Week-1-homework

Question 2.1 Describe a situation or problem from your job, everyday life, current events, etc., for which a classification model would be appropriate. List some (up to 5) predictors that you might use.

The stars have to align for me to actually do my homework so let's use a classifier to predict how likely it is that I will be working on my homework for any given day.

Some possible predictors:

1. Woke up before 11am (Yes/No)
2. Longest streak of working days (Range of positive integers)
3. Eaten lunch (Yes/No)
4. Number of homework questions remaining (Range of positive integers)
5. Weekend (Yes/No)

Question 2.2

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

```
# import relevant libraries
library(kernlab)
library(kknn)
```

```
# load data
df = read.delim('week 1 data-summer/data 2.2/credit_card_data-headers.txt')
#head(dataframe, 5)
```

Question 2.2 Part 1

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

For this question, we will be using the `ksvm` function in `kernlab` to classify the full credit card dataset.

First, we will want to find the optimal value of the regularization parameter `C`. In SVM, `C` is the parameter that controls the trade-off between maximizing the margin and minimizing the classification error. A higher value of C allows for more flexibility in the decision boundary, potentially leading to overfitting, while a lower value of C imposes a smoother decision boundary and may lead to underfitting. To tie this back in with the module 2 lectures, the parameter `C` is similar to λ, but not the exact same coefficient.

To find this optimal value of `C`, we will loop over a range of values starting from 10e-25 to 10e25. For each value of `C`, we construct an SVM model and calculate the corresponding accuracy, which represents the ratio of the number of *correct* predictions to the *total* number of predictions. Then store each accuracy in a vector `accuracy_vector`.

We will use accuracy as the metric to determine the best classifier and thus the best value of C.

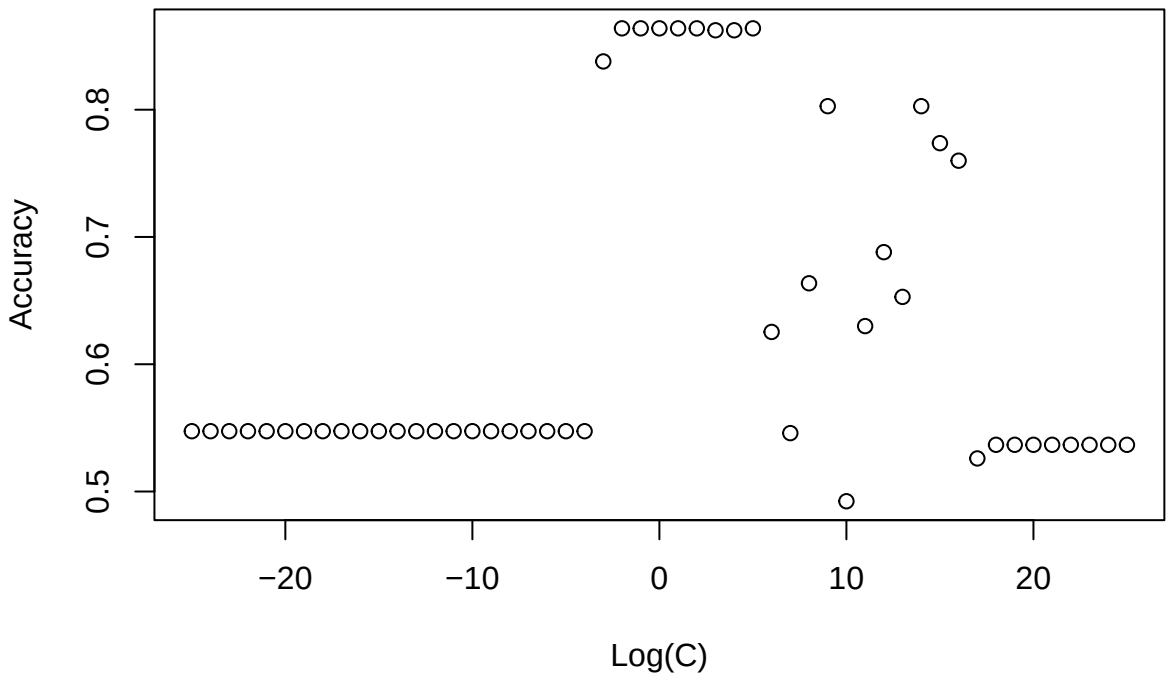
```r
# looping over values of C
logC <- c(-25:25)
c_vals <- 10^(-25:25)
accuracy_vector <- vector("numeric")

# build a svm for every value of C
set.seed(42)
for(c in c_vals) {
  model <- ksvm(as.matrix(df[, 1:10]),
                as.factor(df[, 11]),
                type="C-svc",
                kernel="vanilladot",
                C=c,
                scaled=T)

  # get inidividual predictions
  pred <- predict(model, df[,1:10])
  # calculate accuracy of SVM model
  accuracy <- sum(pred == df[, 11]) / nrow(df)
  # store accuracy in accuracy vector
  accuracy_vector <- c(accuracy_vector, accuracy)
}
```

