to note that a higher test accuracy can occur due to randomness. However, since the training set also increased in accuracy despite training on 10% less data, and that the test accuracy is higher than the training accuracy (by 1%), I will assume that it likely this split performs better generalization on unseen data, so I am keeping the 70/30 split.

```r
#Testing accuracy from training set
predTest<- rep(0,(nrow(testSet))) #vector of 0s for test set
predictions

#perform same loop on test set
for (i in 1:nrow(testSet)){
  testmodel <- kknn((V11)~., testSet[-i,],
testSet[i,],k=48,kernel="optimal", scale = TRUE)
  predTest[i]<- as.integer(fitted(testmodel)+0.5) #rounding to 0 or 1
}
accTest <- sum(predTest == testSet$V11) / nrow(testSet)
print(paste0("Test data accuracy: ", round(accTest * 100), "%"))

## [1] "Test data accuracy: 85%"
```

Question 3.2B asks us to split the data into 3 sets: training, validation, test. Similarly, to part A, I create a randomzied list of indices, and split the randomized indices into their respective index sets. Then I create the actual data sets which pull the information from the credit card data set at the randomized indices. I did 70% for the training set again. The remaining 30%

gets split into 15% validation and 15% test. I also initialize the set to store the predictions of the training test and set kmax = 100 for the knn model.

```
#Question 3.2b
set.seed(123)
bRandIndex <- sample(1:n) #randomize data set (n = total entries,
initialized prior)
#split random indices into sets
indexTrain <- bRandIndex[1:round(0.7 * n)]
indexVal <- bRandIndex[(round(0.7 * n) + 1):(round(0.85 * n))]
indexTest <- bRandIndex[(round(0.85 * n) + 1):n]
#create actual sets
bTrain <- data[indexTrain,] #training set
bVal <- data[indexVal,] #validation set for tuning
bTest <- data[indexTest,] #test set for evaluating
kmax <- 100
bAccTrain <- rep(0,kmax) # b Accuracy Training
```

