Exam Review

Team 10: Matthew Sadosuk, Akram Bijapuri, Justin Pender, Alejandro Valencia Patino

4/12/2021

Chapter 9, Applied exercise 8

For this exercise in chapter 9 we go back and cover support vector classifiers. At the end of the chapter we selected applied exercise 8. This exercise features the OJ data set, where we will be seeing how purchase price of OJ is affected by the 17 other variables in the dataset

To start this problem we need to load in the two libraries: ISLR and e1071, which will give us access to the OJ data set. To keep the results consistent we will set the seed. Now we will create a random sample with 800 observations from the OJ data set and then we will split the data into a test set and training set

```r
rm(list = ls())

# 8. This problem involves the OJ data set which is part of the ISLR package
library(ISLR)
library(e1071)
```

```
## Warning: package 'e1071' was built under R version 4.0.3
```

```r
#(a) Create a training set containing a random sample of 800
# observations, and a test set containing the remaining observations.
set.seed(5208)

train <- sample(nrow(OJ), 800)
OJ_train <- OJ[train, ]
OJ_test <- OJ[-train, ]
```

After initializing all of the parameters we can now start to construct the support vector classifier. In the code below we use the svm function and within the function we are predicting the purchase price onto the entire dataset,we set the kernel to linear, use the training dataset, and set cost = .01. Now we take the summary of svm_linear, and see that there are 439 support vectors that lay along the hyperplane.

```r
#(b) Fit a support vector classifier to the training data using
# cost=0.01, with Purchase as the response and the other variables
# as predictors. Use the summary() function to produce summary
# statistics, and describe the results obtained.

svm_linear <- svm(Purchase ~ . , kernel = "linear", data = OJ_train, cost = 0.01)
summary(svm_linear)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = OJ_train, kernel = "linear", cost = 0.01)
##
```

```
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.01
##
## Number of Support Vectors:  439
##
##  ( 221 218 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

In this step, we created a function to calculate the error rate using the model, dataset, and the object being classified. Inside the function, a confusion matrix is created with the predicted values from the linear model and the OJ_train dataset. The MSE is then calculated and returned using the values from the confusion matrix.

```
#(c) What are the training and test error rates?

# calculate error rate
calc_error_rate <- function(svm_model, dataset, true_classes) {
  confusion_matrix <- table(predict(svm_model, dataset), true_classes)
  return(1 - sum(diag(confusion_matrix)) / sum(confusion_matrix))
}

cat("Training Error Rate:", 100 * calc_error_rate(svm_linear, OJ_train, OJ_train$Purchase), "%\n")
```

```
## Training Error Rate: 16.375 %
```

```
cat("Test Error Rate:", 100 * calc_error_rate(svm_linear, OJ_test, OJ_test$Purchase), "%\n")
```

```
## Test Error Rate: 17.40741 %
```

In this step we will tune the svm model to try to improve the accuracy of the model. In the function we specify the model type = svm, y-variable, data = OJ, kernel = linear, and then display range of costs from .01 to 10. After this we will look at the summary of svm_tune to see what the best performance number was.

```
#(d) Use the tune() function to select an optimal cost. Consider values in the range 0.01 to 10.
set.seed(5208)

svm_tune <- tune(svm, Purchase ~ . , data = OJ_train, kernel = "linear",
                 ranges = list(cost = seq(0.01, 10, length=50)))

summary(svm_tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##       cost
##  0.8255102
```

```
##
## - best performance: 0.17375
##
## - Detailed performance results:
##         cost   error dispersion
## 1  0.0100000 0.17375 0.02729087
## 2  0.2138776 0.17500 0.03486083
## 3  0.4177551 0.17625 0.03356689
## 4  0.6216327 0.17500 0.03486083
## 5  0.8255102 0.17375 0.03356689
## 6  1.0293878 0.17500 0.03818813
## 7  1.2332653 0.17500 0.03818813
## 8  1.4371429 0.17625 0.03747684
## 9  1.6410204 0.17625 0.03747684
## 10 1.8448980 0.17625 0.03747684
## 11 2.0487755 0.17500 0.03584302
## 12 2.2526531 0.17375 0.03557562
## 13 2.4565306 0.17375 0.03557562
## 14 2.6604082 0.17375 0.03557562
## 15 2.8642857 0.17375 0.03557562
## 16 3.0681633 0.17375 0.03557562
## 17 3.2720408 0.17375 0.03557562
## 18 3.4759184 0.17375 0.03557562
## 19 3.6797959 0.17500 0.03535534
## 20 3.8836735 0.17375 0.03653860
## 21 4.0875510 0.17375 0.03653860
## 22 4.2914286 0.17375 0.03653860
## 23 4.4953061 0.17375 0.03653860
## 24 4.6991837 0.17375 0.03653860
## 25 4.9030612 0.17375 0.03653860
## 26 5.1069388 0.17375 0.03653860
## 27 5.3108163 0.17375 0.03653860
## 28 5.5146939 0.17500 0.03726780
## 29 5.7185714 0.17500 0.03726780
## 30 5.9224490 0.17500 0.03726780
## 31 6.1263265 0.17625 0.03928617
## 32 6.3302041 0.17750 0.03525699
## 33 6.5340816 0.17750 0.03525699
## 34 6.7379592 0.17750 0.03525699
## 35 6.9418367 0.17750 0.03525699
## 36 7.1457143 0.17750 0.03525699
## 37 7.3495918 0.17750 0.03525699
## 38 7.5534694 0.17875 0.03537988
## 39 7.7573469 0.17875 0.03537988
## 40 7.9612245 0.17875 0.03537988
## 41 8.1651020 0.17875 0.03537988
## 42 8.3689796 0.17875 0.03537988
## 43 8.5728571 0.17875 0.03537988
## 44 8.7767347 0.17875 0.03537988
## 45 8.9806122 0.17875 0.03537988
## 46 9.1844898 0.17875 0.03537988
## 47 9.3883673 0.18000 0.03343734
## 48 9.5922449 0.17875 0.03537988
## 49 9.7961224 0.17750 0.03525699
```

```
## 50 10.0000000 0.17750 0.03525699
```

```
bestsvm <- svm_tune$best.model
```

In this step we look at the summary of svm_tune and look to see what cost gives the best performance in the model. From looking at the summary list it looked to be a cost of 5.01 gave the best performance. So to make sure this is correct we now use the best.parameters test method to identify the best cost. After this we look at the training and test error rates to see how the tune we did above improved or hurt the model accuracy

```
#(e) Compute the training and test error rates using this new value for cost.


cat("Training Error Rate:", 100 * calc_error_rate(bestsvm, OJ_train, OJ_train$Purchase), "%\n")
```

```
## Training Error Rate: 15.625 %
```

```
cat("Test Error Rate:", 100 * calc_error_rate(bestsvm, OJ_test, OJ_test$Purchase), "%\n")
```

```
## Test Error Rate: 17.03704 %
```

For steps f and g we will follow the same steps that were done in b to e. The only input being changed in the model is the kernel which will be set to radial in f and poly for the second one. At the end I will compare the test and train error rate results and publish which model will be the best

In part c and e we are still using the function that was made above, but the only part we will be changing is the name of the model to the new svm variable.