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

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

Then, plot the values in `accuracy_vector` against the values of $log(C)$:

```r
# plotting accuracy against values of logC
plot(logC, accuracy_vector,
     xlab="Log(C)",
     ylab="Accuracy")
```



Use the `max()` function to find the highest accuracy in `accuracy_vector`.

```
# highest accuracy
max(accuracy_vector)
```

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

Then use the `which()` function to find the index of the highest accuracy.

```
# vector index for best value of C
which(accuracy_vector == max(accuracy_vector))
```

```
## [1] 24 25 26 27 28 31
```

We have a range of indices that corresponds to the highest accuracy score. This means that there are multiple values of `C` that fit our criteria. An index of 24 is a $log(C)$ value of -2, so with this range we have possible values of `C`: 0.01, 0.1, 1, 10, 100, 100000.

However as we covered earlier in the question, there are risks of over- and underfitting if the value of `C` is too high or too low. Thus, we will proceed with moderation and select the index in the middle of the range and set `C=1`.

Now we refit the model with the selected C value.

```
# build new svm with C=1
set.seed(42)
model <- ksvm(as.matrix(df[, 1:10]),
              as.factor(df[, 11]),
              type="C-svc",
              kernel="vanilladot",
              C = 1,
              scaled=T)
```

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

We can then obtain the coefficients and y-intercept of the classifier equation.

```
# calculate coeffs
a <- colSums(model@xmatrix[[1]] * model@coef[[1]])
a
```

```
##            A1            A2            A3            A8            A9
## -0.0011026642 -0.0008980539 -0.0016074557  0.0029041700  1.0047363456
##           A10           A11           A12           A14           A15
## -0.0029852110 -0.0002035179 -0.0005504803 -0.0012519187  0.1064404601
```

```
# calculate intercept
a0 <- -model@b
a0
```

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

Finally, we contruct the classifier equation:

$$0 = -0.0011026642A_1 - 0.0008980539A_2 - 0.0016074557A_3 + 0.0029041700A_8 + 1.0047363456A_9 - 0.0029852110A_{10} - 0.00020351$$

where A_1 to A_{15} are the corresponding column labels.

Just to be sure, let's also view all the predictions made by our model.

```
# viewing predictions
pred <- predict(model,df[,1:10])
pred
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
##  [75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [112] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [149] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [186] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
## [223] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
## [260] 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
## [297] 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
## [334] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
## [371] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [408] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [445] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [482] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [519] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
## [556] 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
## [593] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
## [630] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## Levels: 0 1
```

There appears to be a fairly even spread of 1s and 0s. Following the hint given in the homework, I believe this spread supports our hypothesis that `C=1` is a good value of `C`.

Question 2.2 Part 2

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

For this question, let's first understand what kernels are.

A kernel is a set of mathematical functions used to map input data points into a higher-dimensional space where the separation between two classes becomes easier. This allows SVM to solve complex non-linear problems. [source]

The `kernel` parameter in `ksvm` accepts a list of kernels that can be found in their documentation:

- **`rbfdot`** Radial Basis kernel "Gaussian"
- **`polydot`** Polynomial kernel
- **`vanilladot`** Linear kernel
- **`tanhdot`** Hyperbolic tangent kernel
- **`laplacedot`** Laplacian kernel
- **`besseldot`** Bessel kernel
- **`anovadot`** ANOVA RBF kernel
- **`splinedot`** Spline kernel
- **`stringdot`** String kernel

The `stringdot` kernel specifically is used for string data. As we are dealing only with numerical data, this kernel is not suited to our data and thus will be excluded from the comparison.