The following section uses the training set to build kNN models for k values from 1-100 and then evaluates the accuracy for each k.

```
for (k in 1:kmax) {
  #fit knn on training set, validate on val set
  knntrainmodel <- kknn(V11~., bTrain, bVal, k = k, scale = TRUE)
  #compare each k result using val set
  bPredTrain <- as.integer(fitted(knntrainmodel)+0.5) # round to 0 or
1
  bAccTrain[k] <- sum(bPredTrain == bVal$V11) / nrow(bVal)
}
#accuracy for each k
for (k in 1:kmax) {
  print(paste0("k = ", k, ": Validation Accuracy = ",
round(bAccTrain[k] * 100), "%"))
}

## [1] "k = 1: Validation Accuracy = 83%"
## [1] "k = 2: Validation Accuracy = 83%"
## [1] "k = 3: Validation Accuracy = 83%"
## [1] "k = 4: Validation Accuracy = 83%"
## [1] "k = 5: Validation Accuracy = 85%"
## [1] "k = 6: Validation Accuracy = 87%"
## [1] "k = 7: Validation Accuracy = 90%"
## [1] "k = 8: Validation Accuracy = 90%"
## [1] "k = 9: Validation Accuracy = 90%"
## [1] "k = 10: Validation Accuracy = 89%"
## [1] "k = 11: Validation Accuracy = 88%"
## [1] "k = 12: Validation Accuracy = 87%"
## [1] "k = 13: Validation Accuracy = 86%"
## [1] "k = 14: Validation Accuracy = 86%"
## [1] "k = 15: Validation Accuracy = 86%"
## [1] "k = 16: Validation Accuracy = 84%"
## [1] "k = 17: Validation Accuracy = 84%"
## [1] "k = 18: Validation Accuracy = 84%"
## [1] "k = 19: Validation Accuracy = 84%"
## [1] "k = 20: Validation Accuracy = 84%"
## [1] "k = 21: Validation Accuracy = 84%"
## [1] "k = 22: Validation Accuracy = 84%"
## [1] "k = 23: Validation Accuracy = 85%"
## [1] "k = 24: Validation Accuracy = 85%"
## [1] "k = 25: Validation Accuracy = 85%"
## [1] "k = 26: Validation Accuracy = 85%"
## [1] "k = 27: Validation Accuracy = 85%"
## [1] "k = 28: Validation Accuracy = 85%"
## [1] "k = 29: Validation Accuracy = 85%"
## [1] "k = 30: Validation Accuracy = 85%"
## [1] "k = 31: Validation Accuracy = 85%"
## [1] "k = 32: Validation Accuracy = 85%"
## [1] "k = 33: Validation Accuracy = 85%"
## [1] "k = 34: Validation Accuracy = 85%"
## [1] "k = 35: Validation Accuracy = 84%"
## [1] "k = 36: Validation Accuracy = 84%"
## [1] "k = 37: Validation Accuracy = 85%"
## [1] "k = 38: Validation Accuracy = 85%"
## [1] "k = 39: Validation Accuracy = 85%"
## [1] "k = 40: Validation Accuracy = 85%"
## [1] "k = 41: Validation Accuracy = 85%"
## [1] "k = 42: Validation Accuracy = 85%"
## [1] "k = 43: Validation Accuracy = 84%"
## [1] "k = 44: Validation Accuracy = 84%"
## [1] "k = 45: Validation Accuracy = 84%"
## [1] "k = 46: Validation Accuracy = 84%"
## [1] "k = 47: Validation Accuracy = 84%"
## [1] "k = 48: Validation Accuracy = 84%"
## [1] "k = 49: Validation Accuracy = 84%"
## [1] "k = 50: Validation Accuracy = 84%"
## [1] "k = 51: Validation Accuracy = 84%"
## [1] "k = 52: Validation Accuracy = 84%"
## [1] "k = 53: Validation Accuracy = 84%"
## [1] "k = 54: Validation Accuracy = 85%"
## [1] "k = 55: Validation Accuracy = 85%"
## [1] "k = 56: Validation Accuracy = 85%"
## [1] "k = 57: Validation Accuracy = 85%"
## [1] "k = 58: Validation Accuracy = 85%"
## [1] "k = 59: Validation Accuracy = 85%"
## [1] "k = 60: Validation Accuracy = 85%"
## [1] "k = 61: Validation Accuracy = 85%"
## [1] "k = 62: Validation Accuracy = 85%"
## [1] "k = 63: Validation Accuracy = 85%"
## [1] "k = 64: Validation Accuracy = 84%"
## [1] "k = 65: Validation Accuracy = 84%"
## [1] "k = 66: Validation Accuracy = 84%"
## [1] "k = 67: Validation Accuracy = 84%"
## [1] "k = 68: Validation Accuracy = 84%"
## [1] "k = 69: Validation Accuracy = 84%"
## [1] "k = 70: Validation Accuracy = 84%"
## [1] "k = 71: Validation Accuracy = 84%"
## [1] "k = 72: Validation Accuracy = 84%"
## [1] "k = 73: Validation Accuracy = 84%"
## [1] "k = 74: Validation Accuracy = 84%"
## [1] "k = 75: Validation Accuracy = 84%"
## [1] "k = 76: Validation Accuracy = 84%"
## [1] "k = 77: Validation Accuracy = 85%"
## [1] "k = 78: Validation Accuracy = 85%"
## [1] "k = 79: Validation Accuracy = 85%"
## [1] "k = 80: Validation Accuracy = 85%"
## [1] "k = 81: Validation Accuracy = 85%"
## [1] "k = 82: Validation Accuracy = 85%"
## [1] "k = 83: Validation Accuracy = 85%"
## [1] "k = 84: Validation Accuracy = 85%"
## [1] "k = 85: Validation Accuracy = 85%"
## [1] "k = 86: Validation Accuracy = 85%"
## [1] "k = 87: Validation Accuracy = 85%"
## [1] "k = 88: Validation Accuracy = 85%"
## [1] "k = 89: Validation Accuracy = 85%"
## [1] "k = 90: Validation Accuracy = 85%"
## [1] "k = 91: Validation Accuracy = 85%"
## [1] "k = 92: Validation Accuracy = 85%"
## [1] "k = 93: Validation Accuracy = 85%"
## [1] "k = 94: Validation Accuracy = 85%"
## [1] "k = 95: Validation Accuracy = 85%"
## [1] "k = 96: Validation Accuracy = 85%"
## [1] "k = 97: Validation Accuracy = 85%"
## [1] "k = 98: Validation Accuracy = 85%"
## [1] "k = 99: Validation Accuracy = 85%"
## [1] "k = 100: Validation Accuracy = 85%"
```