```
#(f) Repeat parts (b) through (e) using a support vector machine with a radial kernel. Use the default
# part b: fitting the support vector
set.seed(5208)
svm_radial <- svm(Purchase ~ . , data = OJ_train, kernel = "radial")
summary(svm_radial)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = OJ_train, kernel = "radial")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  371
##
##  ( 187 184 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

```
# part c: Calculating the test and error rate
cat("Training Error Rate:", 100 * calc_error_rate(svm_radial, OJ_train, OJ_train$Purchase), "%\n")
```

```
## Training Error Rate: 14.375 %
```

```r
cat("Test Error Rate:", 100 * calc_error_rate(svm_radial, OJ_test, OJ_test$Purchase), "%\n")
```

```
## Test Error Rate: 17.77778 %
```

```r
# part d: Adding the costs from .01 to 10
set.seed(5208)
svm_tune2 <- tune(svm, Purchase ~ . , data = OJ_train, kernel = "radial",
                  ranges = list(cost = seq(0.01, 10), length = 50))

summary(svm_tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##       cost
##  0.8255102
##
## - best performance: 0.17375
##
## - Detailed performance results:
##          cost   error dispersion
## 1   0.0100000 0.17375 0.02729087
## 2   0.2138776 0.17500 0.03486083
## 3   0.4177551 0.17625 0.03356689
## 4   0.6216327 0.17500 0.03486083
## 5   0.8255102 0.17375 0.03356689
## 6   1.0293878 0.17500 0.03818813
## 7   1.2332653 0.17500 0.03818813
## 8   1.4371429 0.17625 0.03747684
## 9   1.6410204 0.17625 0.03747684
## 10  1.8448980 0.17625 0.03747684
## 11  2.0487755 0.17500 0.03584302
## 12  2.2526531 0.17375 0.03557562
## 13  2.4565306 0.17375 0.03557562
## 14  2.6604082 0.17375 0.03557562
## 15  2.8642857 0.17375 0.03557562
## 16  3.0681633 0.17375 0.03557562
## 17  3.2720408 0.17375 0.03557562
## 18  3.4759184 0.17375 0.03557562
## 19  3.6797959 0.17500 0.03535534
## 20  3.8836735 0.17375 0.03653860
## 21  4.0875510 0.17375 0.03653860
## 22  4.2914286 0.17375 0.03653860
## 23  4.4953061 0.17375 0.03653860
## 24  4.6991837 0.17375 0.03653860
## 25  4.9030612 0.17375 0.03653860
## 26  5.1069388 0.17375 0.03653860
## 27  5.3108163 0.17375 0.03653860
## 28  5.5146939 0.17500 0.03726780
## 29  5.7185714 0.17500 0.03726780
## 30  5.9224490 0.17500 0.03726780
```

```
## 31  6.1263265 0.17625 0.03928617
## 32  6.3302041 0.17750 0.03525699
## 33  6.5340816 0.17750 0.03525699
## 34  6.7379592 0.17750 0.03525699
## 35  6.9418367 0.17750 0.03525699
## 36  7.1457143 0.17750 0.03525699
## 37  7.3495918 0.17750 0.03525699
## 38  7.5534694 0.17875 0.03537988
## 39  7.7573469 0.17875 0.03537988
## 40  7.9612245 0.17875 0.03537988
## 41  8.1651020 0.17875 0.03537988
## 42  8.3689796 0.17875 0.03537988
## 43  8.5728571 0.17875 0.03537988
## 44  8.7767347 0.17875 0.03537988
## 45  8.9806122 0.17875 0.03537988
## 46  9.1844898 0.17875 0.03537988
## 47  9.3883673 0.18000 0.03343734
## 48  9.5922449 0.17875 0.03537988
## 49  9.7961224 0.17750 0.03525699
## 50 10.0000000 0.17750 0.03525699
```

```
best2 <- svm_tune2$best.model


# part e: Take the best performance from part d and set that equal to the cost input

cat("Training Error Rate:", 100 * calc_error_rate(best2, OJ_train, OJ_train$Purchase), "%\n")
```

```
## Training Error Rate: 14.375 %
```

```
cat("Test Error Rate:", 100 * calc_error_rate(best2, OJ_test, OJ_test$Purchase), "%\n")
```

```
## Test Error Rate: 17.77778 %
```

```
#(g) Repeat parts (b) through (e) using a support vector machine with a polynomial kernel. Set degree=2

# part b: fitting the support vector
set.seed(5208)

svm_poly <- svm(Purchase ~ . , data = OJ_train, kernel = "poly", degree = 2)
summary(svm_poly)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = OJ_train, kernel = "poly", degree = 2)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  polynomial
##        cost:  1
##      degree:  2
##      coef.0:  0
##
## Number of Support Vectors:  458
##
```

```
##  ( 234 224 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

```r
# part c: Calculating the test and error rate
cat("Training Error Rate:", 100 * calc_error_rate(svm_poly, OJ_train, OJ_train$Purchase), "%\n")
```

```
## Training Error Rate: 17.125 %
```

```r
cat("Test Error Rate:", 100 * calc_error_rate(svm_poly, OJ_test, OJ_test$Purchase), "%\n")
```

```
## Test Error Rate: 21.48148 %
```

```r
# part d: Adding the costs from .01 to 10

set.seed(5208)
svm_tune3 <- tune(svm, Purchase ~ . , data = OJ_train, kernel = "poly",
                 degree = 2, ranges = list(cost = seq(0.01, 10), length = 50))

summary(svm_tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##       cost
##  0.8255102
##
## - best performance: 0.17375
##
## - Detailed performance results:
##          cost   error dispersion
## 1   0.0100000 0.17375 0.02729087
## 2   0.2138776 0.17500 0.03486083
## 3   0.4177551 0.17625 0.03356689
## 4   0.6216327 0.17500 0.03486083
## 5   0.8255102 0.17375 0.03356689
## 6   1.0293878 0.17500 0.03818813
## 7   1.2332653 0.17500 0.03818813
## 8   1.4371429 0.17625 0.03747684
## 9   1.6410204 0.17625 0.03747684
## 10  1.8448980 0.17625 0.03747684
## 11  2.0487755 0.17500 0.03584302
## 12  2.2526531 0.17375 0.03557562
## 13  2.4565306 0.17375 0.03557562
## 14  2.6604082 0.17375 0.03557562
## 15  2.8642857 0.17375 0.03557562
## 16  3.0681633 0.17375 0.03557562
## 17  3.2720408 0.17375 0.03557562
## 18  3.4759184 0.17375 0.03557562
```

```
## 19   3.6797959 0.17500 0.03535534
## 20   3.8836735 0.17375 0.03653860
## 21   4.0875510 0.17375 0.03653860
## 22   4.2914286 0.17375 0.03653860
## 23   4.4953061 0.17375 0.03653860
## 24   4.6991837 0.17375 0.03653860
## 25   4.9030612 0.17375 0.03653860
## 26   5.1069388 0.17375 0.03653860
## 27   5.3108163 0.17375 0.03653860
## 28   5.5146939 0.17500 0.03726780
## 29   5.7185714 0.17500 0.03726780
## 30   5.9224490 0.17500 0.03726780
## 31   6.1263265 0.17625 0.03928617
## 32   6.3302041 0.17750 0.03525699
## 33   6.5340816 0.17750 0.03525699
## 34   6.7379592 0.17750 0.03525699
## 35   6.9418367 0.17750 0.03525699
## 36   7.1457143 0.17750 0.03525699
## 37   7.3495918 0.17750 0.03525699
## 38   7.5534694 0.17875 0.03537988
## 39   7.7573469 0.17875 0.03537988
## 40   7.9612245 0.17875 0.03537988
## 41   8.1651020 0.17875 0.03537988
## 42   8.3689796 0.17875 0.03537988
## 43   8.5728571 0.17875 0.03537988
## 44   8.7767347 0.17875 0.03537988
## 45   8.9806122 0.17875 0.03537988
## 46   9.1844898 0.17875 0.03537988
## 47   9.3883673 0.18000 0.03343734
## 48   9.5922449 0.17875 0.03537988
## 49   9.7961224 0.17750 0.03525699
## 50  10.0000000 0.17750 0.03525699
```

```r
best3 <- svm_tune3$best.model
# part e: Take the best performance from part d and set that equal to the cost input

svm_poly2 <- svm(Purchase ~ . , data = OJ_train, kernel = "poly",
                degree = 2, cost = svm_tune$best.parameters$cost)

cat("Training Error Rate:", 100 * calc_error_rate(best3, OJ_train, OJ_train$Purchase), "%\n")
```

```
## Training Error Rate: 14.25 %
```

```r
cat("Test Error Rate:", 100 * calc_error_rate(best3, OJ_test, OJ_test$Purchase), "%\n")
```

```
## Test Error Rate: 18.14815 %
```

(h) Overall, which approach seems to give the best results on this data?

Overall, radial basis kernel seems to be producing minimum misclassification error on training set but the linear kernel performs better on test data.

Chapter 10, Applied Exercise 10

A: Generate a simulated data set with 20 observations in each of three classes (i.e. 60 observations total), and 50 variables. Hint: There are a number of functions in R that you can use to generate data. One example is the rnorm() function; runif() is another option. Be sure to add a mean shift to the observations in each class so that there are three distinct classes.

```r
#(a) Generate a simulated data set with 20 observations in each of three classes
set.seed(11)

## Creating variables
class1 <- data.frame(replicate(50, rnorm(20, mean = -2))) #simulated random data
class2 <- data.frame(replicate(50, rnorm(20, mean = 0))) #simulated random data
class3 <- data.frame(replicate(50, rnorm(20, mean = 2))) #simulated random data

set.seed(11)
## Another way to create variables that generates a slightly different image
class1 <- matrix(rnorm(20*50, mean = -2), nrow=20) #simulated random data
class2 <- matrix(rnorm(20*50, mean = 0), nrow=20) #simulated random data
class3 <- matrix(rnorm(20*50, mean = 2), nrow=20) #simulated random data

df <- data.frame(rbind(class1,class2,class3))

