K Nearest Neighbours (KNN)

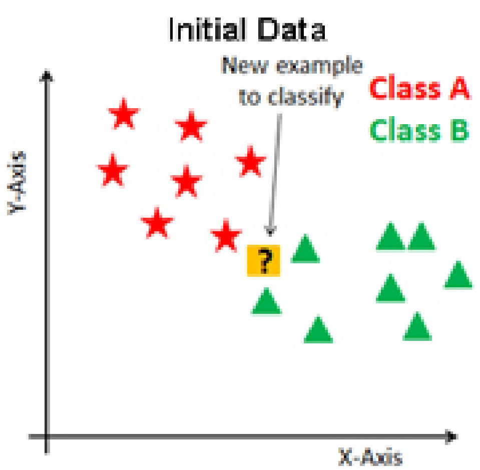
Short Intro

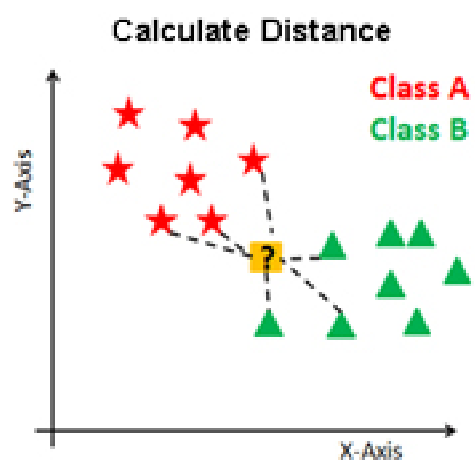
KNN (K Nearest Neighbours) is a non-parametric, lazy learning classification algorithm. Non-parametric means it does not make any assumptions on the underlying data distribution. Therefore, KNN should be one of the first choices for a classification problem when there is little or no prior knowledge about the data distribution. Lazy algorithm (as opposed to an eager algorithm) means it does not use the training data points to do any generalization. In other words, there is no explicit training phase.

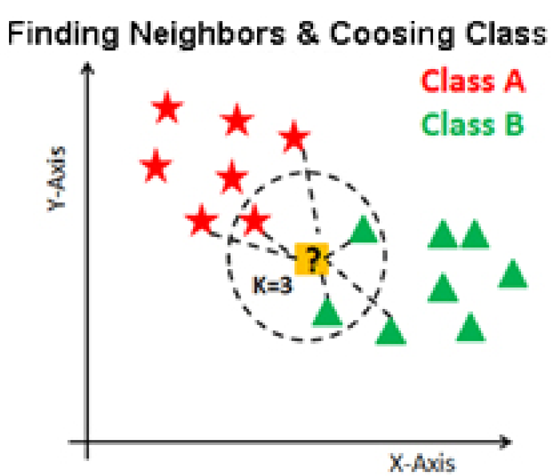
In KNN, a given data point is classified based on the class of the nearest k neighbors. k is usually an odd number in case of a binary classification. In order to find the closest neighors, we calculate feature similarity distance (Euclidean, Manhattan, etc.).

The steps are the following [1]:

1. Calculate distance between the data points
2. Find closest neighbors
3. Choose a class of the majority of neighbors







KNN can be used for classification: the output is a class membership (predicts a class - a discrete value).

An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors. It can also be used for regression: output is the value for the object (predicts continuous values). This value is the average (or median) of the values of its k nearest neighbors.

Load and prepare data set

We will use the same data set (*Carseats*) as in the previous Lab.

```r
# load ISLR package
library(ISLR)

# print dataset structure
str(Carseats)
```

```
## 'data.frame':    400 obs. of  11 variables:
##  $ Sales      : num  9.5 11.22 10.06 7.4 4.15 ...
##  $ CompPrice  : num  138 111 113 117 141 124 115 136 132 132 ...
##  $ Income     : num  73 48 35 100 64 113 105 81 110 113 ...
##  $ Advertising: num  11 16 10 4 3 13 0 15 0 0 ...
##  $ Population : num  276 260 269 466 340 501 45 425 108 131 ...
##  $ Price      : num  120 83 80 97 128 72 108 120 124 124 ...
##  $ ShelveLoc  : Factor w/ 3 levels "Bad","Good","Medium": 1 2 3 3 1 1 3 2 3 3 ...
##  $ Age        : num  42 65 59 55 38 78 71 67 76 76 ...
##  $ Education  : num  17 10 12 14 13 16 15 10 10 17 ...
##  $ Urban      : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 2 1 1 ...
##  $ US         : Factor w/ 2 levels "No","Yes": 2 2 2 2 1 2 1 2 1 2 ...
```