I create a variable bestk, which is the best k value from the previous code. My finding indicate that k = 7 is the best k value and it yields an accuracy of 90%

```r
#find best K value
bestk <- which.max(bAccTrain)
#print best k value
print(paste0("Best k = ", bestk, " with Validation Accuracy = ",
round(bAccTrain[bestk] * 100), "%"))

## [1] "Best k = 7 with Validation Accuracy = 90%"
```

Finally, I evaluate my chosen models performance on the test set. Same code as before, but now using the training set against the test set. The accuracy at best k (k = 7) is 83%. The drop in accuracy can be due to randomness or overfitting the validation set. The model still performs relatively well. Some things I can take to try and increase this would be playing around with the split %.

```r
#test data (test acc of training set)
set.seed(123)
knntestmodel <- kknn(V11~., bTrain, bTest, k = bestk, scale = TRUE)
bPredTest <- as.integer((fitted(knntestmodel) + 0.5)) #round to 0 or 1
for binary comparison
bAccTest <- sum(bPredTest == bTest$V11) / nrow(bTest)
print(paste0("Test data using best k, k = ", bestk, " yields accuracy
= ", round(bAccTest * 100), "%"))

## [1] "Test data using best k, k = 7 yields accuracy = 83%"
```

Question 4.1 Describe a situation or problem from your job, everyday life, current events, etc., for which a clustering model would be appropriate. List some (up to 5) predictors that you might use.

Right now I am unemployed and primarily doing grad school and looking for internships. I go to the gym 4x a week and usually cook for myself since ordering food as gotten expensive (and kinda yucky?? Recently I saw one of the delivery ppl drinking from my drink 😢) Anyway, I've been practicing cooking and am always looking for new recipes. I could probably use a clustering model to group recipes. Predictors would be:

1. Cuisine type - indian, italian, asian, american

2. Number of ingredients - if its a simple recipe, or requires a lot of spices and ingredients
3. Prep time - even simple recipes might take a lot of prep time
4. Main ingredient - is it a meat dish/veggie?

5. Protein - I care a lot about how much protein a dish has compared to its other macros/calories

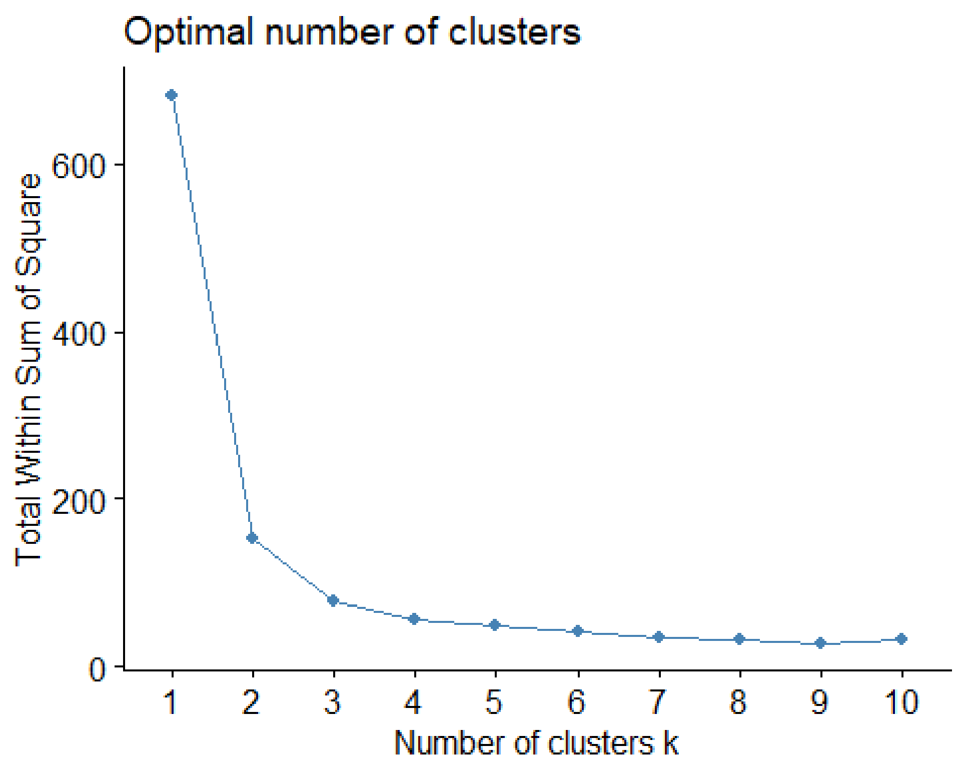
Question 4.2 The iris data set iris.txt contains 150 data points, each with four predictor variables and one categorical response. The predictors are the width and length of the sepal and petal of flowers and the response is the type of flower. The data is available from the R library datasets and can be accessed with iris once the library is loaded. It is also available at the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Iris). The response values are only given to see how well a specific method performed and should not be used to build the model. Use the R function kmeans to cluster the points as well as possible. Report the best combination of predictors, your suggested value of k, and how well your best clustering predicts flower type.