mean(rowMeans(df)[1:20])
```

```
## [1] -1.991209
```

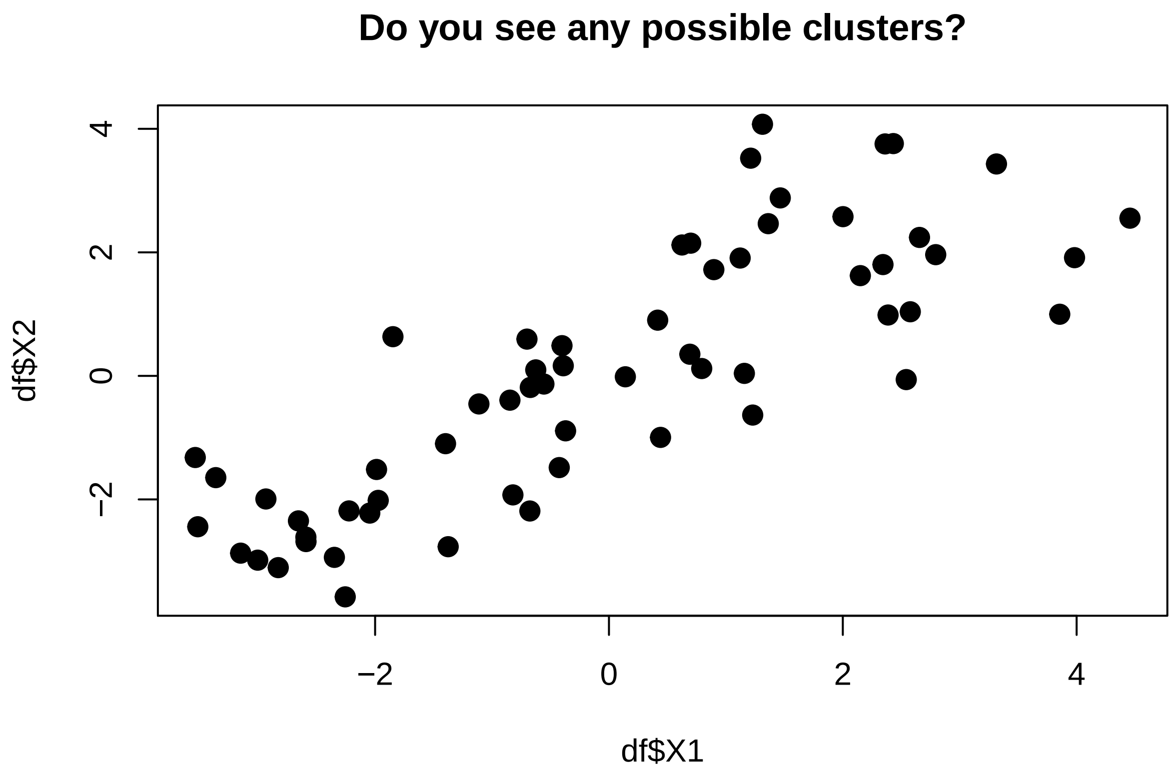
```r
mean(rowMeans(df)[21:40])
```

```
## [1] -0.002607081
```

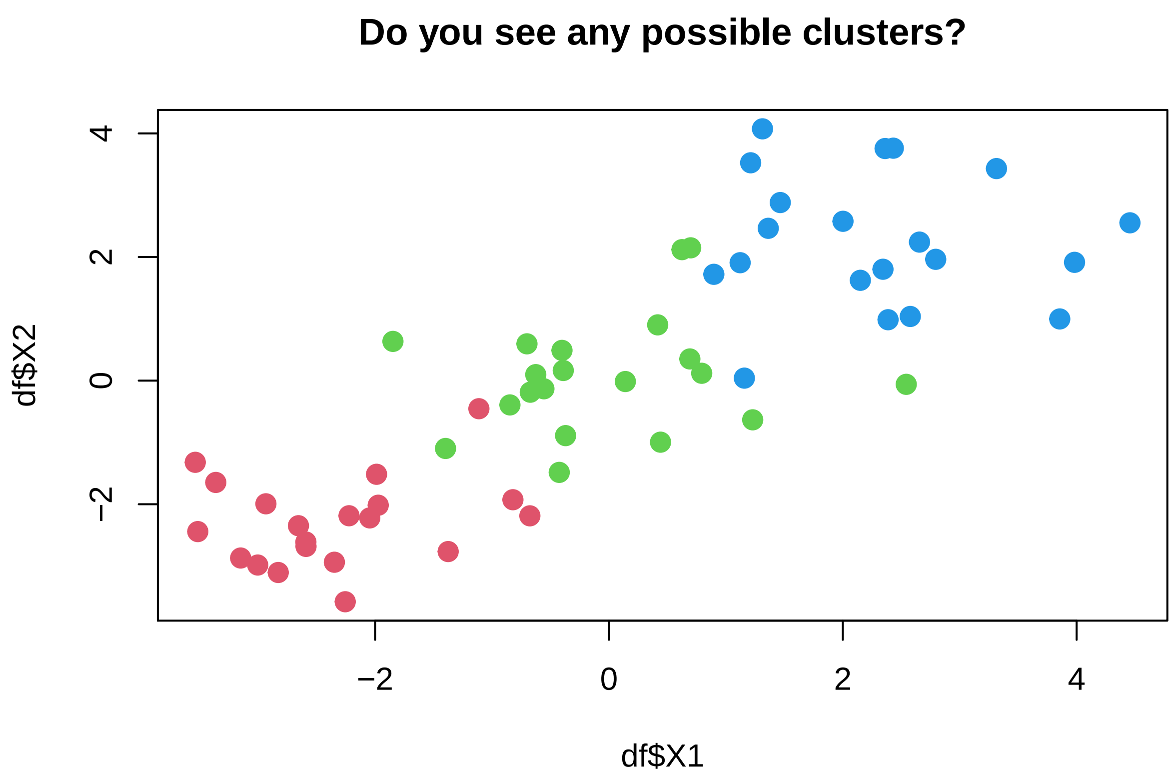
```r
mean(rowMeans(df)[41:60])
```

```
## [1] 2.040481
```

```r
plot(df$X1, df$X2, main="Do you see any possible clusters?",
      pch =20, cex =2)
```




```
## Creating labels for each class
Kclasses = c(rep(1,20), rep(2,20), rep(3,20))

plot(df$X1, df$X2, main="Do you see any possible clusters?",
      pch =20, cex =2, col=Kclasses+1)
```



B: Perform PCA on the 60 observations and plot the first two principal component score vectors. Use a different color to indicate the observations in each of the three classes. If the three classes appear separated in this plot, then continue on to part (c). If not, then return to part (a) and modify the simulation so that there is greater separation between the three classes. Do not continue to part (c) until the three classes show at least some separation in the first two principal component score vectors.

```r
set.seed(11)
## Performing PCA on df
pr.out <- prcomp(df, scale=TRUE)

## Importance of Components
summary(pr.out)
```

```
## Importance of components:
##                           PC1     PC2     PC3     PC4     PC5     PC6     PC7
## Standard deviation     6.0925 1.00545 0.88449 0.84805 0.83033 0.81802 0.79885
## Proportion of Variance 0.7424 0.02022 0.01565 0.01438 0.01379 0.01338 0.01276
## Cumulative Proportion  0.7424 0.76259 0.77823 0.79262 0.80641 0.81979 0.83255
##                            PC8     PC9    PC10    PC11    PC12    PC13    PC14
## Standard deviation     0.76922 0.73733 0.71342 0.69425 0.67106 0.65815 0.65650
## Proportion of Variance 0.01183 0.01087 0.01018 0.00964 0.00901 0.00866 0.00862
## Cumulative Proportion  0.84439 0.85526 0.86544 0.87508 0.88408 0.89275 0.90137
##                           PC15    PC16    PC17    PC18    PC19    PC20    PC21
## Standard deviation     0.62972 0.60915 0.58952 0.56647 0.55064 0.53166 0.51634
## Proportion of Variance 0.00793 0.00742 0.00695 0.00642 0.00606 0.00565 0.00533
## Cumulative Proportion  0.90930 0.91672 0.92367 0.93009 0.93615 0.94181 0.94714
##                           PC22    PC23    PC24    PC25    PC26    PC27    PC28
```

```
## Standard deviation     0.47600 0.46687 0.45605 0.44028 0.43110 0.41947 0.40843
## Proportion of Variance 0.00453 0.00436 0.00416 0.00388 0.00372 0.00352 0.00334
## Cumulative Proportion  0.95167 0.95603 0.96019 0.96407 0.96778 0.97130 0.97464
##                           PC29    PC30    PC31    PC32    PC33    PC34    PC35
## Standard deviation     0.40329 0.37797 0.36256 0.34861 0.31094 0.29419 0.27082
## Proportion of Variance 0.00325 0.00286 0.00263 0.00243 0.00193 0.00173 0.00147
## Cumulative Proportion  0.97789 0.98075 0.98338 0.98581 0.98774 0.98947 0.99094
##                           PC36    PC37    PC38    PC39   PC40    PC41    PC42
## Standard deviation     0.25917 0.24713 0.23921 0.21338 0.1994 0.19685 0.17973
## Proportion of Variance 0.00134 0.00122 0.00114 0.00091 0.0008 0.00077 0.00065
## Cumulative Proportion  0.99228 0.99350 0.99465 0.99556 0.9963 0.99713 0.99777
##                           PC43    PC44   PC45   PC46    PC47    PC48    PC49
## Standard deviation     0.16614 0.14872 0.1421 0.1226 0.10220 0.09138 0.06846
## Proportion of Variance 0.00055 0.00044 0.0004 0.0003 0.00021 0.00017 0.00009
## Cumulative Proportion  0.99833 0.99877 0.9992 0.9995 0.99968 0.99985 0.99994
##                           PC50
## Standard deviation     0.05324
## Proportion of Variance 0.00006
## Cumulative Proportion  1.00000
```