To find the best performing kernel, we will loop through a list of kernels and create a SVM using each kernel. Similar to the process in Q2.2.1, we will use accuracy as the metric for determining the best kernel for our model. Accuracy will be calculated for each SVM, then stored in the accuracy vector. We will use the value of `C` that we have determined in Q2.2.1.

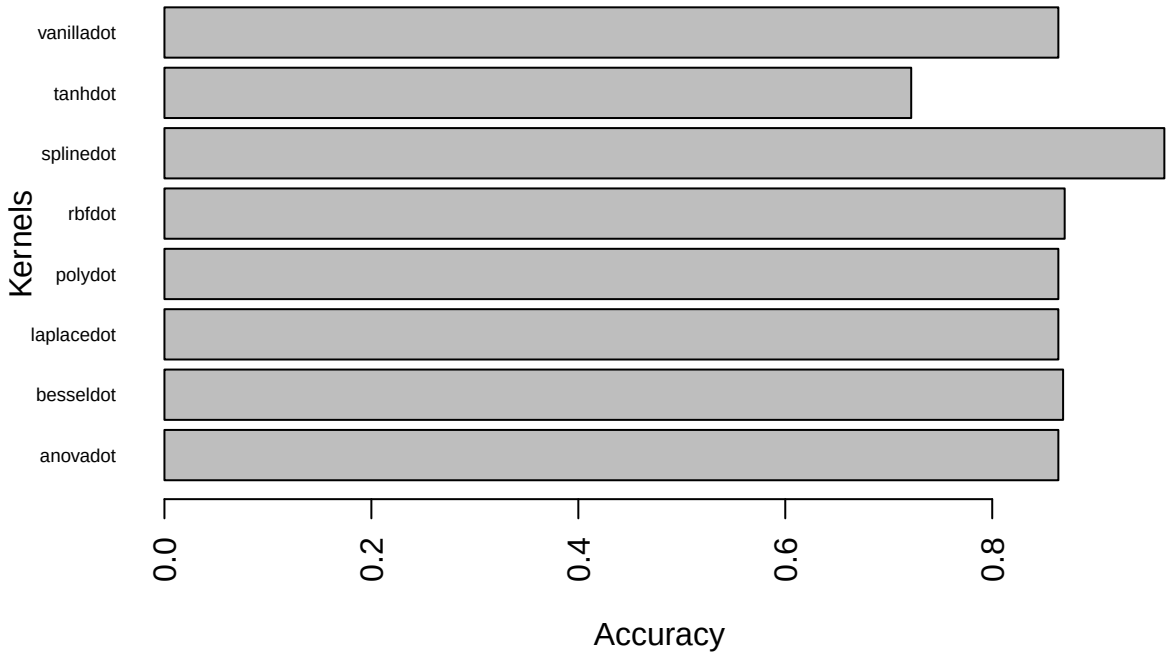
```
myKernels <- c("vanilladot","polydot","besseldot", "laplacedot", "rbfdot", "tanhdot", "anovadot", "splin
accuracy_vector <- vector("numeric")

set.seed(42)
for(i in 1:length(myKernels)){
  model <-  ksvm(as.matrix(df[,1:10]),
```

```
                  as.factor(df[,11]),
                  type="C-svc",
                  kernel=myKernels[[i]],
                  C=1,
                  scaled=T)

  # get predictions
  pred <- predict(model,df[,1:10])
  # get accuracy
  accuracy <- sum(pred == df[, 11]) / nrow(df)
  # store accuracy of each kernel in accuracy vector
  accuracy_vector <- c(accuracy_vector, accuracy)
}
```

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

Next, we will create a dataframe with the list of kernels and their corresponding accuracies. Once this is done, we can plot our data as bar chart to visualize it.

```
# create dataframe
df_2 <- data.frame(kernels = unlist(myKernels),
                   accuracy_score = unlist(accuracy_vector))

# plot a horizontal barplot
barplot(df_2$accuracy_score ~ df_2$kernels,
        horiz=T,
        las=2,
        cex.names=0.6,
        xlab="Accuracy",
        ylab="Kernels")
```



From the bar chart, we observe that our top performer is the `splinedot` kernel.