First thing I did was load the 'iris' dataset into my own set irisData. Unlike the credit card data, the 'iris' dataset has a response variable is non-numeric. I converted and mapped that part so I can have a numerical response variable, which we will use to calculate accuracy.

```r
#Question 4.2
irisData <- iris #load data into new variable
#convert species (response variable) to number
#we will use the numerical value to calc accuracy
mapping <- c("setosa" = 1, "versicolor" = 2, "virginica" = 3)
irisData$Species <- mapping[irisData$Species]
```

The next step I took was initializing a function to perform kmeans, The function takes two arguments: a dataset, and a k value. Then I used the elbow method using the fviz_nbclust function (from factoextra) to find the optimal k value. Optimal k = 3. Note: You'll notice I exlcuded the 'species' column (irisData[-5]). The 'species' column is the response variable for this dataset. This is important because clustering is unsupervised. Kmeans clustering does not include labels or response variables when clustering. It is imperative to only use the predictors to avoid impacting the results of the clustering (ie. misleading clusters/artifical clustering/overfitting)

```r
#initialize function that performs kmean and return wss
performKMeans <- function(data, k) {
  kmeans(data, centers = k, nstart = 25)$tot.withinss
}
#elbow method/WSS used to find optimal k
fviz_nbclust(irisData[-5], kmeans, method = "wss")
```



```r
optimalK <- 3
```

To find the best combination of predictors. I listed them all in a vector called ComboList. I exlcuded single predictors, since I want at least 2 predictors. I created an empty list to store the Within Square of Sums (Wss) values. I looped over the combinations in the list and used the function I made earlier to perform kmeans on each combination when k = 3 (optimal K value from elbow method).

```r
#combo of predictors
ComboList <- list(
  c("Sepal.Length", "Sepal.Width"),
  c("Sepal.Length", "Petal.Length"),
  c("Sepal.Length", "Petal.Width"),
  c("Sepal.Width", "Petal.Length"),
  c("Sepal.Width", "Petal.Width"),
  c("Petal.Length", "Petal.Width"),
  c("Sepal.Length", "Sepal.Width", "Petal.Length"),
  c("Sepal.Length", "Sepal.Width", "Petal.Width"),
  c("Sepal.Length", "Petal.Length", "Petal.Width"),
  c("Sepal.Width", "Petal.Length", "Petal.Width"),
  c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")
)
```

```r
#create list for results(wss)
results <- list()

#test each combo
for (predictors in ComboList) {
  subsetData <- irisData[, predictors]
  wssValues <- performKMeans(subsetData, optimalK)

  #store wss for each combo in results
  results[[paste(predictors, collapse = ", ")]] <- wssValues
}

#prints results
print(results)
```

```
## $`Sepal.Length, Sepal.Width`
## [1] 37.0507
##
## $`Sepal.Length, Petal.Length`
## [1] 53.80998
##
## $`Sepal.Length, Petal.Width`
## [1] 32.72653
##
## $`Sepal.Width, Petal.Length`
## [1] 40.73707
##
## $`Sepal.Width, Petal.Width`
## [1] 20.6024
##
## $`Petal.Length, Petal.Width`
## [1] 31.37136
##
## $`Sepal.Length, Sepal.Width, Petal.Length`
## [1] 69.42974
##
## $`Sepal.Length, Sepal.Width, Petal.Width`
## [1] 48.66078
##
## $`Sepal.Length, Petal.Length, Petal.Width`
## [1] 63.34212
##
## $`Sepal.Width, Petal.Length, Petal.Width`
## [1] 47.86643
##
```

```
## $`Sepal.Length, Sepal.Width, Petal.Length, Petal.Width`
## [1] 78.85144
```