```r
names(pr.out)
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

```r
## principal component loading vector for PC1 and PC2 (eigenvectors)
pr.out$rotation[,1:2]
```

```
##           PC1         PC2
## X1  0.1472509  0.04595396
## X2  0.1481135  0.03263273
## X3  0.1458147  0.03942399
## X4  0.1392895  0.22560966
## X5  0.1451974  0.20457303
## X6  0.1407336 -0.08035216
## X7  0.1332540  0.05426612
## X8  0.1359890  0.08400194
## X9  0.1498258 -0.07498545
## X10 0.1412016 -0.05220658
## X11 0.1366251 -0.24979809
## X12 0.1427552 -0.31900871
## X13 0.1383078 -0.16046672
## X14 0.1361809  0.23507268
## X15 0.1389910 -0.13521748
## X16 0.1398818  0.11802984
## X17 0.1398109  0.06684219
## X18 0.1409540 -0.14702354
## X19 0.1410075  0.08674256
## X20 0.1359544  0.05781093
## X21 0.1337148  0.02678342
## X22 0.1464989  0.04387719
## X23 0.1386568  0.11963186
## X24 0.1433014 -0.25205386
## X25 0.1253455  0.33241627
## X26 0.1376508 -0.29226860
## X27 0.1424681 -0.05633898
```

```
## X28 0.1423733 -0.00319691
## X29 0.1416582  0.10951215
## X30 0.1444054  0.04266241
## X31 0.1460609 -0.20603375
## X32 0.1375750 -0.13541850
## X33 0.1428687 -0.05906949
## X34 0.1452764  0.15292579
## X35 0.1442630 -0.03820505
## X36 0.1342506 -0.22534922
## X37 0.1437288  0.04626775
## X38 0.1466354 -0.02995913
## X39 0.1383515  0.15316988
## X40 0.1462410 -0.09014020
## X41 0.1460626 -0.15523497
## X42 0.1386865  0.01768255
## X43 0.1438416  0.12236541
## X44 0.1458618  0.03657175
## X45 0.1472727 -0.02644143
## X46 0.1440026  0.11269673
## X47 0.1346191  0.01982171
## X48 0.1386932  0.04339451
## X49 0.1427917  0.19757290
## X50 0.1467951 -0.00274777
```

```r
## The sum of squares of the loadings for each Principal Component will equal 1
sum(pr.out$rotation[,1]^2)
```

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

```r
## The principal component scores for PC1 for each observation in the data set
## The goal of PCA is to maximize this variation
pr.out$x[,1]
```

```
##  [1] -8.31023882 -6.92518185 -7.47723969 -7.30529582 -7.05688882 -6.90400111
##  [7] -8.20673690 -7.08466330 -7.79664376 -7.63616661 -7.28223000 -7.20467088
## [13] -6.86939294 -7.16990333 -7.20381158 -7.24897406 -7.36501526 -7.02051564
## [19] -6.81863153 -7.96345516  0.24426594  0.83637838 -0.59341276  0.14846070
## [25] -0.76127780 -0.77110050  0.43839522 -0.50193833 -0.41600126  0.43737623
## [31] -0.82695861 -0.77221208 -0.60327046  0.10023683  0.21888296  0.18717755
## [37]  0.34621140  0.37544306  0.46620369  0.03470056  7.27139595  7.56791226
## [43]  7.60343685  7.92411552  7.39690773  7.42433817  7.13903922  8.03146822
## [49]  7.52297837  7.21320553  7.10011900  8.20855456  7.39443983  6.68223385
## [55]  7.10342650  7.44831688  6.34478256  8.03554063  7.48384969  7.36603502
```

```r
mean(pr.out$x[,1])
```

```
## [1] -5.506301e-17
```

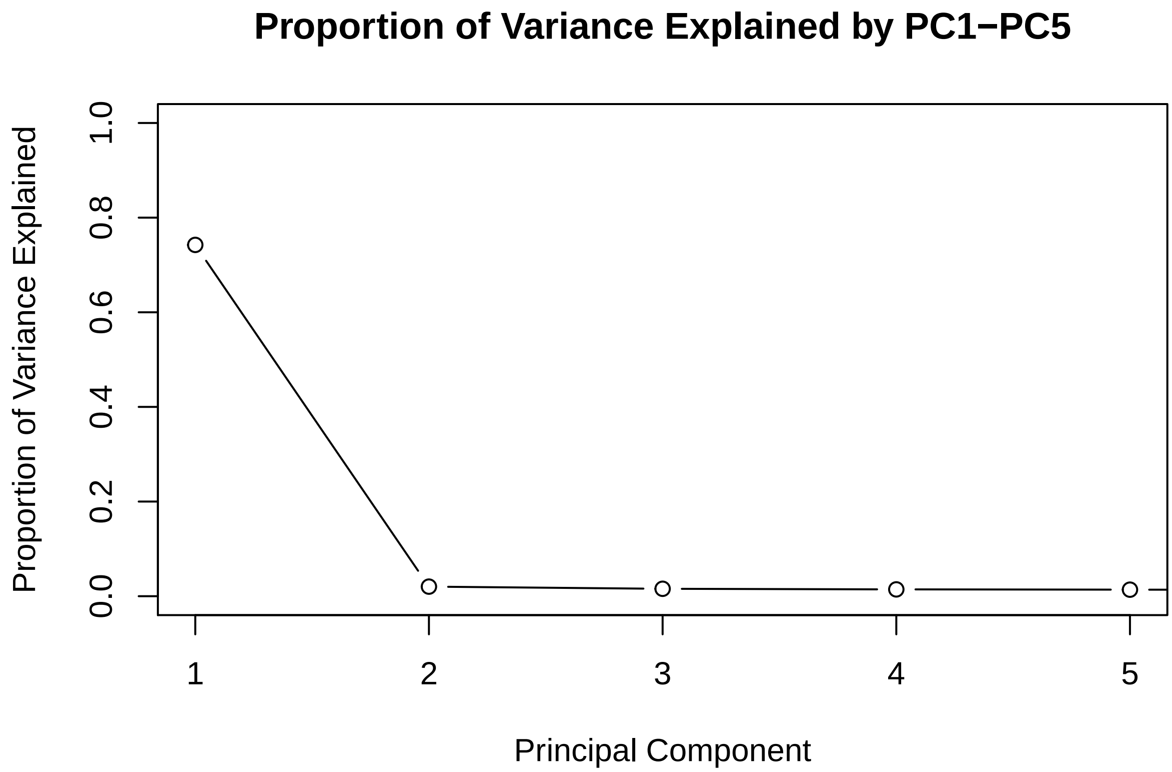
```r
## Proportion of variance explained by PC1
var(pr.out$x[,1])
```

```
## [1] 37.11837
```

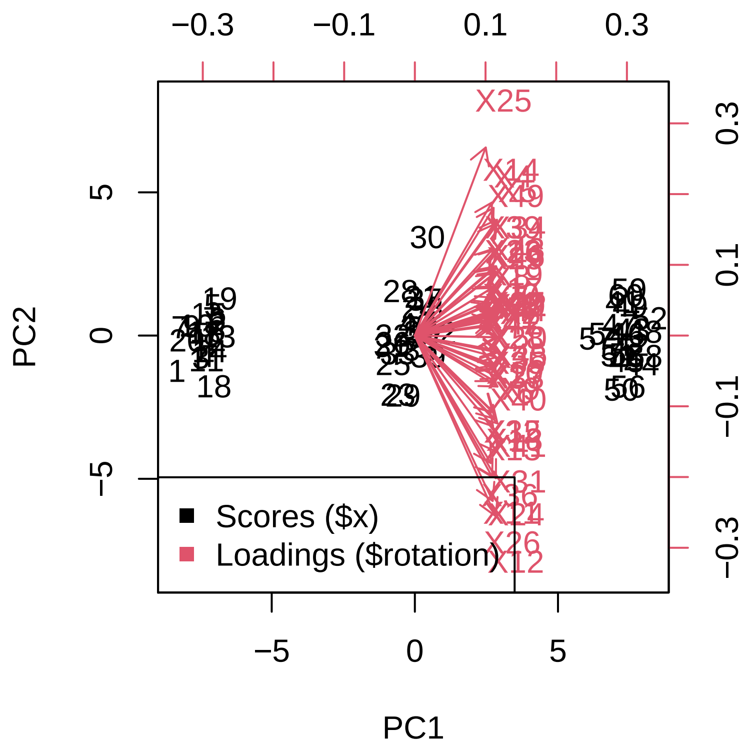
```r
pr.var <- pr.out$sdev^2
pr.var[1]
```

```
## [1] 37.11837
```