As we did before, we will introduce a categorical (factor) variable *HighSales* to be used as the outcome variable (variable defining the class for each observation). If a sale is greater than the 3rd quartile (9.32), it qualifies as a high sale:

```r
# calculate 3rd quartile
sales.3Q <- quantile(Carseats$Sales, 0.75)

# create a new variable HighSales based on the value of the 3rd quartile
Carseats$HighSales <- ifelse(test = Carseats$Sales > sales.3Q,
                             yes = 'Yes',
                             no = 'No')

# convert HighSales from character to factor
Carseats$HighSales <- as.factor(Carseats$HighSales)
```

We'll remove the *Sales* variable, as we do not need it anymore.

```r
# remove the Sales variable
Carseats <- Carseats[,-1]
```

Standardize numerical attributes

Recall that the *kNN* algorithm primarily works with numerical data. So, if we want to use categorical and/or binary variables, we have to transform them into numerical variables.

kNN is very sensitive to differences in the value range of predictor variables. This is because predictors with a wider range of values (e.g. *Price*) would diminish the influence of variables with significantly narrower range (e.g. *Education*).

Let's check our variables and their value ranges.

```r
# print the summary of the dataset
summary(Carseats)
```

```
##    CompPrice       Income        Advertising       Population
##  Min.   : 77   Min.   : 21.00   Min.   : 0.000   Min.   : 10.0
##  1st Qu.:115   1st Qu.: 42.75   1st Qu.: 0.000   1st Qu.:139.0
##  Median :125   Median : 69.00   Median : 5.000   Median :272.0
##  Mean   :125   Mean   : 68.66   Mean   : 6.635   Mean   :264.8
##  3rd Qu.:135   3rd Qu.: 91.00   3rd Qu.:12.000   3rd Qu.:398.5
##  Max.   :175   Max.   :120.00   Max.   :29.000   Max.   :509.0
##      Price        ShelveLoc        Age          Education    Urban
##  Min.   : 24.0   Bad   : 96   Min.   :25.00   Min.   :10.0   No :118
##  1st Qu.:100.0   Good  : 85   1st Qu.:39.75   1st Qu.:12.0   Yes:282
##  Median :117.0   Medium:219   Median :54.50   Median :14.0
##  Mean   :115.8                Mean   :53.32   Mean   :13.9
##  3rd Qu.:131.0                3rd Qu.:66.00   3rd Qu.:16.0
##  Max.   :191.0                Max.   :80.00   Max.   :18.0
##    US       HighSales
##  No :142   No :301
##  Yes:258   Yes: 99
##
##
##
##
```

Value intervals differ for all variables. We should, obviously, rescale our numerical variables.

Rescaling can be generally done in two ways:

- *Normalization* - reducing variable values to a common value range, typically [0,1]; this is often done using the formula:

$$Z = \frac{X - min(X)}{max(X) - min(X)}$$

- *Standardization* - rescaling variables so that their *mean = 0* and *SD = 1*. For the variable X that is normally distributed, this is done by computing:

$$Z = \frac{X - mean(X)}{SD(X)}$$

  If the variable X is not normally distributed, standardization is typically done using median and inter-quartile range (IQR):