Let's also take a closer look at the accuracy values from the dataframe:

```
# view full dataframe
df_2
```

```
##      kernels accuracy_score
## 1 vanilladot      0.8639144
## 2    polydot      0.8639144
## 3  besseldot      0.8685015
## 4 laplacedot      0.8639144
## 5     rbfdot      0.8700306
## 6    tanhdot      0.7217125
## 7   anovadot      0.8639144
## 8  splinedot      0.9663609
```

The `splinedot` kernel has an incredibly high accuracy of 0.966, followed by the `rbfdot` kernel at 0.870. Although the `splinedot` model is our top performing model, its accuracy is abnormally high, and it stands to reason that there may be a possibility of over-fitting for that model.

Question 2.2 Part 3

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

For this question, we will be using the `kknn` function in the `kknn` package to classify the credit card dataset.

We will want to find the optimal value of k, which represents the k number of neighbouring data points that will be used to classify a new data point.

Just like in Q2.2.1, to find this optimal value of k, we will loop over a range of values from 1 to 50 and create a KNN model for each k value.

For each value of k, we calculate the corresponding model accuracy, which is the ratio of the number of *correct* predictions to the total number of predictions. We will use accuracy as the metric to determine the best classifier and thus the best value of k.

At the end of the loop, store the accuracy of the model in a vector called `accuracy_vector`.

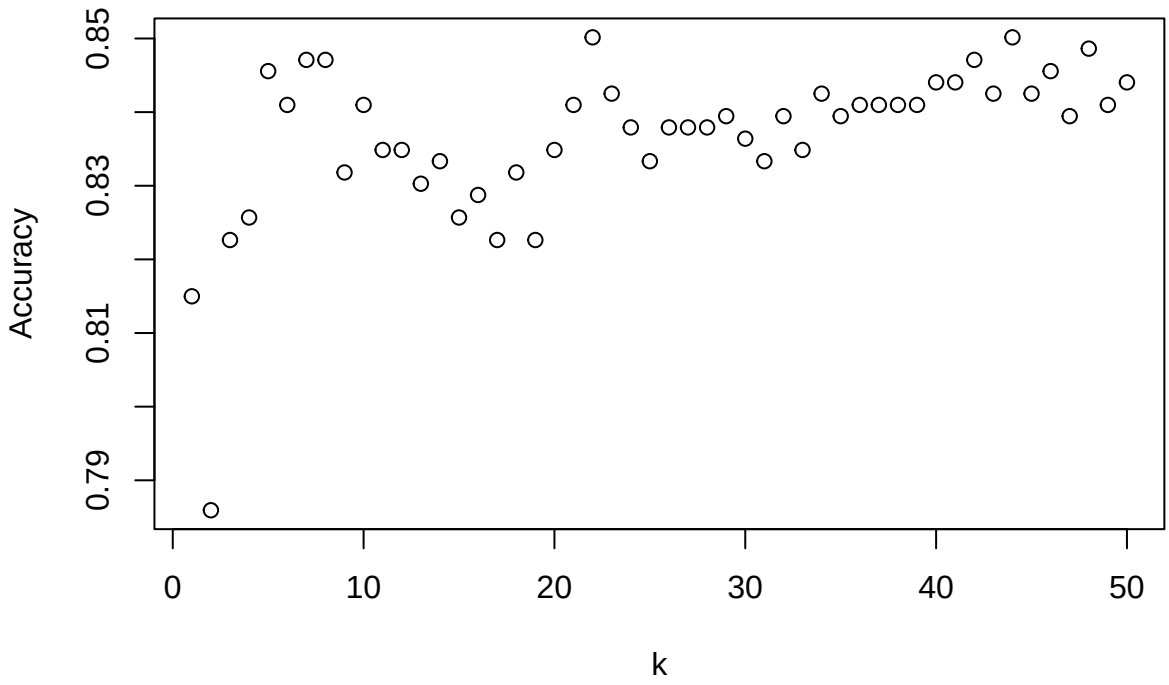
However, before we do anything, we need to initialize an empty vector the same length as the number of rows of the dataset in order to store each prediction generated by the KNN model. [source]

Because kknn returns predictions as a continuous variable instead of purely 1s and 0s, round each output to 1/0 and store in the prediction vector we made earlier

```
pred <- vector(length=nrow(df)) # initialize empty vector
accuracy_vector <- vector("numeric")

# loop over values of k
# build knn model for each value of k
set.seed(42)
for (K in 1:50) {
  for (i in 1:nrow(df)) {
    model <- kknn(formula=df[-i,11]~.,
                  df[-i,1:10], #train (without ith data point)
                  df[i,1:10],  #test (with ith data point)
                  k=K,
                  kernel="rectangular",
                  scale=T)

    # fill pred vector with predictions
    pred[i] <- round(predict(model)) # round pred to 1 or 0
    # get accuracy
    accuracy <- sum(pred == df[,11]) / nrow(df)
  }

  # store accuracy for each k value in accuracy vector
  accuracy_vector <- c(accuracy_vector, accuracy)
}
```

Then, plot the values in `accuracy_vector` against the values of k.

```
# plotting accuracy against values of k
plot(accuracy_vector,
     xlab="k",
     ylab="Accuracy")
```



Use the `max()` function to find the highest accuracy in the vector.

```
# highest accuracy
max(accuracy_vector)
```

```
## [1] 0.8501529
```

Then use the `which()` function to find the index of the highest accuracy.

```
# best value of k
which(accuracy_vector == max(accuracy_vector))
```

```
## [1] 22 44
```

Thus we can suggest a value of k=22 or 44 to be optimal, because it results in the highest accuracy of 0.8501529.

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)

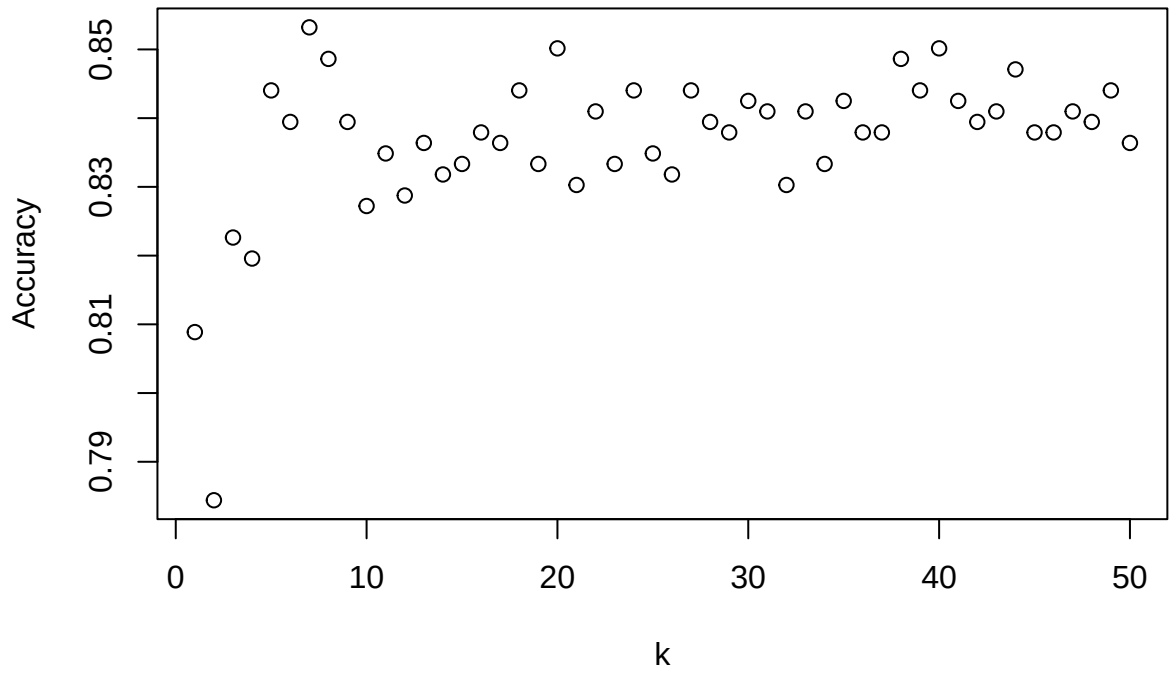
For this question, we will be training KNN models with k-fold cross validation using the `cv.kknn` function from the `kknn` package. We will use a 10-fold cross validation.

Using the same method as the previous question, we will simply loop through 50 values of k, calculate the accuracy, store these accuracy values in our accuracy vector, then use the highest accuracy to select the optimal value of k.

```r
# k-fold cross validation with cv.kknn

accuracy_vector <- vector("numeric")

set.seed(42)
for (K in 1:50) {
  kmodel3 <- cv.kknn(formula=R1~.,
                     df,
                     kcv=10, # k-fold cross validation
                     k=K,
                     kernel="rectangular",
                     scale=T)
  kmodel3 <- data.frame(kmodel3)
  kmodelpred2 <- kmodel3[,2] # preds in 2nd col
  rpred2 <- round(kmodelpred2) # round to 1 or 0
  accuracy <- sum(rpred2 == df[,11]) / nrow(df)
  accuracy_vector <- c(accuracy_vector, accuracy)
}
```

Then, plot the values in `accuracy_vector` against the values of k

```r
# plot accuracy for each value of k
plot(accuracy_vector,
     xlab="k",
     ylab="Accuracy")
```



Use the `max()` function to find the highest accuracy in the vector.

```
# max accuracy
max(accuracy_vector)
```

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

Then use the `which()` function to find the index of the highest accuracy.

```
# best value of k
which(accuracy_vector == max(accuracy_vector))
```

```
## [1] 7
```

Thus we can suggest a value of k=7 to be optimal, because it results in the highest accuracy of 0.853211.

I find it weird that the accuracy of the KNN model

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

For this questions, we will use the `splitTools` package to split our data into train, validation and test datasets. We will use 60% of the data for training, 20% for validation and 20% for testing.

```
# splitting data with splitTools
# https://cran.r-project.org/web/packages/splitTools/vignettes/splitTools.html
library(splitTools)

# split data into partitions
set.seed(42)
split_data <- partition(df$R1, p = c(train = 0.6, valid = 0.2, test = 0.2))

df_train <- df[split_data$train, ]
df_valid <- df[split_data$valid, ]
df_test <- df[split_data$test, ]
```

Let's train an SVM with the train dataset using the best value of C and the best kernel that we found earlier.

```
# train an svm
set.seed(42)
svm_model <- ksvm(as.matrix(df_train[, 1:10]),
                  as.factor(df_train[, 11]),
                  type="C-svc",
                  kernel="splinedot",
                  C=1,
                  scaled=T)
```

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

Now let's train a KNN model using the best value of k that we found earlier.

```r
# train a knn
set.seed(42)
knn_model <- train.kknn(as.factor(R1)~.,
                        df_train,
                        ks = 7,
                        kernel="rectangular",
                        scale=T)
```

```r
# validate each model
svm_valid_pred <- predict(svm_model, df_valid[,1:10])
knn_valid_pred <- predict(knn_model, df_valid[,1:10])

svm_valid_acc <- sum(svm_valid_pred == df_valid[, 11]) / nrow(df_valid)
knn_valid_acc <- sum(knn_valid_pred == df_valid[, 11]) / nrow(df_valid)
```

Create, and print a vector of the SVM accuracy and KNN accuracy so that we can see the accuracy scores side-by-side.

```r
# compare validation accuracies
valid_acc <- c(svm_valid_acc, knn_valid_acc)
valid_acc
```

```
## [1] 0.7938931 0.8549618
```

The KNN model has the higher accuracy from the validation set. Therefore we will select this model to use for further testing.

Now this model will be evaluated on the test dataset.

```r
# predicting based on test data
test_pred <- predict(knn_model, df_test[,1:10])
test_acc <- sum(test_pred == df_test[, 11]) / nrow(df_test)
test_acc
```

```
## [1] 0.8015267
```

The KNN model is a better model for our data due to its higher performance on the validation set compared to SVM model. When testing the KNN model on the test dataset, our model has an accuracy of 0.802 down from the accuracy of 0.855 from the validation set.

Another interesting thing to note is that even though the `splinedot` kernel gave us a very high accuracy in Q2.2.2, we can see now that when we evaluate it on the validation dataset, it gives us a much lower accuracy, strongly supporting our theory that the high accuracy from earlier was due to over-fitting.