DSO 530: KNN-Classification

Abbass Al Sharif

This tutorial is designed to help you learn how to: (1) Run KNN for classification problems in R, (2) learn how to create a "for loop" in R. We will be using an R package called ISLR which has the datasets used in our text book. In addition, we need to install "class" library to be able to run R functions for KNN.

```
#load ISLR and class package (make sure to install it first)
library(ISLR) ## the book's datasets package
library(class) ## for the KNN methods
```

We will apply the KNN approach to the Caravan data set, which is part of the ISLR library. This data set includes 85 predictors that measure demographic characteristics for 5,822 individuals. The response variable is Purchase, which indicates whether or not a given individual purchases a caravan insurance policy. In this data set, only 6% of people purchased caravan insurance.

```
attach(Caravan)
dim(Caravan)
```

```
## [1] 5822   86
```

```
summary(Purchase)
```

```
##   No  Yes
## 5474  348
```

```
348/5822
```

```
## [1] 0.05977
```

Because the KNN classifier predicts the class of a given test observation by identifying the observations that are nearest to it, the scale of the variables matters. Any variables that are on a large scale will have a much larger effect on the distance between the observations, and hence on the KNN classifier, than variables that are on a small scale.

```
# Check the variance for the first two variables
var(Caravan[,1])
```

```
## [1] 165
```

```
var (Caravan[ ,2])
```

```
## [1] 0.1647
```

We are now going to standarize all the X variables except Y (Purchase). The Purchase variable is in column 86 of our dataset, so let's save it in a seperate variable because the knn() function needs it as a seperate argument.

```r
# save the Purchase column in a seperate variable
purchase = Caravan[,86]

# Standarize the dataset using "scale()" R function
standardized_Caravan=scale(Caravan[,-86])
var(standardized_Caravan[,1])
```

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

```r
var(standardized_Caravan[,2])
```

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

We can see that now that all independent variables (X's) have a mean of 1 and standard deviation of 0. Great, then let's divide our dataset into testing and training data. For now, let's just split it, and make the first 1000 observations to be our test data.

```r
test_index = 1:1000
test_data = standardized_Caravan[test_index,]
test_purchase = purchase[test_index]
```

So our training data will have everything else but the testing data observations.

```r
train_data = standardized_Caravan[-test_index,]
train_purchase = purchase[-test_index]
```

Rememeber that we are trying to come up with a model to predict whether someone will purchase or not. We will use the knn() function to do so, and we will focus on 4 of its agruments that we need to specify. The first argument is a data frame that contains the training data set (remember that we don't have the Y here), the second argument is a data frame that contains the testing data set (again no Y variable), the third argument is the train_purchase column (Y) that we save earlier, and the fourth argument is the k (how many neighbors). Let's start with k = 1. knn() function returns a vector of predicted Y's.

```r
predicted_purchase = knn(train_data,test_data,train_purchase,k=1)
head(predicted_purchase)
```

```
## [1] No No No No No No
## Levels: No Yes
```

Now let's evaulate the model we trained and see how much is our misclassification error rate.

```r
mean(test_purchase != predicted_purchase)
```

```
## [1] 0.119
```

You might have a number that is slightly different from the one I got here as my misclassification error. This is because the knn() function has a random component in it, and so everytime you run, you might get a different answer. To overcome this issue, you can set the random number generator seed for a constant that you would use everytime you run knn on this data. SO,

```r
set.seed(1)
predicted_purchase = knn(train_data,test_data,train_purchase,k=1)
mean(test_purchase != predicted_purchase)
```

```
## [1] 0.118
```

Look what happens when we change the see to 3 and then to 1.

```r
set.seed(3)
predicted_purchase = knn(train_data,test_data,train_purchase,k=1)
mean(test_purchase != predicted_purchase)
```

```
## [1] 0.116
```

```r
set.seed(1)
predicted_purchase = knn(train_data,test_data,train_purchase,k=1)
mean(test_purchase != predicted_purchase)
```

```
## [1] 0.118
```

We got the same misclassification error when we had the seed set to 1. You can choose whatever seed you want, but you have to stick to it.

Okay, now remember that k = 1. Let's change k to be equal to 4, and see if that would minimize our misclassification error.

```r
set.seed(1)
predicted_purchase = knn(train_data,test_data,train_purchase,k=4)
mean(test_purchase != predicted_purchase)
```

```
## [1] 0.072
```

It actually did!! Lovely, but can we do better? Should we manually change k and see which k gives us the minimal misclassification rate? NO! we have computers, so let's automate the process with a for() loop. A loop in R repeats the same command as much as you specify. For example, if we want to check for k =1 up to 100, then we have to write 3 x 100 lines of code, but with a for loop, you just need 4 lines of code, and you can repeat those 3 lines up to as many as you want.

```r
  predicted_purchase = NULL
  error_rate = NULL

for(i in 1:30){
  set.seed(1)
  predicted_purchase = knn(train_data,test_data,train_purchase,k=i)
  error_rate[i] = mean(test_purchase != predicted_purchase)
}
print(error_rate)
```

```
##  [1] 0.118 0.109 0.074 0.072 0.066 0.063 0.062 0.062 0.058 0.058 0.059
## [12] 0.058 0.059 0.059 0.059 0.059 0.059 0.059 0.059 0.059 0.059 0.059
## [23] 0.059 0.059 0.059 0.059 0.059 0.059 0.059 0.059
```

```
### find the minimum error rate
min_error_rate = min(error_rate)
print(min_error_rate)
```

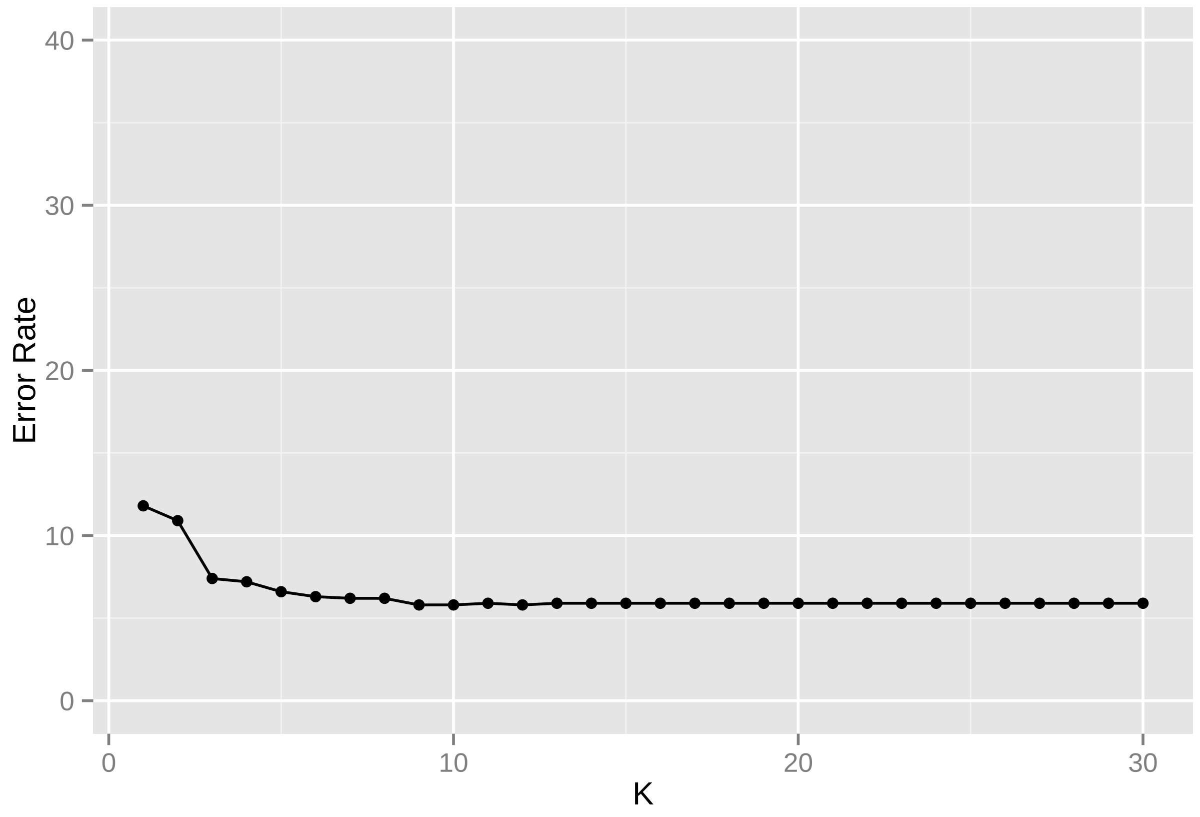
```
## [1] 0.058
```

```
### get the index of that error rate, which is the k
K = which(error_rate == min_error_rate)
print(K)
```

```
## [1]  9 10 12
```

Let's plot!

```
library(ggplot2)
qplot(1:30, error_rate*100, xlab = "K",
      ylab = "Error Rate",
      ylim = c(0,40), geom=c("point", "line"))
```


Error Rate
K