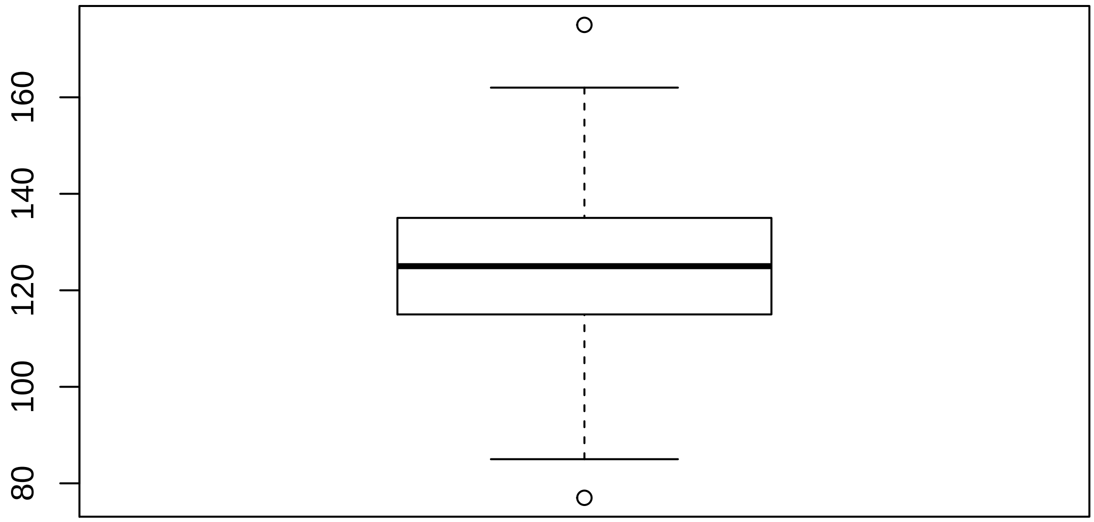
$$Z = \frac{X - median(X)}{IQR(X)}$$

  , where $IQR(X) = Q3(X) - Q1(x)$.

Normalization should be avoided if (numerical) variables have outliers; standardization should be used instead. In the absence of outliers, either of the two can be used.

Let's check the presence of outliers in the *CompPrice* variable.

```r
# plot the boxplot for the CompPrice variable
boxplot(Carseats[,1])
```




```r
# print the number of outliers in the CompPrice variable
length(boxplot.stats(Carseats[,1])$out)
```

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

Let's do the same for all numeric variables.

```r
# filter all numerica variables
numeric.vars <- c(1:5,7,8) # indices of numeric columns

# apply the function for returning the number of outliers for all numeric variables
apply(X = Carseats[,numeric.vars], # select numeric columns
      MARGIN = 2, # apply the function to columns
      FUN = function(x) {
        length(boxplot.stats(x)$out)
      }) # the function to be applied to each column
```

```
##   CompPrice      Income Advertising  Population       Price         Age
##           2           0           0           0           5           0
##   Education
##           0
```

Only 2 variables (*CompPrice*, *Price*) have just a few outliers. Hence, either of the scaling methods can be used.

We will rescale our numerical variables by standardizing them (typical approach). To determine how to standardize the variables, we need to check their distribution (if they follow Normal distribution or not).

We will use *Shapiro-Wilk test* to check for normality. The *null hypothesis* of this test is that a sample comes from a normally distributed population; if the test is not significant (p>0.05), we can assume that the null hypothesis holds.

```r
# apply the test to each numerical column (variable)
apply(X = Carseats[,numeric.vars],
      MARGIN = 2,
      FUN = shapiro.test)
```

```
## $CompPrice
```

```
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.99843, p-value = 0.9772
##
##
## $Income
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.9611, p-value = 8.396e-09
##
##
## $Advertising
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.87354, p-value < 2.2e-16
##
##
## $Population
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.95201, p-value = 4.081e-10
##
##
## $Price
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.99592, p-value = 0.3902
##
##
## $Age
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.95672, p-value = 1.865e-09
##
##
## $Education
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.9242, p-value = 2.427e-13
```

Only *CompPrice* and *Price* are normally distributed. So, we will standardize *Price* and *CompPrice* using mean and SD, and for other variables, we'll use median and IQR.

To do the scaling, we will use the *scale* function (from the base package).

```
# get the documentation for the scale function
?scale
```

We'll start by rescaling variables that are not normally distributed.

```
# select not-normally distributed numerical columns (variables)
carseats.st <- Carseats[, c(2,3,4,7,8)]

# apply the scalling function to each column
carseats.st <- apply(X = carseats.st,
                     MARGIN = 2,
                     FUN = function(x){
                       scale(x, center = median(x), scale = IQR(x))
                     })

# since apply() f. returns a list, convert it to a data frame
carseats.st <- as.data.frame(carseats.st)
```

Then, we'll standardize and add normally distributed ones.

```
# standardize the Price variable (and convert to vector)
carseats.st$Price <- as.vector(scale(x = Carseats$Price, center = TRUE, scale = TRUE))

# standardize the CompPrice variable (and convert to vector)
carseats.st$CompPrice <- as.vector(scale(x = Carseats$CompPrice, center = TRUE, scale = TRUE))
```

Note: the scale() f. returns a matrix with just one column; so, it is effectively a vector and we transform it into a vector using the as.vector() f.

Now, we need to handle binary and categorical variables.

Transform factor (binary and categorical) variables

Transform binary variables into numerical.

```
# transform the Urban variable to integer
carseats.st$Urban <- as.integer(Carseats$Urban)

# transform the US variable to integer
carseats.st$US <- as.integer(Carseats$US)
```

It is often considered more correct to first encode categorical variables as binary dummy variables, and then transform the resulting binary variables into numerical ones. However, for simplicity reasons, and since our categorical variable - *ShelveLoc* - is ordered, we will directly transform it into a numerical variable.

