Clustering and Cross Validation

ISYE 6501

Week 2 Assignment

About

The purpose of this assignment is to apply clustering and cross validation techniques to sample data sets.

About the Dataset

We are presented with an input data on credit card application information that includes 10 attributes and one response variable. The response variable is Boolean value of 1 and 0 indicating if the credit card application was approved or declined.

Using basic R functions such as head(), summary(), I infer that some attributes are quantitative while others are categorical, binary to be specific; some are integers (discrete values) while others are continuous.

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

Q 3.1a Cross Validation

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)

Background and Strategy

If we used the entire dataset for the training, we would end up with a model that is optimized for the specific dataset. Any real and random effects within the dataset is factored into the model and for new data point coming in, our model may perform poorly. In otherwards, we inadvertently built a model specific to one dataset and not **generalized**.

To avoid this, we split the data into sets called training (Eg: 70%), validate (15%) and test (15%) dataset. We run our models on the training dataset to find the neighbors, use validate data to find best k value and finally use test data to evaluate the effectiveness of the model.

This, however, reserves some data for testing which could otherwise be used for training the model. As you can see from above, only 70% of data is available for training now. To avoid this, we split the dataset into two: training (say 80%) and test (say 20%). Further, we split training dataset into N sets (say 4 sets) and run the model in different combinations of these sets. Basically, we pick 3 out of the 4 sets for training and run validation on the remining set. This will result in 4 accuracy measures for each k value. For a given k, the accuracy is taken to be mean of these 4 measurements.

Testing with KKNN

To test this out, I split the data into training and test dataset and further split the training dataset into 4 sets. Extending on the function I wrote for last week's assignment, I added a wrapper function that will calculate mean accuracy for a given k for all combinations of these training sets.

Prepare the data and create required data splits

```
# split data for k-fold cross validation
# data$train with 80% data (data_train)
# and data$test with 20%
split_data_2 <- function(data_) {
  spread <- c(train = .8, test = .2)
  group_fn <- sample(cut(
    seq(nrow(data_)),
    nrow(data_)*cumsum(c(0,spread)),
    labels = names(spread)
  ))
  data_split = split(data_, group_fn)
  data_split
}
```

```
# split training data into 4 sets : set1, set2, set3, set4
# with 25% spread each
split_data_tr <- function(tr_data_) {
  spread_k <- c(set1 = .25, set2 = .25, set3 = .25, set4 = .25)
  group_fn2 <- sample(cut(
    seq(nrow(tr_data_)),
    nrow(tr_data_)*cumsum(c(0,spread_k)),
    labels = names(spread_k)
  ))
  tr_data_split = split(tr_data_, group_fn2)
  tr_data_split
}

data_split <- split_data_2(data)
data_train_sets <- split_data_tr(data_split$train)
```

```
> str(data_train_sets)
List of 4
 $ set1:'data.frame':   130 obs. of  11 variables:
  ..$ V1 : int [1:130] 1 1 0 1 1 1 0 1 1 1 ...
  ..$ V2 : num [1:130] 32.1 33.2 45.8 42.1 42 ...
  ..$ V3 : num [1:130] 4 1.04 10.5 1.04 9.79 ...
  ..$ V4 : num [1:130] 2.5 6.5 5 5 7.96 ...
  ..$ V5 : int [1:130] 1 1 1 1 1 1 1 1 0 1 ...
  ..$ V6 : int [1:130] 1 1 0 0 0 0 1 0 1 1 ...
  ..$ V7 : int [1:130] 0 0 7 6 8 15 0 3 0 0 ...
  ..$ V8 : int [1:130] 0 0 0 0 1 0 0 1 1 0 ...
  ..$ V9 : int [1:130] 360 164 0 500 0 0 0 311 100 239 ...
  ..$ V10: int [1:130] 0 31285 0 10000 0 5000 4000 300 0 200 ...
  ..$ V11: int [1:130] 1 1 1 1 1 1 1 1 1 1 ...
 $ set2:'data.frame':   131 obs. of  11 variables:
  ..$ V1 : int [1:131] 1 0 1 1 0 0 0 1 0 1 ...
  ..$ V2 : num [1:131] 20.2 22.9 54.4 29.9 19.2 ...
  ..$ V3 : num [1:131] 5.62 11.59 0.5 1.83 8.59 ...
  ..$ V4 : num [1:131] 1.71 0.04 3.96 4.33 0.75 ...
  ..$ V5 : int [1:131] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ V6 : int [1:131] 1 1 1 1 0 0 0 0 0 0 ...
  ..$ V7 : int [1:131] 0 0 0 0 7 2 9 5 5 11 ...
  ..$ V8 : int [1:131] 1 1 1 1 1 1 0 0 0 0 ...
  ..$ V9 : int [1:131] 120 80 180 260 96 100 0 168 0 434 ...
  ..$ V10: int [1:131] 0 1349 314 200 0 0 0 0 560 35 ...
  ..$ V11: int [1:131] 1 1 1 1 1 1 1 1 1 1 ...
 $ set3:'data.frame':   131 obs. of  11 variables:
  ..$ V1 : int [1:131] 0 1 1 0 1 1 1 0 0 1 ...
  ..$ V2 : num [1:131] 24.5 21.8 34.2 25.8 26 ...
  ..$ V3 : num [1:131] 0.5 0.25 9.17 0.5 1 2.04 0.5 0.835 1 0.375
...
  ..$ V4 : num [1:131] 1.5 0.665 4.5 0.875 1.75 ...
  ..$ V5 : int [1:131] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ V6 : int [1:131] 1 1 0 1 1 1 1 0 0 0 ...
  ..$ V7 : int [1:131] 0 0 12 0 0 0 0 1 11 3 ...
  ..$ V8 : int [1:131] 1 0 0 0 0 0 0 1 1 1 ...
  ..$ V9 : int [1:131] 280 0 0 491 280 400 320 0 0 260 ...
  ..$ V10: int [1:131] 824 0 221 0 0 5800 0 0 456 15108 ...
  ..$ V11: int [1:131] 1 1 1 1 1 1 1 1 1 1 ...
```

```
 $ set4:'data.frame':   131 obs. of  11 variables:
  ..$ V1 : int [1:131] 1 1 1 0 0 1 1 1 1 1 ...
  ..$ V2 : num [1:131] 42.5 22.1 48.1 23.2 41.2 ...
  ..$ V3 : num [1:131] 4.92 0.83 6.04 5.88 6.5 ...
  ..$ V4 : num [1:131] 3.17 2.17 0.04 3.17 0.5 ...
  ..$ V5 : int [1:131] 1 0 0 1 1 1 1 1 1 1 ...
  ..$ V6 : int [1:131] 1 1 1 0 0 0 0 0 0 1 ...
  ..$ V7 : int [1:131] 0 0 0 10 3 3 11 11 40 0 ...
  ..$ V8 : int [1:131] 0 0 1 1 0 1 0 1 1 0 ...
  ..$ V9 : int [1:131] 52 128 0 120 145 0 30 0 0 0 ...
  ..$ V10: int [1:131] 1442 0 2690 245 0 0 300 2283 15 0 ...
  ..$ V11: int [1:131] 1 1 1 1 1 1 1 1 1 1 ...
```

The following function runs KNN model for a given k, validates it against the training dataset and gives the accuracy measure. (# predicted values that are same as actual value)

```
#-----------------------------------------------------------------
# Parameterized knn model wrapper - v2
# Run KNN model for given k on given training data,
# use test dataset for accuracy prediction and return computed
accuracy
# Arguments:
#   neighbours_: number of neighbors (k)
#   train_: training data set
#   test_ : test data set
#
knn_model_3 <-
  function(kmax_,
           train_,
           test_,
           distance_ = 2,
           kernel_ = 'optimal') {
    cat ('\nknn_model_3: neighbours: ', kmax_,
         'distance:', distance_,
         'kernel:', kernel_)

    model <- train.kknn(
      V11 ~ .,
      data = train_,
      kmax = kmax_,
      distance = distance_,
      kernel = kernel_,
      scale = TRUE
    )

    a <- 0
    for (i in seq(1, kmax_)) {
      a <- a +
        sum(round(model$fitted.values[[i]]) == data_split$train[,
11]) / nrow(data_split$train)
    }
    cat ('\n... average accuracy on training data: ', a / kmax_)


    fit <- predict(model, test_[, -11])
```

```
    fitted <- table(test_[, 11], round(fit))
    accuracy <- (sum(diag(fitted))) / sum(fitted)
    cat ('\n... average accuracy on test data: ', accuracy)
    accuracy
  }
```

The following function runs KNN model for different combinations of sets and returns the average accuracy for a given k.

```
#-----------------------------------------------------------------
# do k-fold cross validation for spefic k
# Run KNN model for different combiations of sets
# return mean of accuracies for all runs
# Arguments:
#   data_ : data with multiple sets (see split function above)
#   Neighbors_: number of neighbors
#   nsets: number of sets training data is split
kfold_cross_validate <-
  function(data_,
           neighbors_ = 10,
           nsets_ = 4,
           kernel_ = 'optimal') {
    r <- knn_model_3(neighbors_,
                     rbind(data_$set1, data_$set2, data_$set3),
                     data_$set4,
                     kernel_)
    r <- r + knn_model_3(neighbors_,
                         rbind(data_$set2, data_$set3, data_$set4),
                         data_$set1,
                         kernel_)
    r <- r + knn_model_3(neighbors_,
                         rbind(data_$set3, data_$set4, data_$set1),
                         data_$set2,
                         kernel_)
    r <- r + knn_model_3(neighbors_,
                         rbind(data_$set4, data_$set1, data_$set2),
                         data_$set3,
                         kernel_)
    accuracy <- r / nsets_
    accuracy
  }
```

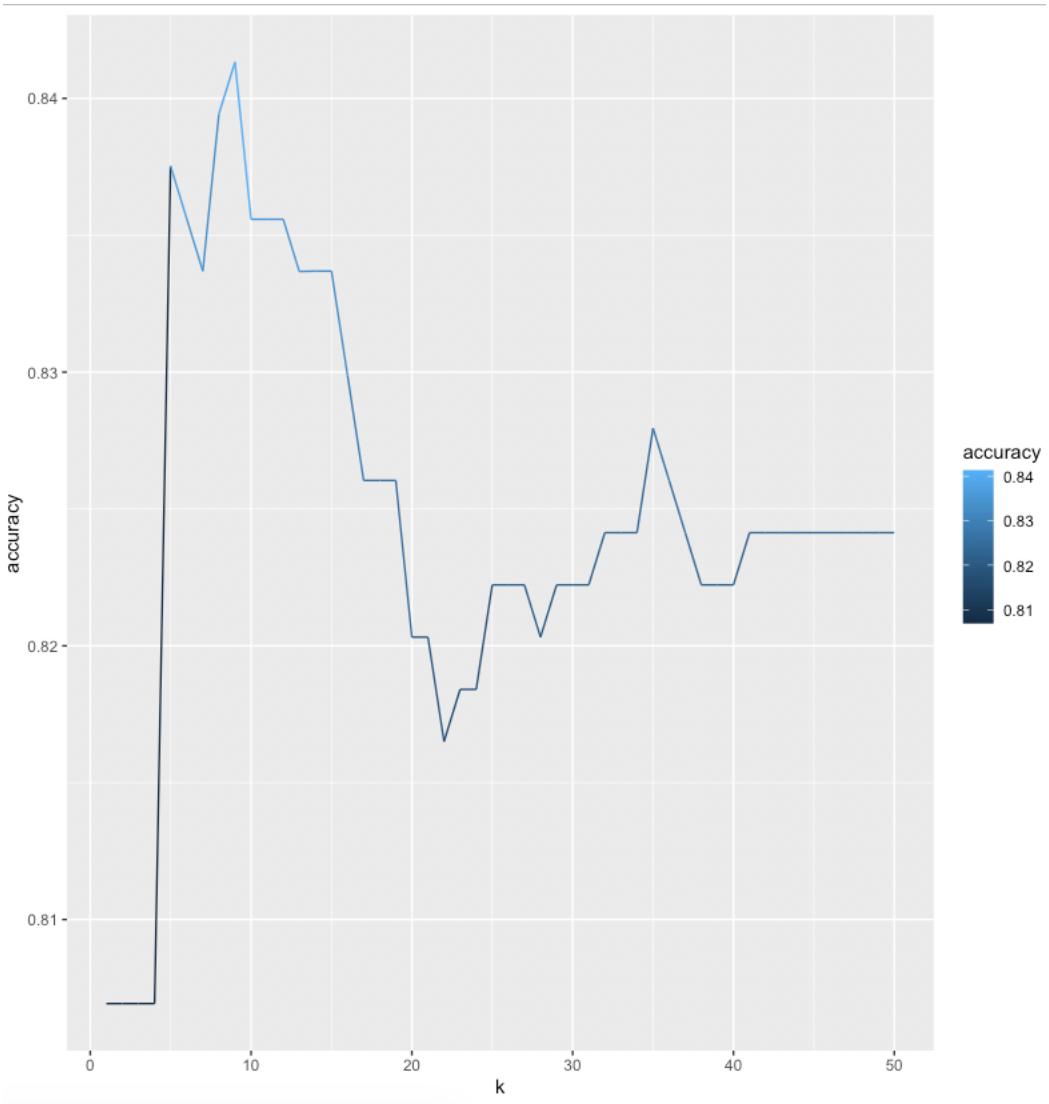
The following runs the model for different k values and stores results in a data frame. The idea is to use the results data frame for further analysis.

```
# run kfold cross validation for different k values and
# store results in a dataframe
results <- data.frame()
max_n <- 50 # max num neighbors to try
for (k in seq(1, max_n)) {
  r <- kfold_cross_validate(data_train_sets, k)
  results <- rbind(results, r)
```

```
}
```

Looking at the results, we can see we get best results for this specific spread of data at k = 9.

```
> ggplot(results, aes(k, accuracy, color=accuracy)) + geom_line()
> n <- results[results$accuracy == max(results$accuracy),]
  k  accuracy
n
9 9 0.8413241
```



Now we can use our test data to evaluate the effectiveness of our model. The test data is supposed to be “unseen” and a good representation of any new input data.

```
> knn_model_3(n$k, data_split$train, data_split$test)
```

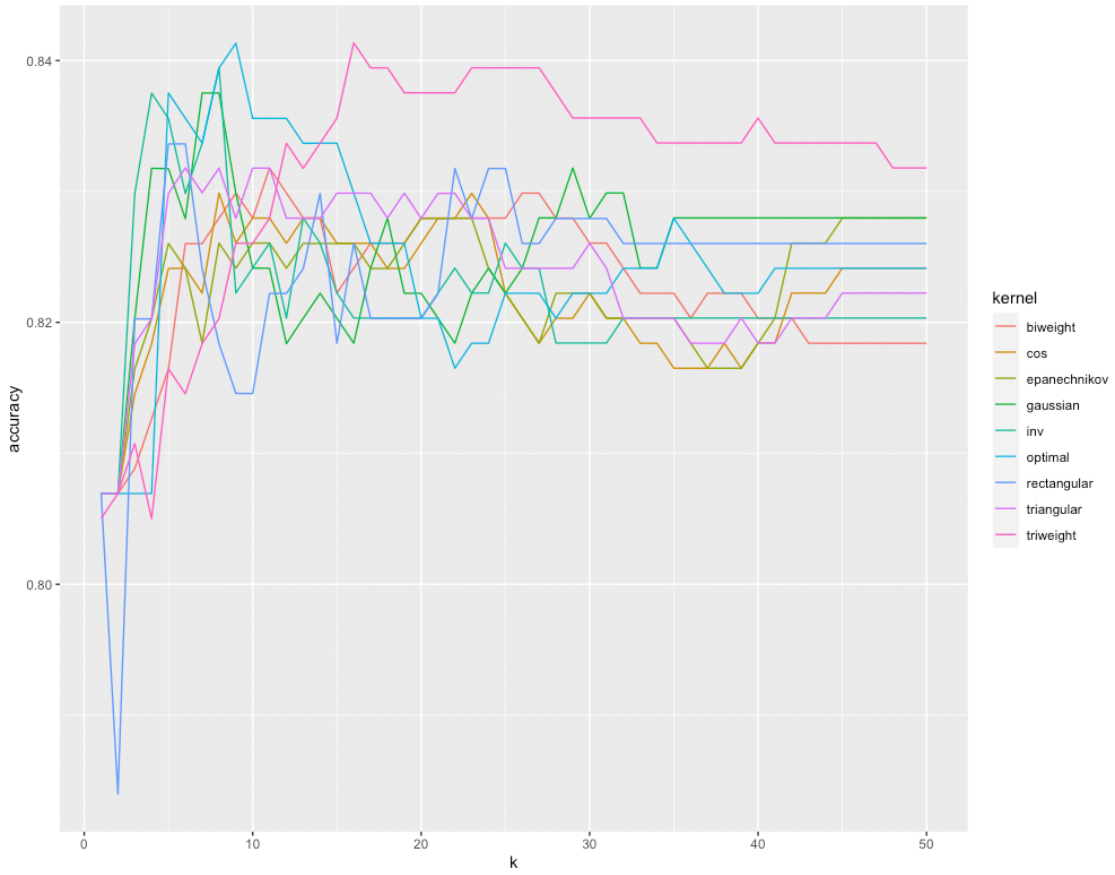
```
[1] 0.8396947
```

Testing with other kernels

When attempting with other kernels, I see a variation in the accuracy results with “triweight” kernel giving a better accuracy, but for larger k values.

```
results <- data.frame()
kmax <- 50 # max num neighbors to try
nsets = 4
for (kernel in c(
  "rectangular",
  "triangular",
  "epanechnikov",
  "biweight",
  "triweight",
  "cos",
  "inv",
  "gaussian",
  "optimal"
)) {
  cat ('\nTrying kernel:', kernel, 'nsets:', nsets, 'kmax:', kmax)
  for (k in seq(1, kmax)) {
    accuracy <- kfold_cross_validate(data_train_sets, k, nsets,
kernel)
    results <-
      rbind(results, data.frame(kernel, k, nsets, accuracy))
  }
}
```

```
ggplot(results, aes(k, accuracy, color = kernel)) + geom_line()
```



Testing with KSVM

Using similar strategy, we could also examine how SVM model fares when we train and test with different data sets.

```
#-----------------------------------------------------
# Parameterized SVM Model
# cost_ : C value
# kernel_ : svm kernel
# type_: svm type
# train_ : training data
# test_ : test data
svm_model_3 <- function(cost_, train_, test_, kernel_ =
"vanilladot", type_ = "C-svc") {
  cat ('\nCalculating for Kernel:', kernel_, 'Type:', type_, 'Cost
:', cost_)
  t0 <- proc.time() # track how long it takes
  model <- ksvm(as.matrix(train_[,1:10]),
                as.factor(train_[,11]),
                type=type_,
                kernel=kernel_,
                C=cost_,
                scaled=TRUE)
  a <- colSums(model@xmatrix[[1]] * model@coef[[1]])
  a0 <- -model@b
  pred <- predict(model,test_[,1:10])
```

```
  accuracy <- sum(pred == test_[,11]) / nrow(test_)
  accuracy
}
```

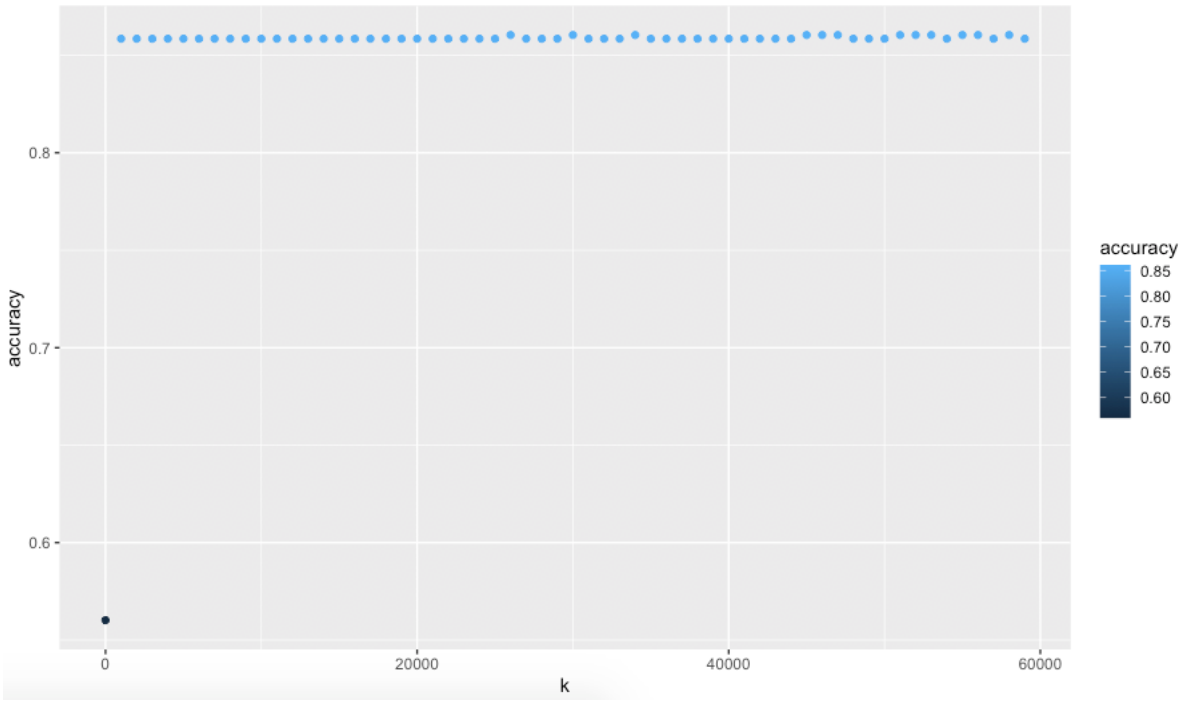
Wrapper to call this over training sets combination and averaging over the values.

```
#------------------------------------------------------------
# test with svm model
#
kfold_cross_validate_with_svm <- function(data_, cost_, nsets_=4) {
  r <- svm_model_3 (cost_,
                    rbind(data_$set1, data_$set2, data_$set3),
                    data_$set4)
  r <- r + svm_model_3(cost_,
                       rbind(data_$set2, data_$set3, data_$set4),
                       data_$set1)
  r <- r + svm_model_3(cost_,
                       rbind(data_$set3, data_$set4, data_$set1),
                       data_$set2)
  r <- r + svm_model_3(cost_,
                       rbind(data_$set4, data_$set1, data_$set2),
                       data_$set3)
  accuracy <- r/nsets_
  data.frame(k, accuracy)
}
```

Finally call it with a wide range for cost values. As before, store response in a data frame for further plotting and analysis.

```
results <- data.frame()
N <- 10
for (k in seq(1/10.^N, 10.^N, by=1000)) {
  r <- kfold_cross_validate_with_svm (data_train_sets, k)
  results <- rbind(results, r)
}
```

As explained by TA's in office hours, varying the cost by small value in SVM model doesn't really alter the average accuracy significantly. In the interest of saving my computer fans, I had to stop the run after the loop ran for more than 30 mins. I saw some variability in the accuracy, but not a lot.



But what's interesting is that in average, SVM models seem to yield better overall accuracy compared to KNN.

```
> results[results$accuracy == max(results$accuracy),][1,]
       k  accuracy
27 26000 0.8604228
```

Exploratory Analysis

Training with and without Test Data

If we measure the accuracy with same data set that we used for training, **we get a slightly better results than using test data**. This is compatible with our understanding – ie. The model is optimized for the dataset and performs better, but not general enough to handle new data.

```
for (i in seq(1,N)) {
  accuracy<-
sum(round(model$fitted.values[[i]])==data_split$train[,11]) /
nrow(data_split$train)
  cat (i, '=>', a,'\n')
  out <- rbind(out, data.frame(i,accuracy))
}
> out[out$accuracy == max(out$accuracy),]
    i  accuracy
10 10 0.8604207
11 11 0.8604207
```

Root Mean Squared Error

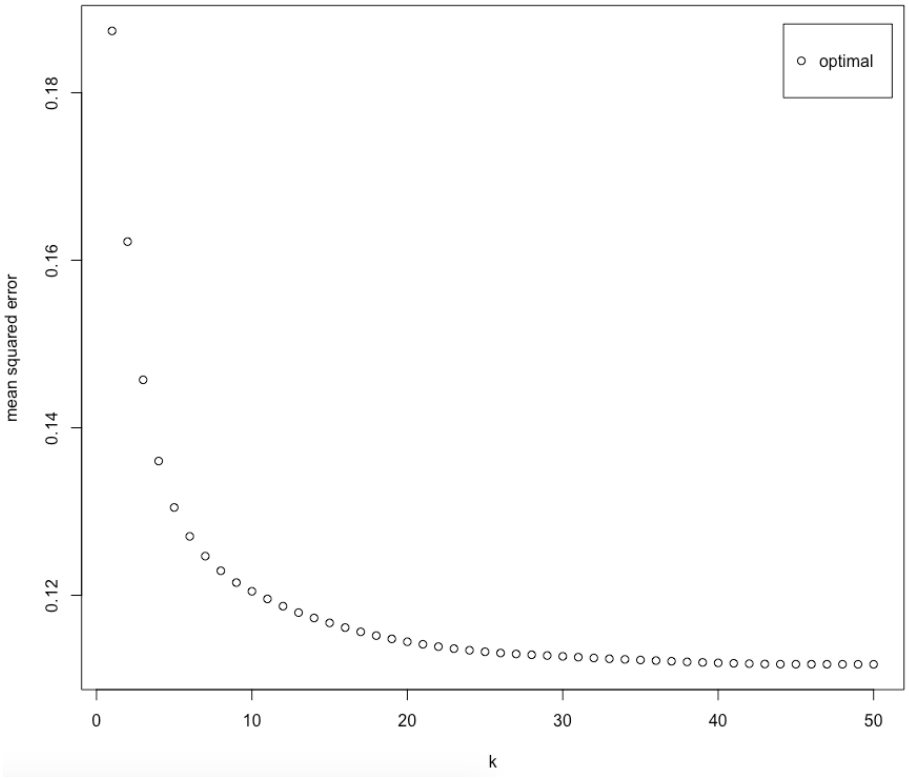
Looking further into the model, when we plot the **mean squared errors** for various k values, we see it **decreasing RMSE as we increased number of neighbors**. The best model is the one that minimizes the RMSE. This is compatible with the output from printing model variable.

```
> model <- train.kknn(V11 ~ .,
                      data = data_split$train,
                      kmax = 50,
                      scale = TRUE)
> plot(model)

> model

Call:
train.kknn(formula = V11 ~ ., data = data_split$train, kmax = 50,
scale = TRUE)

Type of response variable: continuous
minimal mean absolute error: 0.2008066
Minimal mean squared error: 0.105889
Best kernel: optimal
Best k: 50
```

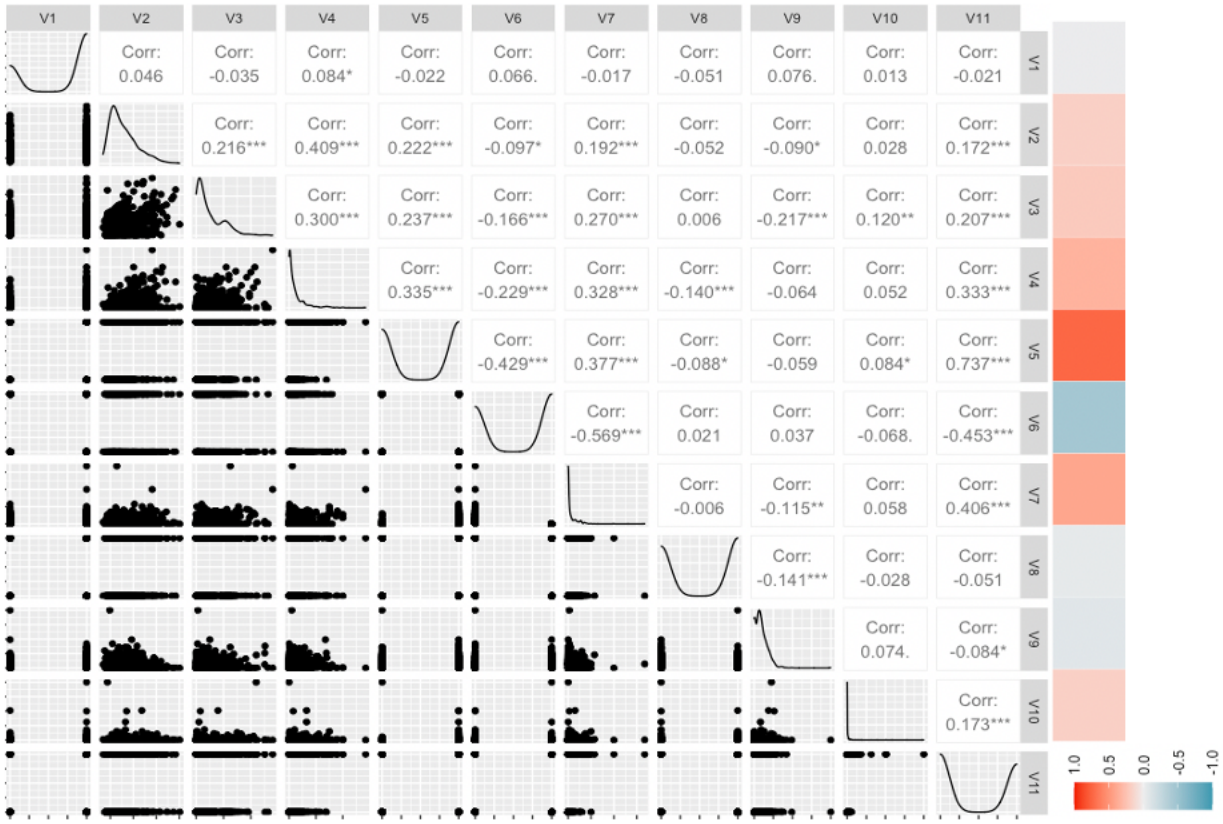


Effect of large k

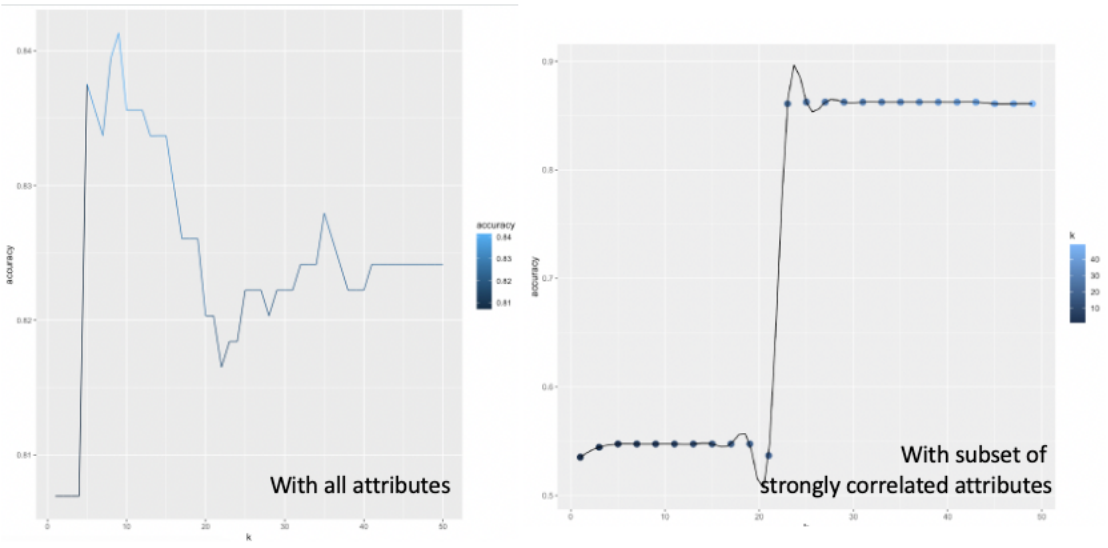
One interesting observation on cross validation results is that adding more neighbors to our model didn't help improve the accuracy. I had observed the same in the previous week assignment as well. If I drop weakly correlated attributes and compute and calculate only against strongly correlated attributes, I see better outcome.

This makes me believe that **presence of large number of attributes influence the model and picking the right attributes for model is as important** as picking other model parameters such as k and kernel.

Influence of each attribute to the response variable is plotted below.



Accuracy for all attributes vs only strongly correlated attributes (V5, V6, V7) is plotted below.



Comparing Accuracy Across Models

Model	Accuracy	Comments
KKNN with Test Data	0.8396947	Model runs fast, but needs more tuning at evaluation time
KKNN without Test Data	0.8604207	Not recommended
KSVM model	0.8604228	Model runs very slow, but doesn't require much tuning at evaluation time

Q 3.1b Splitting data

(b) splitting the data into training, validation, and test data sets (pick either KNN or SVM; the other is optional).

KKNN

Using slight modification to split above, we now split the data into 3 sets

- train with 60% data
- test with 20%
- validate with remaining 20%

```
# split data into train, test, validate with 60%, 20%, 20% spread
split_data_3 <- function(data_) {
  spread <- c(train = .6, test = .2, validate = .2)
  group_fn <- sample(cut(
    seq(nrow(data_)),
    nrow(data)*cumsum(c(0,spread)),
    labels = names(spread)
  ))
  data_split = split(data_, group_fn)
  data_split
}

data_split <- split_data_3(data)
```

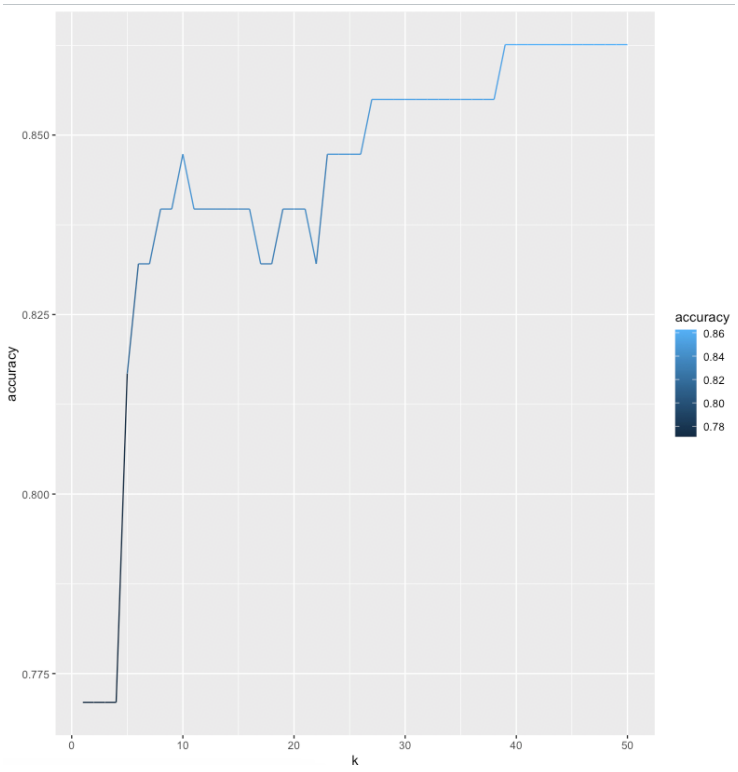
```
> str(data_split)
List of 3
 $ train   :'data.frame':     392 obs. of  11 variables:
  ..$ V1 : int [1:392] 0 0 1 0 1 1 0 1 1 0 ...
  ..$ V2 : num [1:392] 58.7 24.5 27.8 22.9 29.9 ...
  ..$ V3 : num [1:392] 4.46 0.5 1.54 11.59 1.83 ...
  ..$ V4 : num [1:392] 3.04 1.5 3.75 0.04 4.33 ...
  ..$ V5 : int [1:392] 1 1 1 1 1 0 1 1 1 1 ...
  ..$ V6 : int [1:392] 0 1 0 1 1 1 0 0 1 0 ...
  ..$ V7 : int [1:392] 6 0 5 0 0 0 7 17 0 1 ...
  ..$ V8 : int [1:392] 1 1 0 1 1 1 0 1 1 1 ...
  ..$ V9 : int [1:392] 43 280 100 80 260 0 0 200 300 120 ...
  ..$ V10: int [1:392] 560 824 3 1349 200 2690 0 1208 0 11 ...
  ..$ V11: int [1:392] 1 1 1 1 1 1 1 1 1 1 ...
 $ test    :'data.frame':     131 obs. of  11 variables:
  ..$ V1 : int [1:131] 1 1 1 1 1 1 0 0 1 1 ...
  ..$ V2 : num [1:131] 20.2 32.1 33.2 22.1 36.7 ...
  ..$ V3 : num [1:131] 5.62 4 1.04 0.83 4.42 ...
  ..$ V4 : num [1:131] 1.71 2.5 6.5 2.17 0.25 ...
  ..$ V5 : int [1:131] 1 1 1 0 1 1 1 1 1 1 ...
  ..$ V6 : int [1:131] 1 1 1 1 0 0 0 0 0 0 ...
  ..$ V7 : int [1:131] 0 0 0 0 10 3 10 3 17 3 ...
  ..$ V8 : int [1:131] 1 0 0 0 0 0 1 0 0 1 ...
  ..$ V9 : int [1:131] 120 360 164 128 320 396 120 145 0 0 ...
  ..$ V10: int [1:131] 0 0 31285 0 0 0 245 0 0 0 ...
  ..$ V11: int [1:131] 1 1 1 1 1 1 1 1 1 1 ...
 $ validate:'data.frame':     131 obs. of  11 variables:
```

```
  ..$ V1 : int [1:131] 1 1 1 0 1 0 0 0 1 1 ...
  ..$ V2 : num [1:131] 30.8 54.4 42.5 38.2 21.8 ...
  ..$ V3 : num [1:131] 0 0.5 4.92 6 0.25 ...
  ..$ V4 : num [1:131] 1.25 3.96 3.165 1 0.665 ...
  ..$ V5 : int [1:131] 1 1 1 1 1 1 1 1 1 ...
  ..$ V6 : int [1:131] 0 1 1 1 1 0 0 0 0 0 ...
  ..$ V7 : int [1:131] 1 0 0 0 0 7 6 9 5 2 ...
  ..$ V8 : int [1:131] 1 1 0 0 0 1 0 0 0 1 ...
  ..$ V9 : int [1:131] 202 180 52 0 0 96 0 0 168 260 ...
  ..$ V10: int [1:131] 0 314 1442 0 0 0 1260 0 0 500 ...
  ..$ V11: int [1:131] 1 1 1 1 1 1 1 1 1 1 ...
```

We will use the same parameterized KNN model wrapper to use the training data for learning and validate data for verifying our model. As before, we will collect the results in a data frame.

```
results2 <- data.frame()
max_n <- 50 # max num neighbors to try
for (k in seq(1, max_n)) {
  r <- knn_model_3(k, data_split$train, data_split$validate)
  results2 <- rbind(results2, data.frame(k, r))
}
```

We see this time the best accuracy is arrived at 39 neighbors. Running this against the test data, we get 0.86% accuracy measure from our model.



```
> n<-results2[results2$accuracy == max(results2$accuracy),][1,]
> n
    k  accuracy
39 39 0.8625954
> knn_model_3(n$k, data_split$train, data_split$test)
[1] 0.8625954
```

Conclusion

Based on the results and runtimes, I conclude that the cross validation taking much longer to converge on the results. This is because for each k value, it must run the model k times and the average the accuracy over all runs.

This, however, makes more data available for training and hence a less biased model than running validation on same data set or splitting the data into three buckets for training, test and validate.

Q 4.1 – Clustering example

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.

Iam a music buff listening to all kind of music listening to a wide range of music - from Carnatic music (https://en.wikipedia.org/wiki/Carnatic_music) from my mother land to Jazz and Pop. Naturally, my Spotify playlists are ever expanding reflecting changes in my exposure, accessibility and perhaps age. What used to be a simple 3 or 4 playlists (like: Morning Music, Music for Walking, Bedtime music) now has over 100 playlists.

When I hear a new track / reinvent an old track, I invariably must decide which playlist (category) the track goes in. If there is no fair categorization exists, then a new one gets created.

Attributes in deciding this categorization are:

- Genre of the track
- General mood of the track
- Length of the track
- Type of the track (with *apologies* for using unfamiliar terms – South Indian classical music system has different types of tracks such as Varnam, Krithi, Tanam, Virutham, Thillana, etc.)
- Beat (again going back to South Indian classical music – beats with 3 or 6, 4 or 8, 5, 7, and 9 measure have distinct rhythms)
- Artist / Band
- Vocal / Instrument heavy

Iam sure Spotify itself uses some unsupervised learning algorithms to both categorize its users and they music they listen to (https://www.analyticssteps.com/blogs/how-spotify-uses-machine-learning-models) . But this is another, personalized layer on top of that - currently a manual and intuitive process with a click and drop.

One difference between this example and the strict clustering algorithm we have seen in this course so far is that sometimes, a track gets added to more than one playlist (**multi or overlapping clustering**).

Q 4.3 Clustering IRIS dataset

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.

About Dataset

The famous IRIS dataset has few attributes of the flower such as the sepal length and width and petal length and the width. All these are continuous variables. The response variable is the Species – or the type of flower.

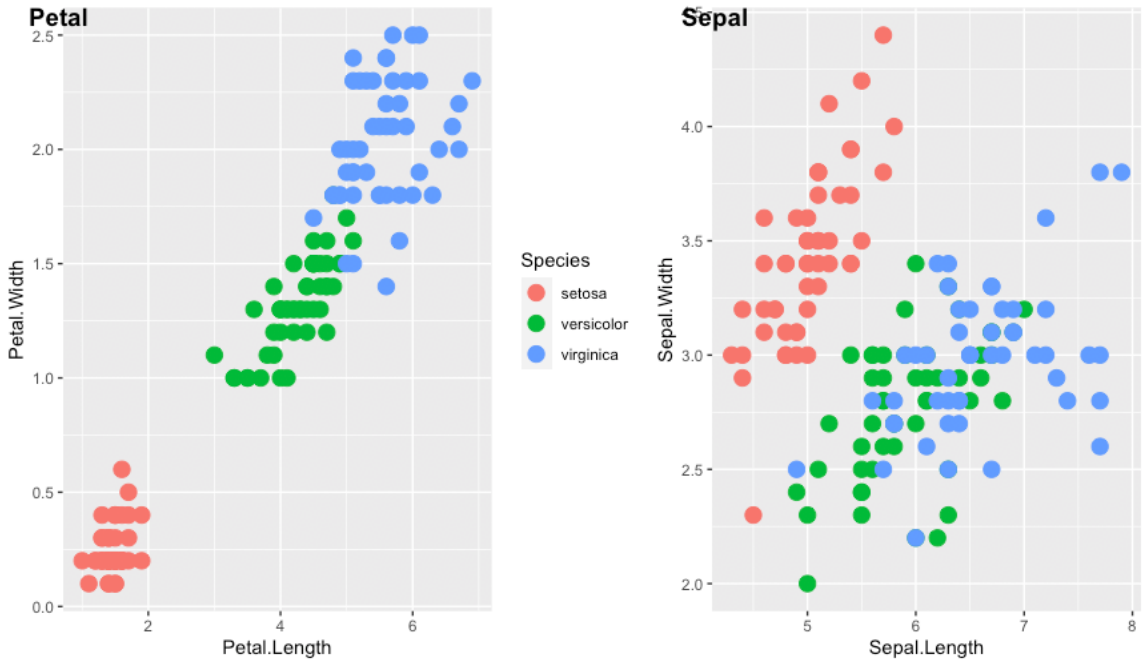
```
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
Species
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
versicolor:50
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
virginica :50
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

Though we are asked to use the dataset for clustering (unsupervised learning), since we have the response variable already, let's just take a quick look to see what kind of relationship exists.

Attribute relationship

We can readily see that Setosa has distinct attributes where as Virginica and Versicolor have somewhat similar attributes (esp. looking only at Sepal properties).

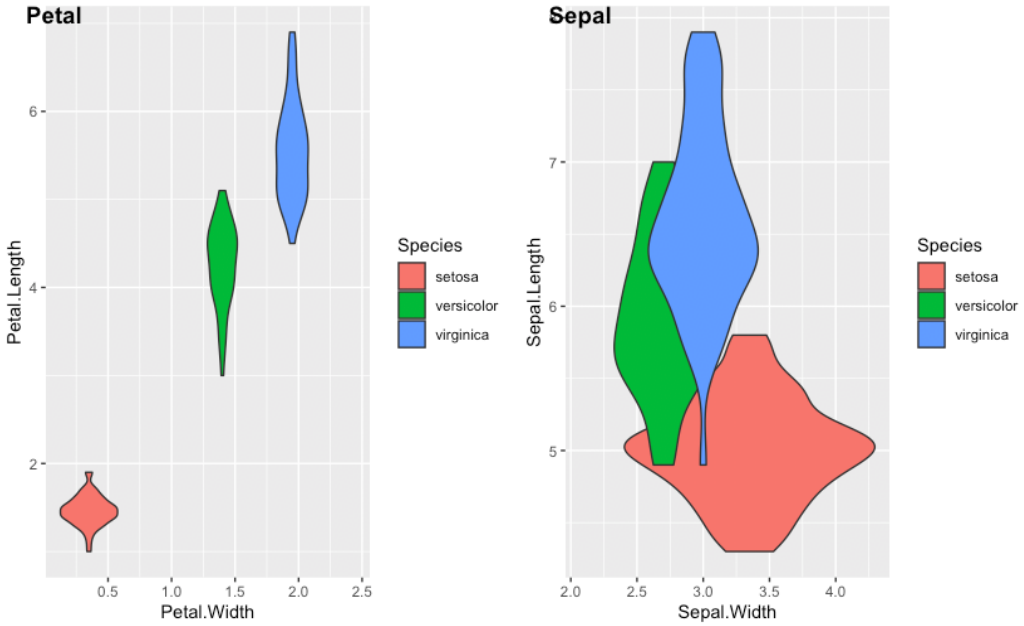
```
p1 <- ggplot(data, aes(Petal.Length, Petal.Width)) +
geom_point(aes(col=Species), size=4)
p2 <- ggplot(data, aes(Sepal.Length, Sepal.Width)) +
geom_point(aes(col=Species), size=4)
ggarrange(p1, p2,
          labels = c("Petal", "Sepal"),
          ncol = 2, nrow = 1)
```



Attribute Values

Looking at the summary above, we observer that the Sepal and Petal properties all have similar range of values. Relatively speaking, Petal width (range: 0.1 – 2.5) has narrow distribution compared to other properties. Petal Length seem to have greater distribution (1 – 6.9). It will be clear to see this visually.

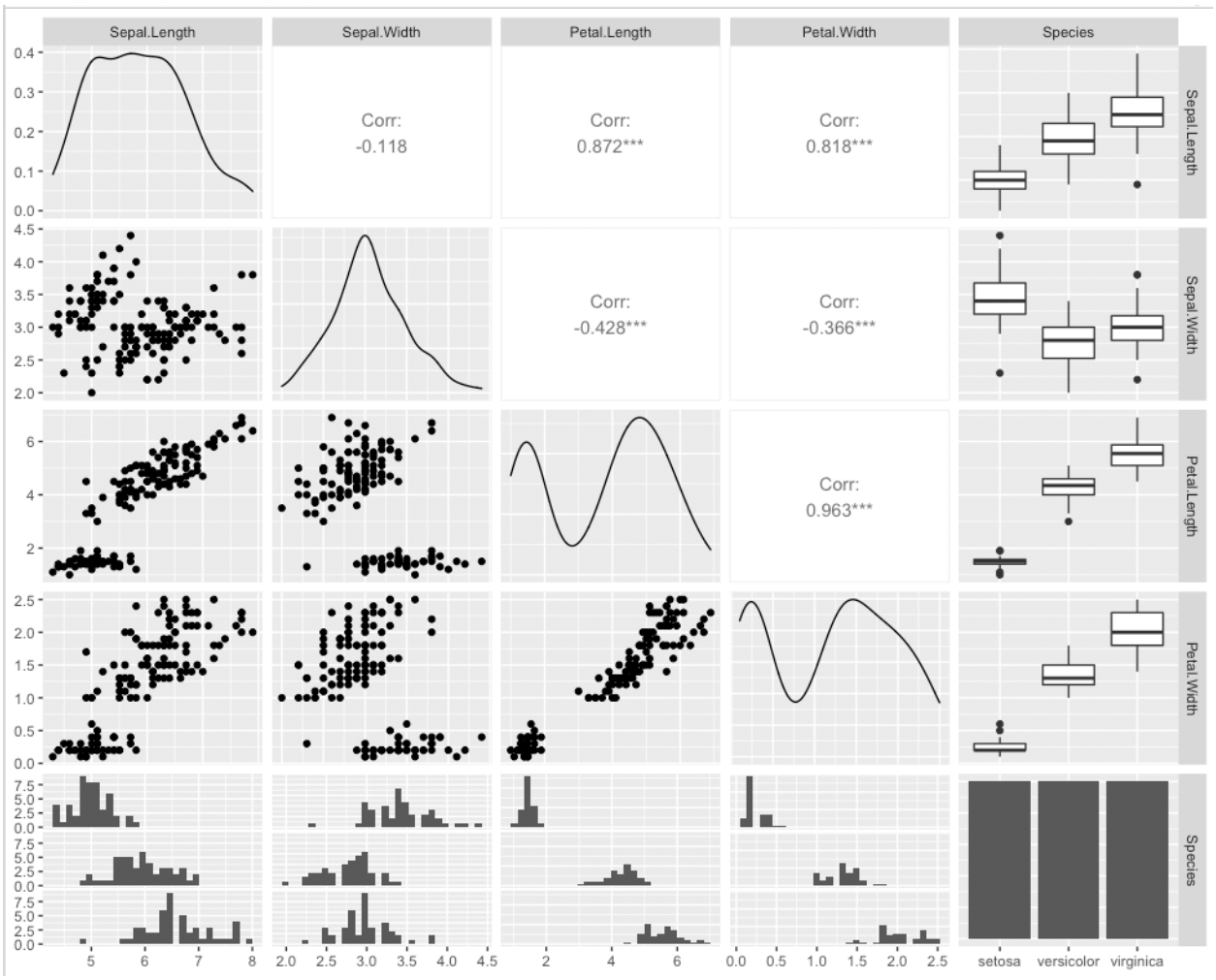
```
# Plot distribution of attributes across Species
p1 <- data %>%
  ggplot(aes(x=Petal.Width,
             y=Petal.Length,
             fill=Species)) +
  geom_violin()
p2<- data %>%
  ggplot(aes(x=Sepal.Width,
             y=Sepal.Length,
             fill=Species))+
  geom_violin()
ggarrange(p1, p2,
          labels = c("Petal", "Sepal"),
          ncol = 2, nrow = 1)
```



Attribute correlation

All this information is also visually summarized below.

```
ggpairs(data)
```



From the above data and plots, we can conclude that all four attributes have influence on the species, with varying degree of influence.

K Means

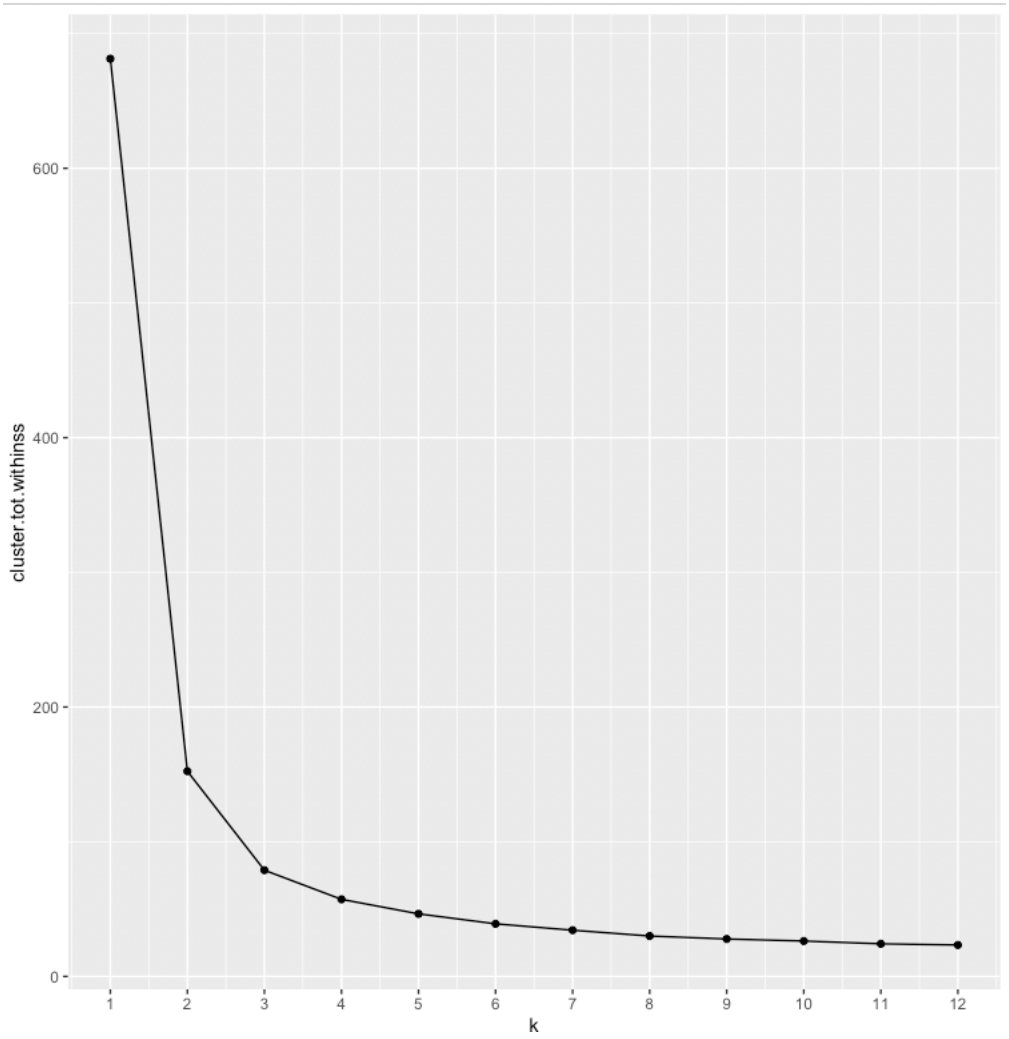
Using kmeans() function, lets examine the clustering results for different number of clusters (from 1 to 12 in the following code snippet). As before, lets store the values in a data frame for further analysis. In particular, we are interested in looking at **tot.withinss (within cluster sum of squares) .** Since this is a measure of total intra-cluster variance, our objective is to pick number of cluster that minimizes this.

```
nclusters_ <- 12 # max no. clusters
nstart <- 25 # test upto 25 initial configuration
results = data.frame()
for (k in seq(1:nclusters_)) {
  cluster <- kmeans(data[,1:4], center=k, nstart=nstart_)
  results <- rbind (results, data.frame(k, cluster$tot.withinss))
```

```
}
```

By plotting the elbow plot of the withinss for various number of clusters, we can identify **3 to be the optimal number of clusters** since it minimizes the intra cluster variance.

```
ggplot(results, aes(x = k, y = cluster.tot.withinss)) +
  geom_line() + geom_point()+
  scale_x_continuous(breaks = 1:nclusters_)
```



Verification

We can rerun the kmeans with 3 centers and check how well it classified the data set. To compare the number cluster index the clusters variable has with the string Species in the dataset, we need to first transform them to a common format. Below, using data-frame spmap, the index is mapped to string and compared.

```
clusters <- kmeans(data[,1:4], center=3, nstart=10)
data$Species_predicted_ind <- clusters$cluster

# map index to species
spmap <- c('versicolor', 'virginica', 'setosa')
names(spmap) <- c(1,2,3)
data$Species_predicted <- spmap[data$Species_predicted_ind]
accuracy <- sum(data$Species == data$Species_predicted) / nrow(data)
```

We find the accuracy of the model to be quite high – close to 90%.

```
> accuracy
[1] 0.8933333
```

References

1. Course Videos and Office hours (of course!). Some seed code lifted from office hour discussion.
2. https://www.statlearning.com/
3. http://rstudio-pubs-static.s3.amazonaws.com/24844_335efcfc09954ad99c4e05d9548ed2ad.html
4. https://levelup.gitconnected.com/knn-k-nearest-neighbours-d3ce76380e14
5. https://web.archive.org/web/20200121091131/http://www.statsoft.com/Textbook/k-Nearest-Neighbors
6. https://www.youtube.com/channel/UCtYLUTtgS3k1Fg4y5tAhLbw