Since WSS measures variance within a cluster, a smaller WSS value indicates better clustering. I'm ignoring the wss value for all 4 predictors and instead looking at 2-predictor and 3-predictor combinations. 'sepal.width & petal.width'had the lowest wss of the 2-pred combos (20.6024) and 'sepal.width, petal.length, petal.width' had the lowest wss amongst the 3-pred combos (47.86643). I ended up going with the 3-Predictor combination 'sepal.width, petal.length, petal.width' and put them into a vector called bestpredictors. I then filtered the dataset by only using those columns in bestpredictors. I will explain my decision for choosing that combination in the coming sections

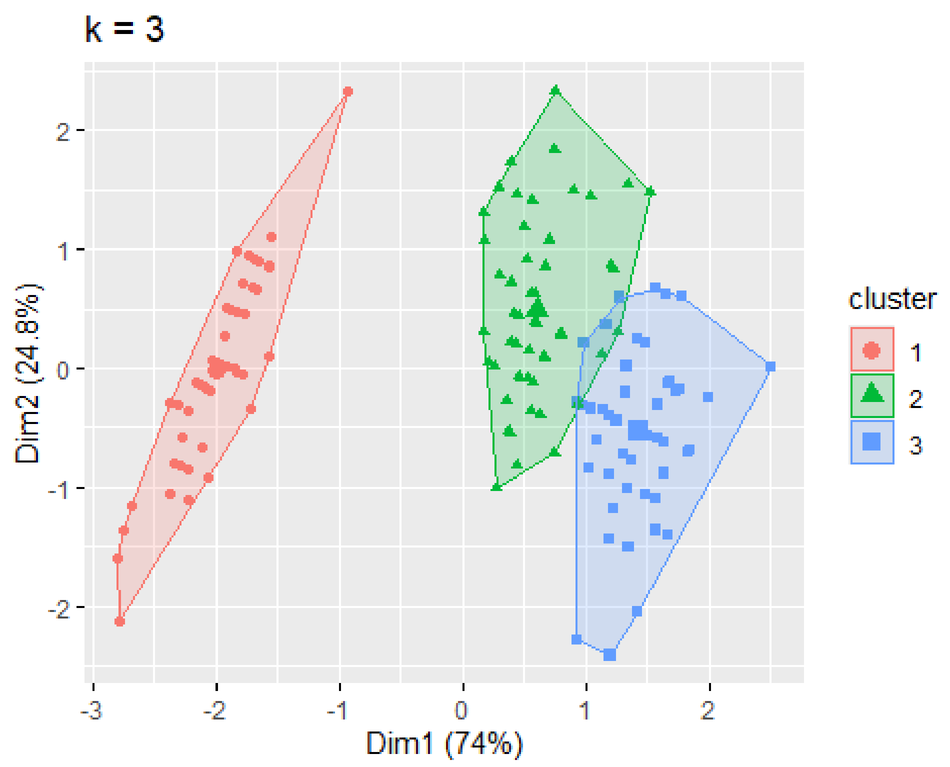
```
#manually choose best predictors
bestpredictors <- c("Sepal.Width", "Petal.Length", "Petal.Width")
irisFiltered <- irisData[, bestpredictors]
```

Firstly, I performed kmean clustering on my filtered dataset (which is only the chosen predictors). Then using the factoextra/ggplot library function to visualize the clusters. This is the cluster for my chosen 3-predictor combination. Take note, that there are a few points that are grouped in both cluster 2 and cluster 3. I had also tried it with the aforementioned 2-predictor combination, which showed 3 clusters with no overlapping points. Despite that I still chose the 3-predictor for two reasons. You will see in the coming section, the 3-pred combo yields higher accuracy. But also, when comparing the two combinations. 2-Pred has no overlap while 3-Pred might say a few points can be either this cluster or that cluster. I think that information is important and would rather have that so I know which points were a bit challenging to group and we can take that information into consideration.