First, let's check the order of *ShelveLoc* levels.

```
# print the levels of the ShelveLoc variable
levels(Carseats$ShelveLoc)
```

```
## [1] "Bad"    "Good"   "Medium"
```

Obviously, the order is not a 'natural' one. So, we need to change the order of levels.

```r
# update the order of levels for the ShelveLoc variable to: "Bad", "Medium", "Good"
Carseats$ShelveLoc <- factor(Carseats$ShelveLoc, levels = c("Bad", "Medium", "Good"))
levels(Carseats$ShelveLoc)
```

```
## [1] "Bad"    "Medium" "Good"
```

Now, we can transform the *ShelveLoc* into a numerical variable.

```r
# convert ShelveLoc into a numeric variable
carseats.st$ShelveLoc <- as.integer(Carseats$ShelveLoc)
```

TASK: Try to create dummy variables for *ShelveLoc* and build a model with these new variables; this page shows how to create dummy variables using the *caret* package.

Finally, add the outcome (class) variable.

```r
# add the outcome variable HighSales
carseats.st$HighSales <- Carseats$HighSales
```

Examine the transformed data set.

```r
# print the structure of the data frame
str(carseats.st)
```

```
## 'data.frame':    400 obs. of  11 variables:
##  $ Income     : num  0.0829 -0.4352 -0.7047 0.6425 -0.1036 ...
##  $ Advertising: num  0.5 0.9167 0.4167 -0.0833 -0.1667 ...
##  $ Population : num  0.0154 -0.0462 -0.0116 0.7476 0.262 ...
##  $ Age        : num  -0.476 0.4 0.171 0.019 -0.629 ...
##  $ Education  : num  0.75 -1 -0.5 0 -0.25 0.5 0.25 -1 -1 0.75 ...
##  $ Price      : num  0.178 -1.385 -1.512 -0.794 0.515 ...
##  $ CompPrice  : num  0.849 -0.911 -0.781 -0.52 1.045 ...
##  $ Urban      : int  2 2 2 2 2 1 2 2 1 1 ...
##  $ US         : int  2 2 2 2 1 2 1 2 1 2 ...
##  $ ShelveLoc  : int  1 3 2 2 1 1 2 3 2 2 ...
##  $ HighSales  : Factor w/ 2 levels "No","Yes": 2 2 2 1 1 2 1 2 1 1 ...
```

```r
# print the summary of the data frame
summary(carseats.st)
```

```
##      Income           Advertising        Population
##  Min.   :-0.994819   Min.   :-0.4167   Min.   :-1.00963
##  1st Qu.:-0.544041   1st Qu.:-0.4167   1st Qu.:-0.51252
##  Median : 0.000000   Median : 0.0000   Median : 0.00000
##  Mean   :-0.007098   Mean   : 0.1363   Mean   :-0.02759
##  3rd Qu.: 0.455958   3rd Qu.: 0.5833   3rd Qu.: 0.48748
##  Max.   : 1.056995   Max.   : 2.0000   Max.   : 0.91329
##       Age             Education          Price            CompPrice
##  Min.   :-1.12381   Min.   :-1.000   Min.   :-3.87702   Min.   :-3.12856
##  1st Qu.:-0.56190   1st Qu.:-0.500   1st Qu.:-0.66711   1st Qu.:-0.65049
##  Median : 0.00000   Median : 0.000   Median : 0.05089   Median : 0.00163
##  Mean   :-0.04486   Mean   :-0.025   Mean   : 0.00000   Mean   : 0.00000
##  3rd Qu.: 0.43810   3rd Qu.: 0.500   3rd Qu.: 0.64219   3rd Qu.: 0.65375
##  Max.   : 0.97143   Max.   : 1.000   Max.   : 3.17633   Max.   : 3.26225
##      Urban             US           ShelveLoc      HighSales
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   No :301
##  1st Qu.:1.000   1st Qu.:1.000   1st Qu.:2.000   Yes: 99
##  Median :2.000   Median :2.000   Median :2.000
```

```
##   Mean   :1.705   Mean   :1.645   Mean   :1.972
##   3rd Qu.:2.000   3rd Qu.:2.000   3rd Qu.:2.000
##   Max.   :2.000   Max.   :2.000   Max.   :3.000
```

Now that we have prepared the data, we can proceed to create sets for training and testing.