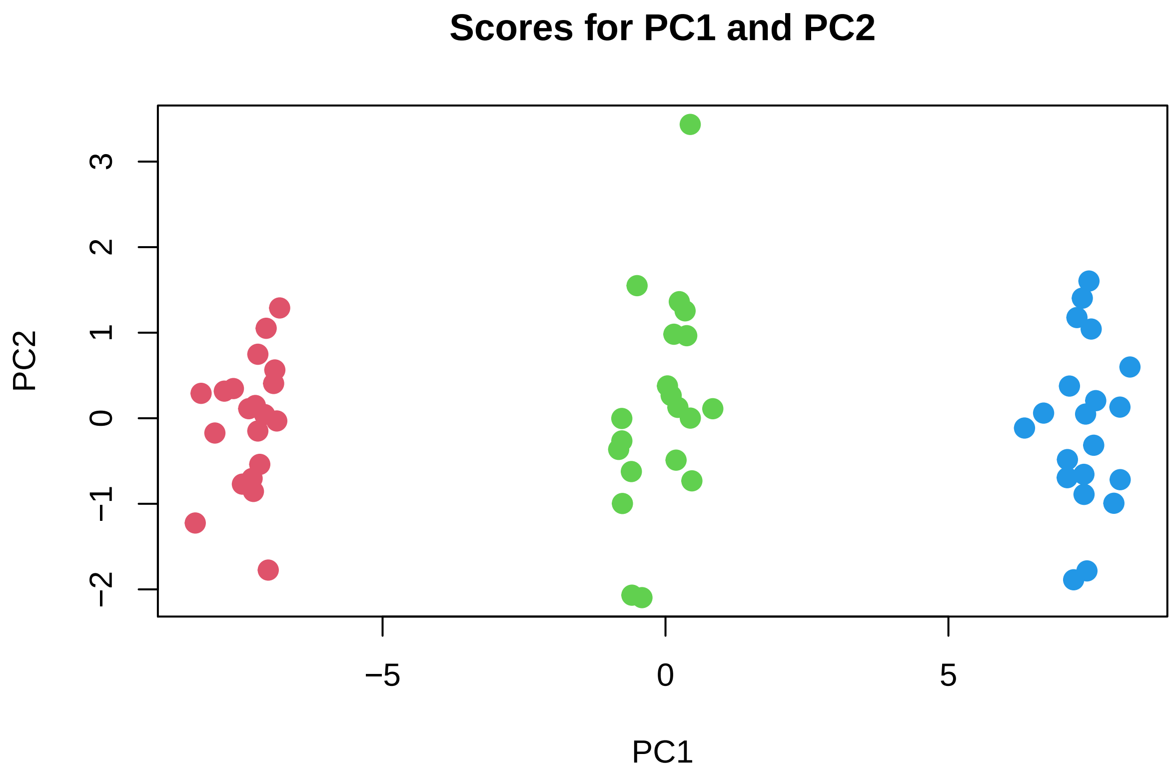
```r
pve <- pr.var/sum(pr.var)


## Plotting the proportion of variance explained
plot(pve, main = "Proportion of Variance Explained by PC1-PC5",
     xlab="Principal Component",
     ylab="Proportion of Variance Explained",
     ylim=c(0,1), xlim=c(1,5),
     type='b')
```




```r
## Plotting the scores for the first two principle components and their loading vectors
biplot(pr.out, scale=0)
legend('bottomleft', c('Scores ($x)', 'Loadings ($rotation)'),
       col=c(1,2), pch=15)
```




```r
## Plotting scores for PC1 and PC2
plot(pr.out$x, col=Kclasses+1,
     pch=20, cex=2,
     main='Scores for PC1 and PC2')
```



C: Perform K-means clustering of the observations with K = 3. How well do the clusters that you obtained in K-means clustering compare to the true class labels? Hint: You can use the table() function in R to compare the true class labels to the class labels obtained by clustering. Be careful how you interpret the results: K-means clustering will arbitrarily number the clusters, so you cannot simply check whether the true class labels and clustering labels are the same.

```
set.seed(11)


km.out3 <- kmeans(df,3,nstart=20) #nstart 20: random sets are chosen
km.out3$cluster
```

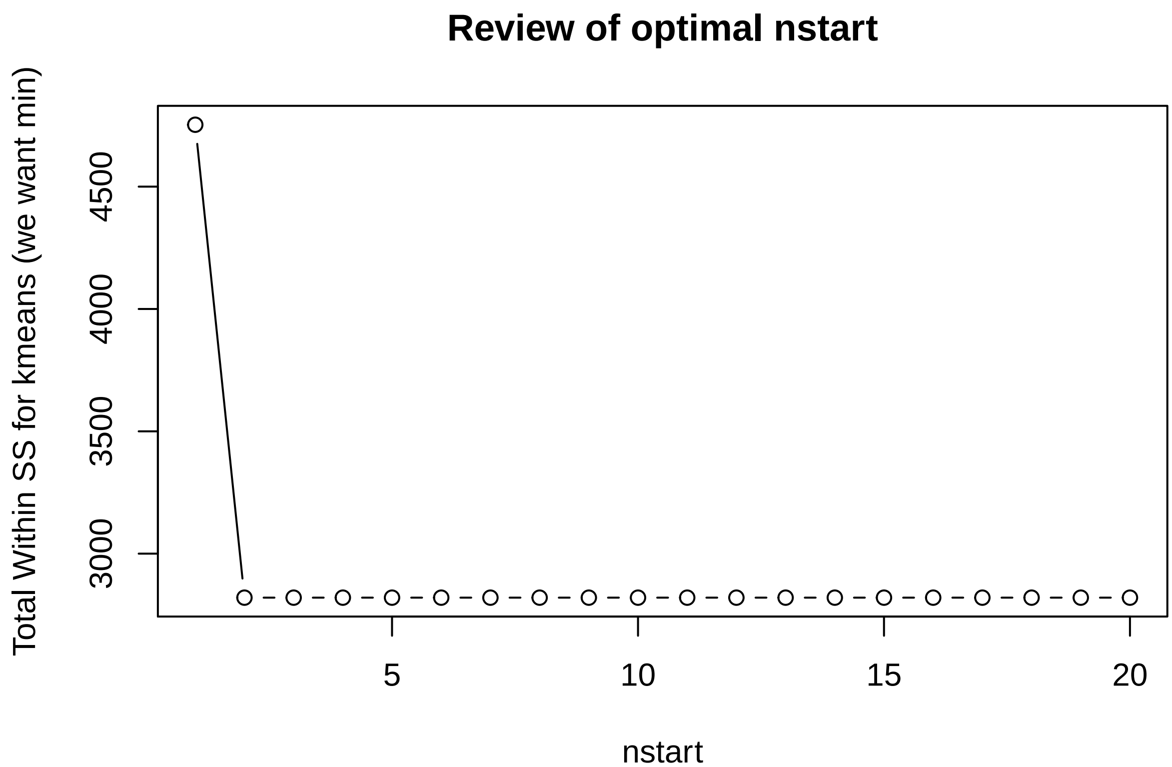
```
##  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [39] 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