Both plots are going to say "hey I clustered everything but some might be wrong." But this one also says "hey I didnt really know which one 'x' points belonged to so they could be either/or. From there, one can decide to exclude those points, or do whatever they see fit.

```
#reproducibility
set.seed(123)
#kmeans  with optimal k
kMeansCluster <- kmeans(irisFiltered, centers = optimalK, nstart = 25)
#create and print plot
plot1 <- fviz_cluster(kMeansCluster, geom = "point", data =
```

```r
irisFiltered) + ggtitle("k = 3")
print(plot1)
```



I printed the information from the structure kMeansCluster and stored the cluster assignments into a vector. These are the clusters that my code assigned each entry. The reason I do this is because I will have to compare these predicted assignments to the true assignments (if you recall, I had exlcluded the 'species' column)

```r
#clustering info
str(kMeansCluster)

## List of 9
##  $ cluster     : int [1:150] 1 1 1 1 1 1 1 1 1 1 ...
##  $ centers     : num [1:3, 1:3] 3.43 2.75 3 1.46 4.28 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:3] "1" "2" "3"
##   .. ..$ : chr [1:3] "Sepal.Width" "Petal.Length" "Petal.Width"
##  $ totss       : num 579
##  $ withinss    : num [1:3] 9.06 18.86 19.94
##  $ tot.withinss: num 47.9
##  $ betweenss   : num 531
##  $ size        : int [1:3] 50 53 47
##  $ iter        : int 2
##  $ ifault      : int 0
##  - attr(*, "class")= chr "kmeans"

#store cluster assignments
predictedClusters <- kMeansCluster$cluster
```

There are a few steps I need to take to find the accuracy of the clustering. First, I created a contingency table which showed how things were clustered. Cluster 1 is made up entirely of species 1 (setosa), Cluster 2 had 48 points of species 2(veriscolor) and 5 points of species 3(virginica). Cluster 3 has 2 points of species 2(veriscolor) and 45 points of species 3 (virginica).

```
#create mapping of clusters to species based on majority
#create contingency table of cluster assignments to species
clusterToSpecies <- table(predictedClusters, irisData$Species)
print(clusterToSpecies)

##
## predictedClusters  1  2  3
##                  1 50  0  0
##                  2  0 48  5
##                  3  0  2 45
```

Next, I have to find the most common species in each cluster (well technically I wrote it a few lines up but I need to code it because the computer doesn't know it). The first line of code applies the which.max function across the rows of the contingincy table and returns index of highest value (which is the most common species for that cluster). Going further, create a vector for the predicted species, then map each cluster assignment to the most common species in that cluster.

Basically, we already know what clusters have the most of what species, but the computer only knows there are 3 clusters. We have to say "hey you made 3 clusters and cluster 1 is most likely setosa, cluster 2 is most likely veriscolor, and cluster 3 is most likely virginica". So now I'm assigning the predicted clusters into their predicted species.

Finally, we create a vector that has the 'species' coloumn which tells us the true/accurate species for a row. We do the math to find how accurate our clustering was. This clustering, specifically with this 3 predictors combination, yields 95% accuracy. The aforementioned 2-pred combo was 93% accurate. This 2% higher accuracy, coupled with my logic I explained before, were the two main reasons I decided on these 3 predictors.

```r
#find most common species in a cluster
clusterSpeciesMapping <- apply(clusterToSpecies, 1, which.max)
#map predicted clusters to species
predictedSpecies <- clusterSpeciesMapping[predictedClusters]
#calculate accuracy
trueSpecies <- irisData$Species
accuracy <- sum(predictedSpecies == trueSpecies) / length(trueSpecies)
#print accuracy
print(paste0(accuracy, " or ", round(accuracy * 100), "%"))
```

```
## [1] "0.953333333333333 or 95%"
```