Create train and test data sets

We'll use the *caret* package for partitioning the dataset into train and test sets.

```r
# load the caret package
library(caret)
```

We'll take 80% of observations for the training set and the rest for the test set.

```r
# set seed
set.seed(1010)

# create train and test sets
train.indices <- createDataPartition(carseats.st$HighSales, p = 0.8, list = FALSE)
train.data <- carseats.st[train.indices,]
test.data <- carseats.st[-train.indices,]
```

Model building

To build a kNN classification model, we will use the *knn* f. from the *class* package.

```r
# load the class package
library(class)

?knn
```

As the knn() function description indicates, we need to provide the function with:

- training data without the class variable,
- test data without the class variable,
- class values for the training set,
- a number of neighbors to consider.

```r
# createa a knn model with k=5
knn.pred <- knn(train = train.data[,-11],
                test = test.data[,-11],
                cl = train.data$HighSales,
                k = 5) # 5 is chosen here just as a random guess
```

The result of the *knn* f. are, in fact, predictions on the test set.

```r
# print several predictions
head(knn.pred)
```

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

To evaluate the results, we'll first create the confusion matrix:

```
# create the confusion matrix
knn.cm <- table(true = test.data$HighSales, predicted = knn.pred)
knn.cm
```

```
##      predicted
## true  No Yes
##   No  56   4
##   Yes  7  12
```

We'll use the function for computing the evaluation metrics.

```
# function for computing evaluation metrix
compute.eval.metrics <- function(cmatrix) {
  TP <- cmatrix[1,1] # true positive
  TN <- cmatrix[2,2] # true negative
  FP <- cmatrix[2,1] # false positive
  FN <- cmatrix[1,2] # false negative
  acc = sum(diag(cmatrix)) / sum(cmatrix)
  precision <- TP / (TP + FP)
  recall <- TP / (TP + FN)
  F1 <- 2*precision*recall / (precision + recall)
  c(accuracy = acc, precision = precision, recall = recall, F1 = F1)
}
```

Compute the evaluation metrics based on the confusion matrix.

```
# compute the evaluation metrics
knn.eval <- compute.eval.metrics(knn.cm)
knn.eval
```

```
##  accuracy precision    recall        F1
## 0.8607595 0.8888889 0.9333333 0.9105691
```

Not bad, but we might be able to do better by choosing another value for k.

We made a guess about the number of neighbors, and might not have made the best guess. Instead of guessing, we'll cross-validate models with several different values for k, and see which one gives the best performance; then, we'll use the test set to evaluate the model that proves to be the best on cross-validation.

For finding the optimal value for k through 10-fold cross-validation, we will use the **caret** package and the **e1071** package (internally called by the *caret* package).

```
# load e1071 library
library(e1071)
```

```
# define cross-validation (cv) parameters; we'll perform 10-fold cross-validation
numFolds = trainControl( method = "cv", number = 10)
```

Then, define the range of k values to examine in the cross-validation; we'll take odd numbers between 3 and 25. Recall that in case of binary classification, it is recommended to choose an odd number for k.

```
# define the range for the k values to examine in the cross-validation
cpGrid = expand.grid(.k = seq(from=3, to = 25, by = 2))
```

Now, train the model through cross-validation.