```
km.out3$tot.withinss #we want to minimize
```

```
## [1] 2820.231
```

```
#Let's look at why nstart argument matters:
nstart_withinss = rep(0,20)
set.seed(11)
for (i in 1:length(nstart_withinss)){
  km.out3 <- kmeans(df,3,nstart=i)
  nstart_withinss[i] <- km.out3$tot.withinss
}
plot(nstart_withinss, main = "Review of optimal nstart",
     xlab="nstart",
```

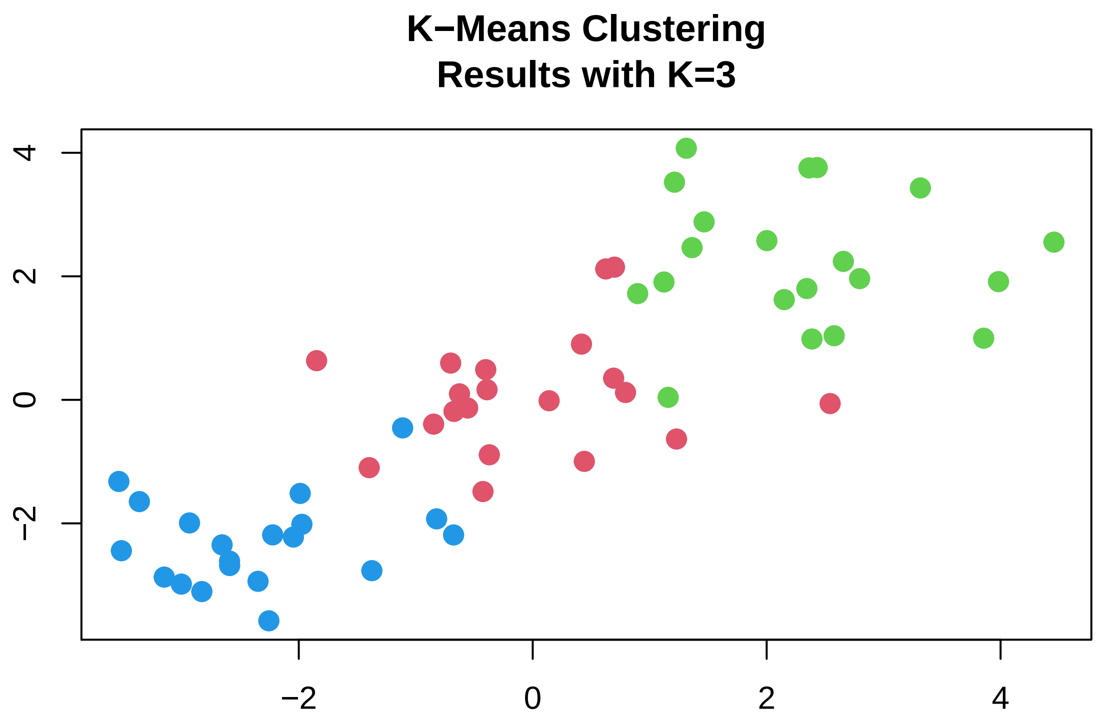
```
     ylab="Total Within SS for kmeans (we want min)",
     type='b')
```



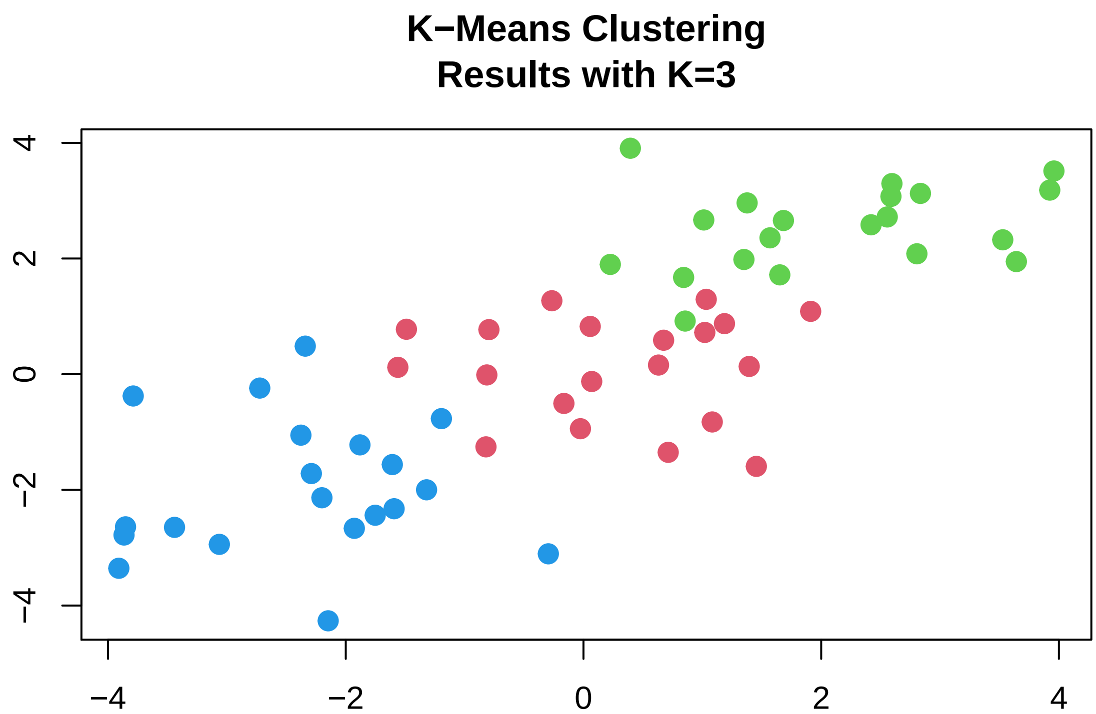

```
#Table of clusters vs how many classes we have:
table(km.out3$cluster, Kclasses, dnn=c("Clusters","Class Labels"))
```

```
##          Class Labels
## Clusters  1  2  3
##         1  0 20  0
##         2  0  0 20
##         3 20  0  0
```

```
#Plots:
plot(df$X1, df$X2, col =(km.out3$cluster +1) , main="K-Means Clustering
Results with K=3", xlab ="",ylab ="", pch =20, cex =2)
```




```
plot(df$X49, df$X50, col =(km.out3$cluster +1) , main="K-Means Clustering
Results with K=3", xlab ="",ylab ="", pch =20, cex =2)
```



D: Perform K-means clustering with K = 2. Describe your results.

```
set.seed(11)
km.out2 <- kmeans(df,2,nstart=20)
km.out2$cluster
```

```
##  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [39] 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

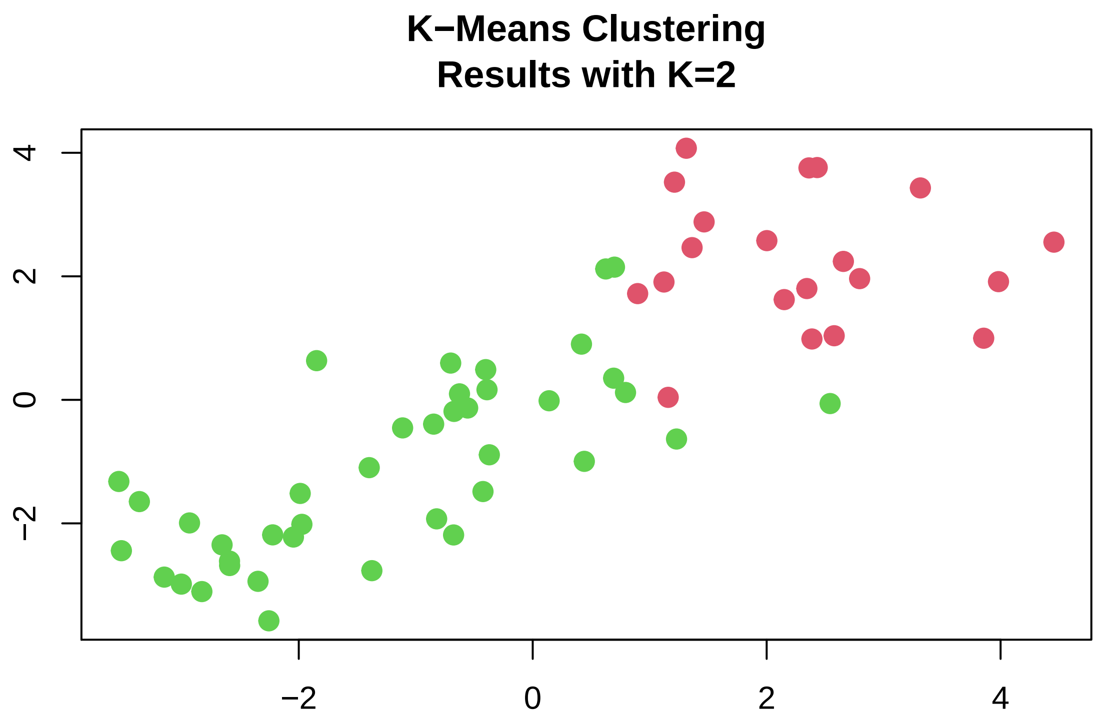
```
km.out2$tot.withinss
```

```
## [1] 4842.328
```

```
table(km.out2$cluster, Kclasses, dnn=c("Clusters","Class Labels"))
```

```
##         Class Labels
## Clusters  1  2  3
##        1  0  0 20
##        2 20 20  0
```

```
plot(df$X1, df$X2, col =(km.out2$cluster +1) , main="K-Means Clustering
Results with K=2", xlab ="",ylab ="", pch =20, cex =2)
```



Describe your results: The model splits K into 2 classes, which is a much higher total within Sum of Squares of 4842 compared to k = 3: totalwithinSS: 2820. We want to minimize this without overfitting. You can see in the graph, using two clusters is not an optimal way to measure this data compared to k=3.

E: Now perform K-means clustering with K = 4. Describe your results.

```
set.seed(11)
km.out4 <- kmeans(df,4,nstart=20)
km.out4$cluster
```

```
##  [1] 4 4 1 4 4 1 1 1 4 1 4 1 4 1 1 1 4 4 4 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [39] 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

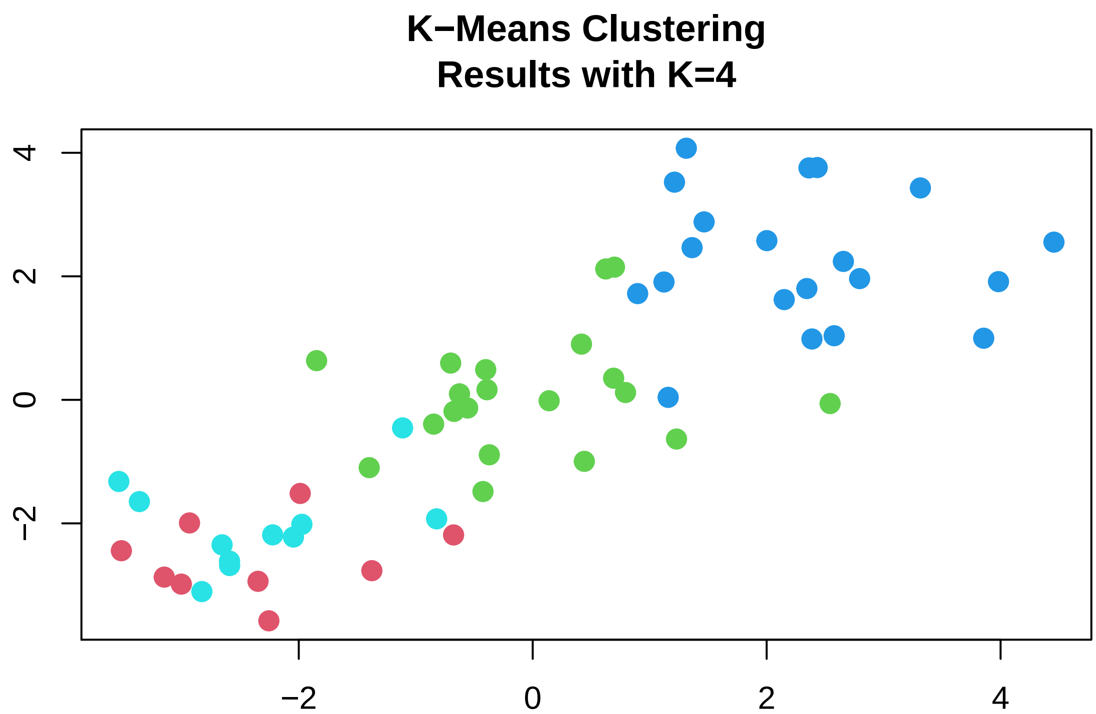
```
km.out4$tot.withinss
```

```
## [1] 2722.595
```

```
table(km.out4$cluster, Kclasses, dnn=c("Clusters","Class Labels"))
```

```
##          Class Labels
## Clusters  1  2  3
##         1  9  0  0
##         2  0 20  0
##         3  0  0 20
##         4 11  0  0
```

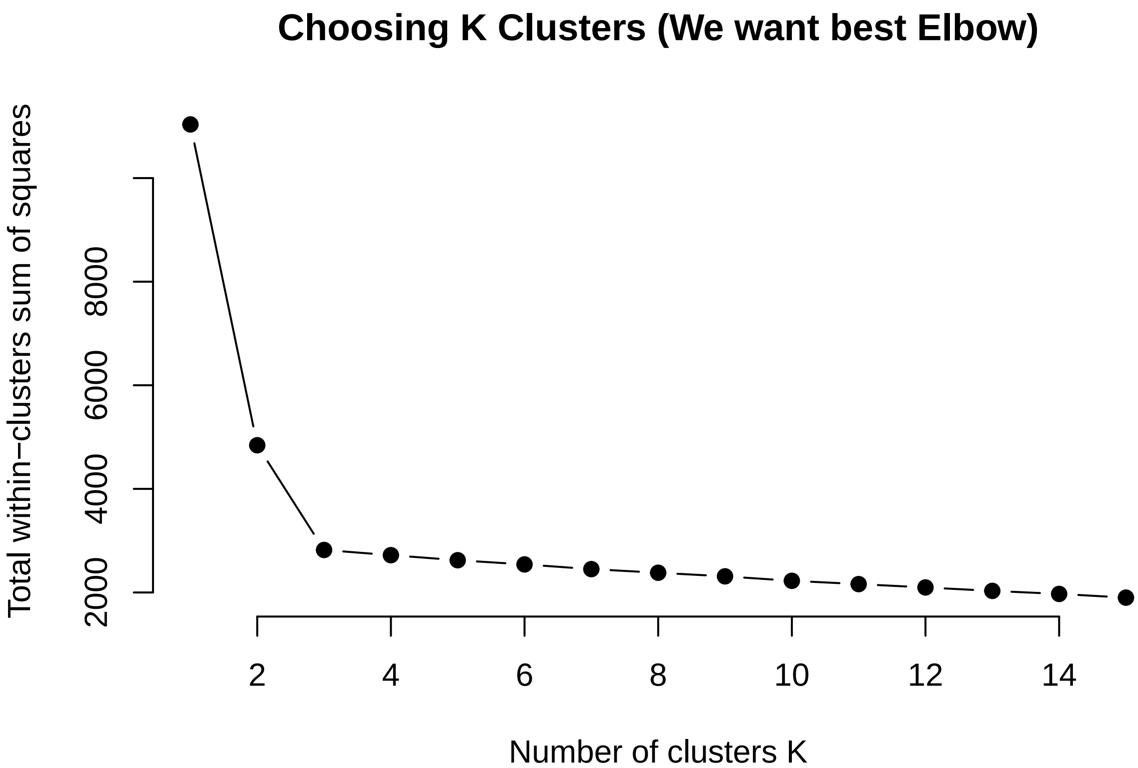
```
plot(df$X1, df$X2, col =(km.out4$cluster +1) , main="K-Means Clustering
Results with K=4", xlab ="",ylab ="", pch =20, cex =2)
```



Describe your results: The model splits K into 4 classes now. Spefically, one of the classes is split into 2, compared to k=3. You can see in the graph, using 4 clusters is not doing well to measure the bottom right cluster.

For the total within sum of squares, we actually do get a lower 2722.595, but it is actually a much smaller jump. This is always going to decrease as k gets higher, since that is what kmeans algorithm is trying to do. So to not force an overfit, we should look at an elbow plot and find the optimal k from there.

```
#Pick the best K using elbow plot
set.seed(11)
kmax<-15
totwithiness<-rep(0,kmax)
for (k in 1:kmax){
  totwithiness[k] =kmeans(df,k,nstart=20)$tot.withinss
}
plot(1:kmax,totwithiness,type="b",pch= 19, frame = FALSE,
     main = "Choosing K Clusters (We want best Elbow)",
     xlab="Number of clusters K", ylab="Total within-clusters sum of squares")
```



In the above plot, you can see the elbow bends at k = 3 which is why that is the best k to use.

F: Now perform K-means clustering with K = 3 on the first two principal component score vectors, rather than on the raw data. That is, perform K-means clustering on the 60 × 2 matrix of which the first column is the first principal component score vector, and the second column is the second principal.

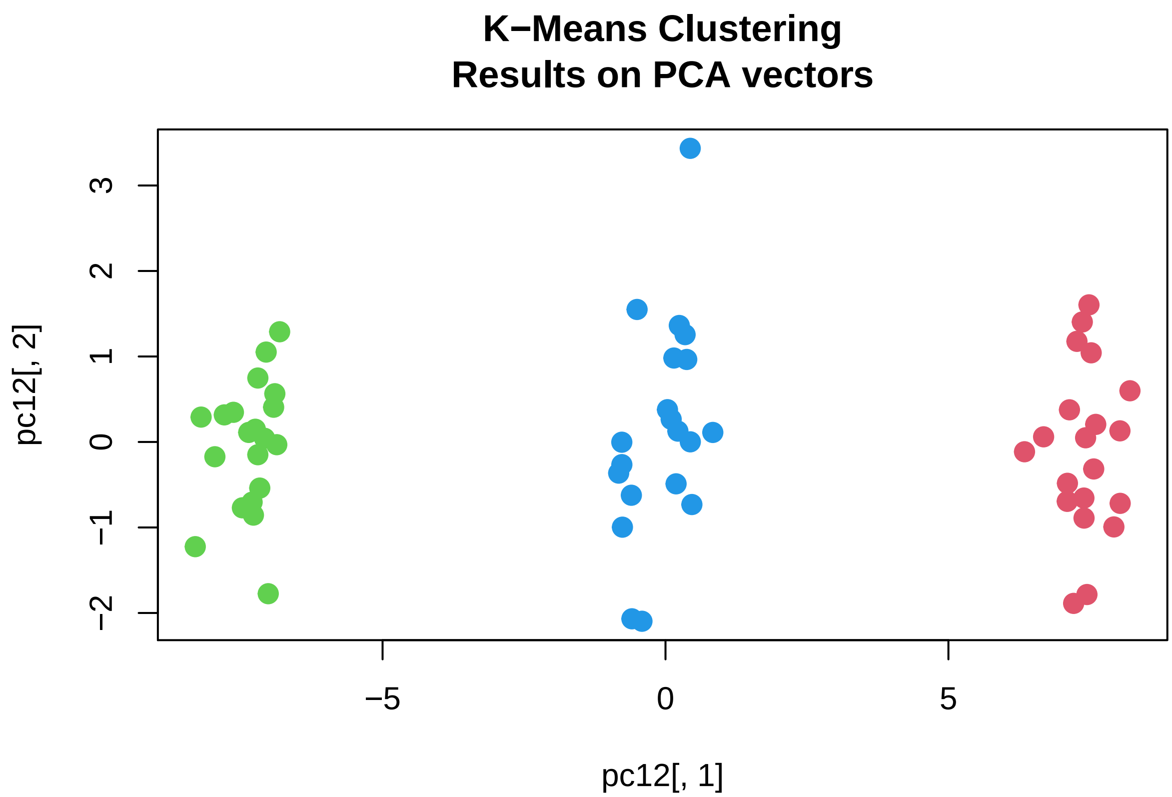
```
set.seed(10)
pc12 <- pr.out$x[,1:2]

km.out.pca <- kmeans(pc12,3,nstart=50)
km.out.pca$cluster
```

```
##  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
## [39] 3 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
table(km.out.pca$cluster, Kclasses, dnn=c("Clusters","Class Labels"))
```

```
##         Class Labels
## Clusters  1  2  3
##        1  0  0 20
##        2 20  0  0
##        3  0 20  0
```

```
plot(pc12[,1], pc12[,2], col =(km.out.pca$cluster +1) , main="K-Means Clustering
Results on PCA vectors", pch =20, cex =2)
```



G: Using the scale() function, perform K-means clustering with K = 3 on the data after scaling each variable to have standard deviation one. How do these results compare to those obtained in (b)? Explain.

```
set.seed(11)

cat("SD before scale:", sd(df$X1),"\n")
```

```
## SD before scale: 2.130162
```

```
scaled_df <- data.frame(scale(df))

cat("SD after scale:", sd(scaled_df$X1),"\n")
```

```
## SD after scale: 1
```

```
km.out3scaled = kmeans(scaled_df,3,nstart=20) #scale
km.out3scaled$cluster
```

```
##  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [39] 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

```
km.out3scaled$tot.withinss
```

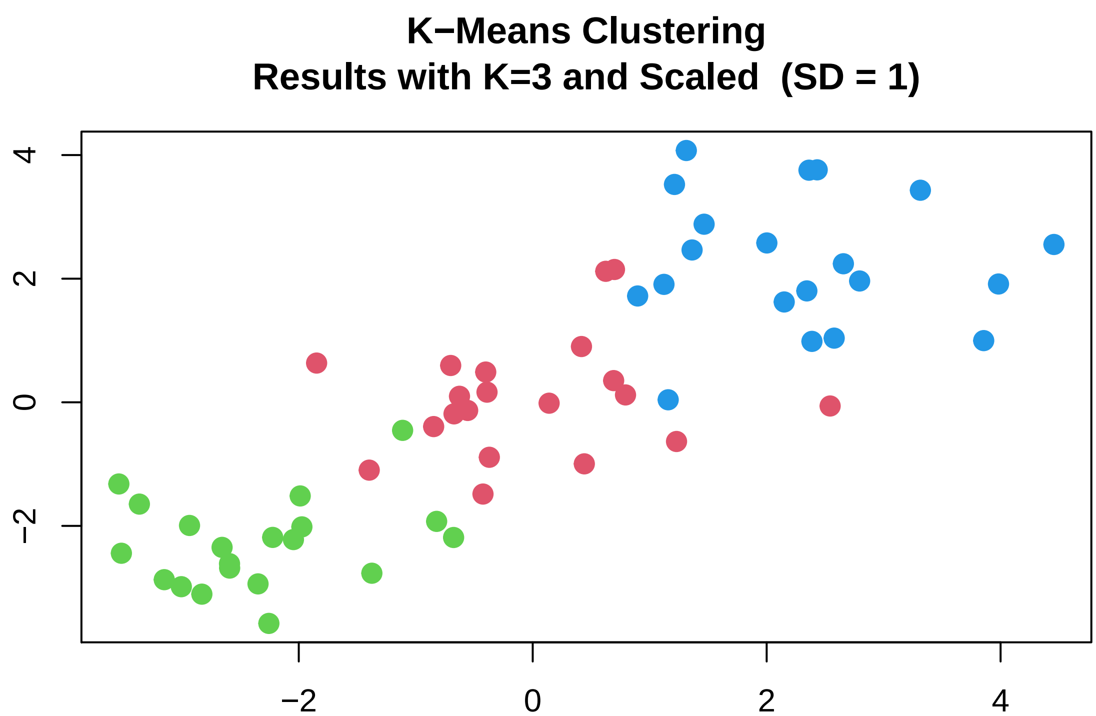
```
## [1] 760.2364
```

```
table(km.out3scaled$cluster, Kclasses, dnn=c("Clusters","Class Labels"))
```

```
##          Class Labels
## Clusters  1  2  3
##        1  0 20  0
```

```
##         2 20  0  0
##         3  0  0 20
```

```
plot(df$X1, df$X2, col =(km.out3scaled$cluster +1) ,main="K-Means Clustering
Results with K=3 and Scaled  (SD = 1)", xlab ="",ylab ="", pch =20, cex =2)
```



The scale() of kmeans plots the same classes as without the scale because the mean shifts we made were already standardized. We took a mean of 2, mean of 0, and mean of -2, which is what our classes is. By standardizing everything with a SD of 1, we did successfully scale the data but since the data was already comparable, this does not change the output.

It is important to scale data that is varying in scales, since k-means uses distance to measure between points, and if it was unscaled it would make one feature have a higher impact than the others. Since this is unsupervised, we are using the features as a measure.

Final Notes

Congratulations on completing 10 chapters of Machine Learning! You should all be proud of yourselves.

- Team 10

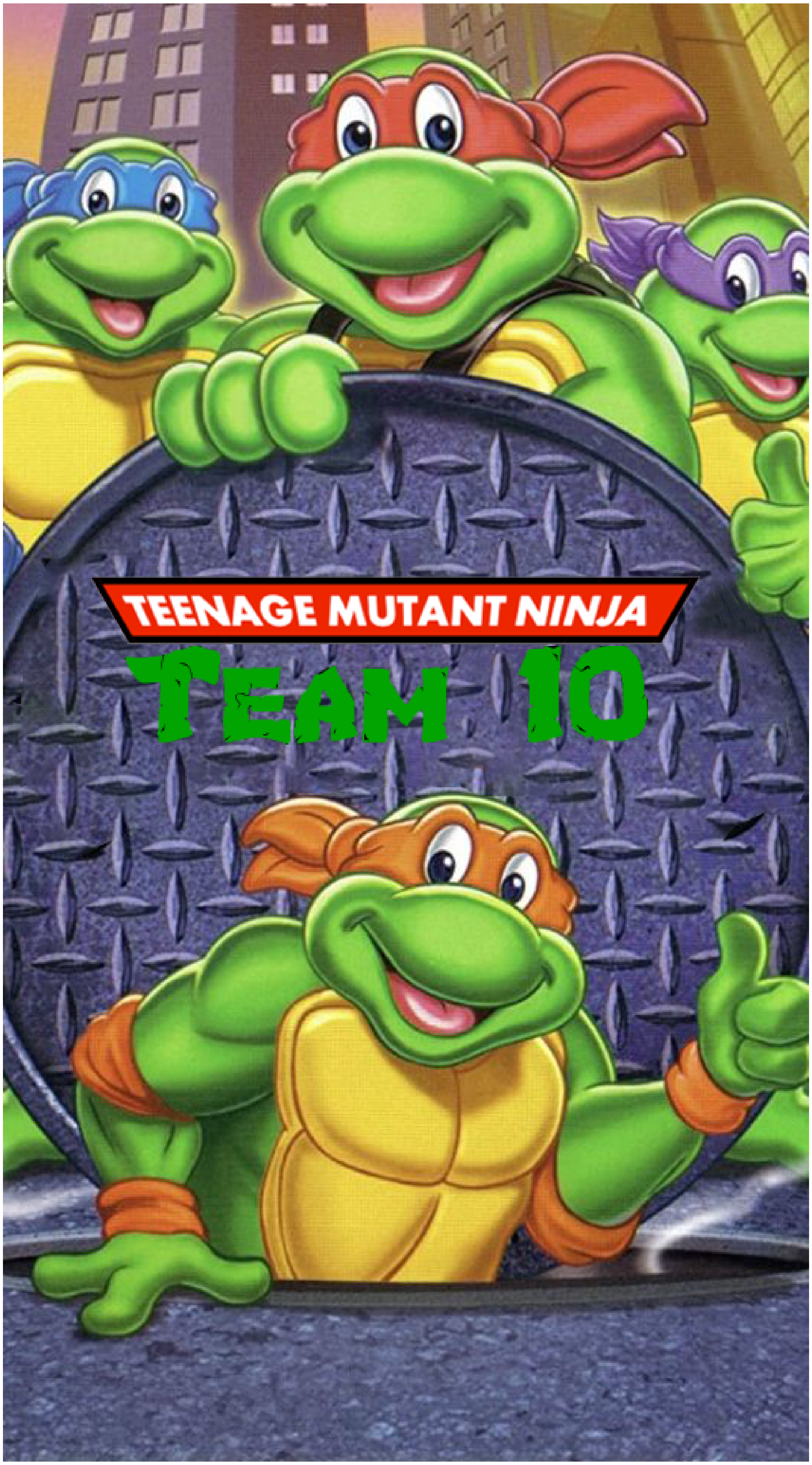


Figure 1: Team10Logo