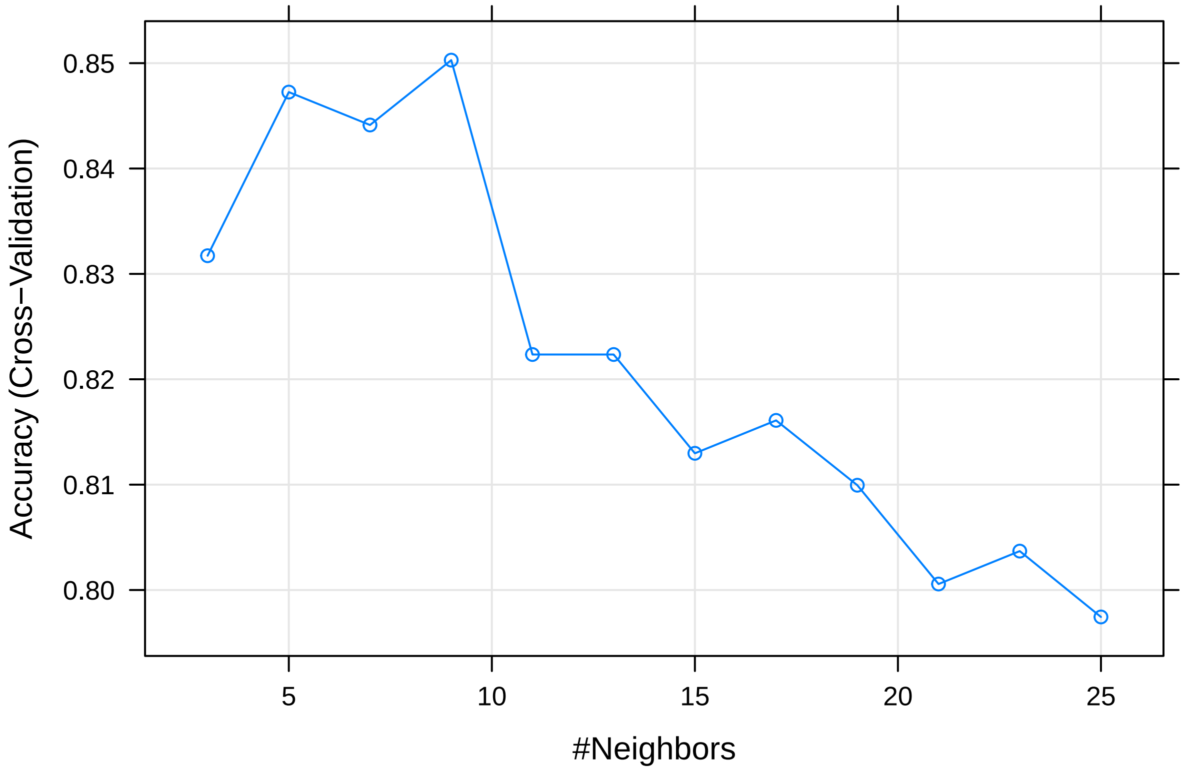
```
# since cross-validation is a probabilistic process, it is advisable to set the seed so that we can rep
set.seed(1010)

# run the cross-validation
```

```r
knn.cv <- train(HighSales ~ .,
                data = train.data,
                method = "knn",
                trControl = numFolds,
                tuneGrid = cpGrid)
knn.cv
```

```
## k-Nearest Neighbors
##
## 321 samples
##  10 predictor
##   2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 289, 288, 289, 289, 289, 289, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    3  0.8317235  0.4967163
##    5  0.8472538  0.5239594
##    7  0.8441288  0.5135494
##    9  0.8502841  0.5166063
##   11  0.8223485  0.4021820
##   13  0.8223485  0.3796718
##   15  0.8129735  0.3257143
##   17  0.8160985  0.3417102
##   19  0.8099432  0.3198316
##   21  0.8005682  0.2752564
##   23  0.8036932  0.2817923
##   25  0.7974432  0.2484590
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 9.
```

We can get a better insight into the cross-validation results by plotting them.

```r
# plot the cross-validation results
plot(knn.cv)
```



k=9 proved to be the best value. Let's build a model with that value for k.

```
# build a new model with k=9
knn.pred2 <- knn(train = train.data[,-11],
                 test = test.data[,-11],
                 cl = train.data$HighSales,
                 k = 9)
```

Create the confusion matrix for the new predictions.

```
# create the confusion matrix
knn.cm2 <- table(true = test.data$HighSales, predicted = knn.pred2)
knn.cm2
```

```
##      predicted
## true  No Yes
##   No  58   2
##   Yes  9  10
```

Compute evaluation measures.

```
# compute the evaluation metrics
knn.eval2 <- compute.eval.metrics(knn.cm2)
knn.eval2
```

```
##  accuracy precision    recall        F1
## 0.8607595 0.8656716 0.9666667 0.9133858
```

This model seems to be better than the previous one, but let's compare the metrics of the two models to check how the new model fares

```
# compare the evaluation metrics for knn1 and knn2 models
data.frame(rbind(knn.eval, knn.eval2), row.names = c("knn 1", "knn 2"))
```

```
##        accuracy precision    recall        F1
## knn 1 0.8607595 0.8888889 0.9333333 0.9105691
## knn 2 0.8607595 0.8656716 0.9666667 0.9133858
```

The first model (*knn1*) is better in term of precision, but weaker with respect to recall.

TASK Create a new model by taking only a subset of variables, for example, those that proved relevant in the DT model and compare the performance with the previously built models.

Potentially useful articles:

- kNN Using caret R package
- Knn classifier implementation in R with caret package

References

[1] Navlani, A. (2018, August 2). KNN Classification using Scikit-learn. Retrieved from https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn