**Tutorial for k-NN**

**Our first k-NN example (Iris data - again!)**

As a first example, we will be working with the Iris Dataset. This data set is a real-world “toy” dataset that is often used to demonstrate concepts in data science. The iris dataset contains information about several flowers selected from three different species of iris: versicolor, setosa, and virginica. The dataset contains the following five pieces of information for 150 flowers:

* The sepal length of the flower.
* The sepal width of the flower.
* The petal length of the flower.
* The petal width of the flower.
* The species of the flower.

**Load and Explore the Data**

library(datasets)

library(ggplot2)

library(gridExtra)

library(class)

library(caret)

We can show the iris data with this command, just type "iris" for show the all data

head(iris)

summary(iris)

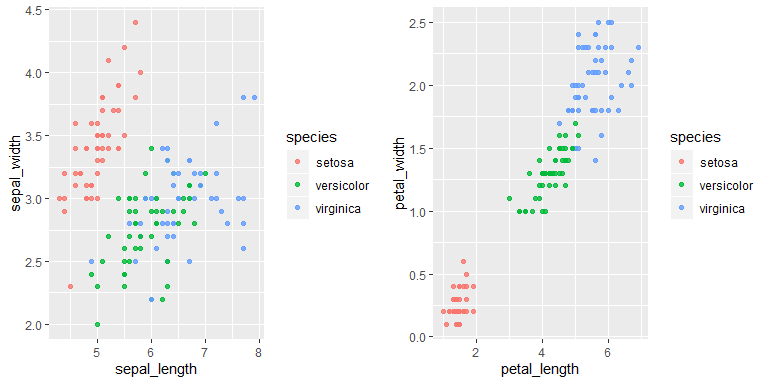
p1 <- ggplot(iris, aes(x=Sepal.Length, y=Sepal.Width, col=Species)) +

geom\_point(alpha=0.8)

p2 <- ggplot(iris, aes(x=Petal.Length, y=Petal.Width, col=Species)) +

geom\_point(alpha=0.8)

grid.arrange(p1, p2, ncol=2)

****

**Build a k-Nearest Neighbours Model**

We will now use the knn function from the class package to create a k-Nearest Neighbours model for the iris dataset.

ind <- sample(1:nrow(iris), nrow(iris)\*0.7)

iris.train <- iris[ind, ]

iris.test <- iris[-ind, ]

We will now use the knn function from the class package to create a k-Nearest Neighbours model for the iris dataset with k=3

knn.fit <- knn(train = iris.train[,1:4], test = iris.test[,1:4], cl = iris.train$Species, k = 3)

error[i] = 1- mean(knn.fit == iris.test$Species)

accuracy <- mean(knn.fit == iris.test$Species)

cat("Training Accuracy: ", accuracy, sep='')

Training Accuracy: 0.9111111

table(iris.test$Species,knn.fit)

knn.fit

setosa versicolor virginica

setosa 11 0 0

versicolor 0 13 1

virginica 0 3 17

nb.test.acc1 <- confusionMatrix(iris.test$Species, knn.fit, mode="everything")

nb.test.acc1

Confusion Matrix and Statistics

Reference

Prediction setosa versicolor virginica

setosa 11 0 0

versicolor 0 13 1

virginica 0 3 17

Overall Statistics

Accuracy : 0.9111

95% CI : (0.7878, 0.9752)

No Information Rate : 0.4

P-Value [Acc > NIR] : 9.959e-13

Kappa : 0.8636

Mcnemar's Test P-Value : NA

Statistics by Class:

Class: setosa Class: versicolor Class: virginica

Sensitivity 1.0000 0.8125 0.9444

Specificity 1.0000 0.9655 0.8889

Pos Pred Value 1.0000 0.9286 0.8500

Neg Pred Value 1.0000 0.9032 0.9600

Precision 1.0000 0.9286 0.8500

Recall 1.0000 0.8125 0.9444

F1 1.0000 0.8667 0.8947

Prevalence 0.2444 0.3556 0.4000

Detection Rate 0.2444 0.2889 0.3778

Detection Prevalence 0.2444 0.3111 0.4444

Balanced Accuracy 1.0000 0.8890 0.9167

################################################

The selection of K=3 in the previous model was somewhat arbitrary. In the figure below, we will plot the KNN model’s training accuracy for several values of K.

error <- c()

valid\_acc <- c()

for (i in 1:19)

{

knn.fit <- knn(train = iris.train[,1:4], test = iris.test[,1:4], cl = iris.train$Species, k = i)

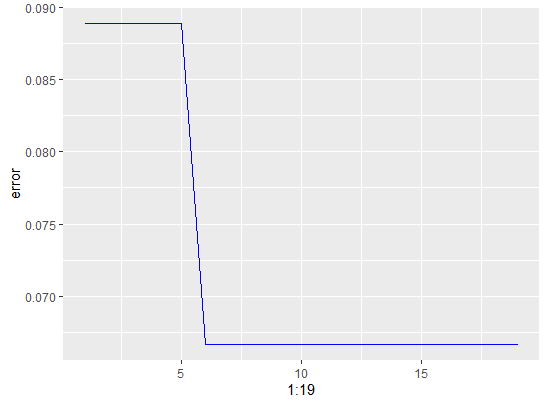
error[i] = 1- mean(knn.fit == iris.test$Species)

valid\_acc <- c(valid\_acc, mean(knn.fit == iris.test$Species))

}

ggplot(data = data.frame(error), aes(x = 1:19, y = error)) +

geom\_line(color = "Blue")



max(valid\_acc)

0.9333333

which.max(valid\_acc)

[1] 6

**Our “biomedical” example using k-NN (Breast Cancer Dataset)**

Routine breast cancer screening allows the disease to be diagnosed and treated prior to it causing noticeable symptoms. The process of early detection involves examining the breast tissue for abnormal lumps or masses. If a lump is found, a fine-needle aspiration biopsy is performed, which uses a hollow needle to extract a small sample of cells from the mass. A clinician then examines the cells under a microscope to determine whether the mass is likely to be malignant or benign. If machine learning could automate the identification of cancerous cells, it would provide considerable benefit to the health system. Automated processes are likely to improve the efficiency of the detection process, allowing physicians to spend less time diagnosing and more time treating the disease. An automated screening system might also provide greater detection accuracy by removing the inherently subjective human component from the process. We will utilize the Breast Cancer Wisconsin (Diagnostic) dataset from the UCI Machine Learning Repository at http://archive.ics.uci.edu/ml. This data was donated by researchers of the University of Wisconsin and includes measurements from digitized images of fine-needle aspirate of a breast mass. The values represent characteristics of the cell nuclei present in the digital image. The breast cancer data includes 569 examples of cancer biopsies, each with 32 features. One feature is an identification number, another is the cancer diagnosis, and 30 are numeric-valued laboratory measurements. The diagnosis is coded as "M" to indicate malignant or "B" to indicate benign. The 30 numeric measurements comprise the mean, standard error, and worst (that is, largest) value for 10 different characteristics of the digitized cell nuclei. These include:

• Radius

• Texture

• Perimeter

• Area

• Smoothness

• Compactness

• Concavity

• Concave points

• Symmetry

• Fractal dimension

Based on these names, all features seem to relate to the shape and size of the cell nuclei. Unless you are an oncologist, you are unlikely to know how each relates to benign or malignant masses. These patterns will be revealed as we continue in the machine learning process.

library(class)

library(caret)

library(readxl) # this is the package for reading xlsx files

wbcd<-read\_excel("C:/R\_codes/wisc\_bc\_data.xlsx")

Using the command str(wbcd), we can confirm that the data is structured with 569 examples and 32 features, as we expected. The first several lines of output are as follows:

str(wbcd)

tibble [569 x 32] (S3: tbl\_df/tbl/data.frame)

$ id : num [1:569] 842302 842517 84300903 84348301 84358402 ...

$ diagnosis : chr [1:569] "M" "M" "M" "M" ...

$ radius\_mean : num [1:569] 18 20.6 19.7 11.4 20.3 ...

$ texture\_mean : num [1:569] 10.4 17.8 21.2 20.4 14.3 ...

$ perimeter\_mean : num [1:569] 122.8 132.9 130 77.6 135.1 ...

$ area\_mean : num [1:569] 1001 1326 1203 386 1297 ...

$ smoothness\_mean : num [1:569] 0.1184 0.0847 0.1096 0.1425 0.1003 ...

$ compactness\_mean : num [1:569] 0.2776 0.0786 0.1599 0.2839 0.1328 ...

The first variable is an integer variable named id. As this is simply a unique identifier (ID) for each patient in the data, it does not provide useful information and we will need to exclude it from the model. As it is located in the first column, we can exclude it by making a copy of the wbcd data frame without column 1:

wbcd <- wbcd[-1]

The next variable, diagnosis, is of particular interest as it is the outcome we hope to predict. This feature indicates whether the example is from a benign or malignant mass. The table() output indicates that 357 masses are benign, while 212 are malignant:

table(wbcd$diagnosis)

B M

357 212

Many R machine learning classifiers require the target feature to be coded as a factor, so we will need to recode the diagnosis variable. We will also take this opportunity to give the "B" and "M" values more informative labels using the labels parameter:

wbcd$diagnosis <- factor(wbcd$diagnosis, levels = c("B", "M"), labels = c("Benign", "Malignant"))

When we look at the prop.table() output, we now find that the values have been labelled Benign and Malignant, with 62.7 percent and 37.3 percent of the masses, respectively:

round(prop.table(table(wbcd$diagnosis)) \* 100, digits = 1)

Benign Malignant

62.7 37.3

The remaining 30 features are all numeric and, as expected, consist of three different of 10 characteristics. For illustrative purposes, we will only take a closer look at three of these features:

summary(wbcd[c("radius\_mean", "area\_mean", "smoothness\_mean")])

Remember, that the distance calculation for k-NN is heavily dependent upon the measurement scale of the input features. Since smoothness ranges from 0.05 to 0.16, while area ranges from 143.5 to 2501.0, the impact of area is going to be much greater than smoothness in the distance calculation. This could potentially cause problems for our classifier, so let's apply normalization to rescale the features to a standard range of values.

To normalize these features, we need to create a normalize() function in R. This function takes a vector x of numeric values, and for each value in x, subtracts the minimum x value and divides by the range of x values. Lastly, the resulting vector is returned. The code for the function is as follows:

normalize <- function(x) {

return ((x - min(x)) / (max(x) - min(x)))

}

We can now apply the normalize() function to the numeric features in our data frame. The lapply() function takes a list and applies a specified function to each list element. As a data frame is a list of equal-length vectors, we can use lapply() to apply normalize() to each feature in the data frame. The final step is to convert the list returned by lapply() to a data frame using the as.data.frame() function. The full process looks like this:

wbcd\_norm <- as.data.frame(lapply(wbcd[2:31], normalize))

We will use the first 469 records for the training dataset and the remaining 100 to simulate new patients.

wbcd\_train <- wbcd\_norm[1:469, ]

wbcd\_test <- wbcd\_norm[470:569, ]

When we constructed our normalized training and test datasets, we excluded the target variable, diagnosis. For training the k-NN model, we will need to store these class labels in factor vectors, split between the training and test datasets:

wbcd\_train\_labels <- wbcd[1:469, 1]

wbcd\_test\_labels <- wbcd[470:569, 1]

This code takes the diagnosis factor in the first column of the wbcd data frame and creates the vectors wbcd\_train\_labels and wbcd\_test\_labels. We will use these for training and evaluating our classifier. As our training data includes 469 instances, we might try k = 21, an odd number roughly equal to the square root of 469.

wbcd\_test\_pred <- knn(train = wbcd\_train, test = wbcd\_test, cl = wbcd\_train\_labels$diagnosis, k = 21)

The next step of the process is to evaluate how well the predicted classes in the wbcd\_test\_pred vector match the actual values in the wbcd\_test\_labels vector. To do this, we can use the CrossTable() function in the gmodels package. We can create a cross tabulation indicating the agreement between the predicted and actual label vectors. Specifying prop.chisq = FALSE will remove the unnecessary chi-square values from the output:

CrossTable(x = wbcd\_test\_labels$diagnosis, y = wbcd\_test\_pred, prop.chisq = FALSE)

Cell Contents

|-------------------------|

| N |

| N / Row Total |

| N / Col Total |

| N / Table Total |

|-------------------------|

Total Observations in Table: 100

| wbcd\_test\_pred

wbcd\_test\_labels$diagnosis | Benign | Malignant | Row Total |

---------------------------|-----------|-----------|-----------|

Benign | 77 | 0 | 77 |

| 1.000 | 0.000 | 0.770 |

| 0.975 | 0.000 | |

| 0.770 | 0.000 | |

---------------------------|-----------|-----------|-----------|

Malignant | 2 | 21 | 23 |

| 0.087 | 0.913 | 0.230 |

| 0.025 | 1.000 | |

| 0.020 | 0.210 | |

---------------------------|-----------|-----------|-----------|

Column Total | 79 | 21 | 100 |

| 0.790 | 0.210 | |

---------------------------|-----------|-----------|-----------|

**Our usual dataset using k-NN (wine data again!)**

The wine data set contains the results of a chemical analysis of wines grown in a specific area of Italy. Three types of wine are represented in the 178 samples, with the results of 13 chemical analyses recorded for each sample. The Type variable has been transformed into a categorical variable.

1. Alcohol
2. Malic Acid
3. Ash
4. Ash Alcalinity
5. Magnesium
6. Total Phenols
7. Flavanoids
8. Nonflavanoid Phenols
9. Proanthocyanins
10. Color Intensity
11. Hue
12. Dilution
13. Proline

This is an example of a pattern recognition problem, where inputs are associated with different classes, and we would like to create a kNN classifier that not only classifies the known wines properly, but can generalize to accurately classify wines that were not used to design the solution. The thirteen neighborhood attributes will act as inputs to our model. kNN is one of the most simplistic machine learning algorithms, and is very useful when it comes to solving classification problems. You need first to install the necessary packages, before calling them via the library command.

wine<-read.csv("C:/R\_codes/winedata.csv") # use your own folder – dataset has been provided before

head(wine)

str(wine)

In the original wine dataset, there are three classes; the samples however have been sorted based on their class. Therefore, in order to create a good quality testing dataset, we need to randomize the order of samples.

# create a vector

v <- 1:178 # as we have 178 samples

# Randomize the order of the vector

v <- sample(v)

# Randomize the order of the data frame

wine\_random <- wine[sample(1:nrow(wine)), ] # why we need to randomize the dataset? Think about

head(wine\_random)

#Let's see the structure of wine data set

str(wine\_random)

#detailed view of the data set

summary(wine\_random)

The summary command helps us understand the data in a better way. It clearly shows different attributes along with min, max, median, and other such statistics. These help us in the coming steps where we might have to scale or normalize the data or features. During this step is where we usually label our input data. Since our current data set is already labeled, we can skip this step for this example problem. Let us visually see how the species are spread. We take help of the famous scatter plot again, but this time we use a package called ggvis. For visualizing, for example, Alcohol and Malic for all 3 classes, we use the following code:

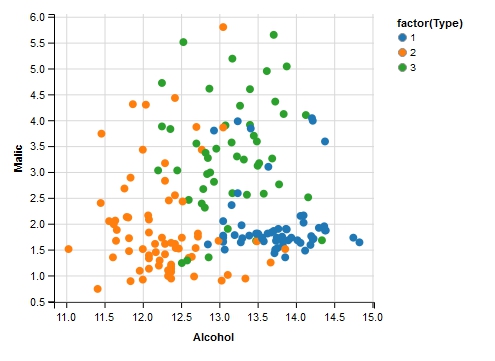
#################################################

library(ggvis)

#plot the species

wine\_random %>% ggvis(~Alcohol, ~Malic, fill = ~factor(Type)) %>%

layer\_points()



The ggvis package is an interactive graphics package in R. It follows a unique way of expressing inputs to generate visualizations. The above code uses the pipe operator, %>%, to pass input data to ggvis and again uses the pipe operator to pass on the output to layer\_points for final plotting. The ~ operator signifies to ggvis that Alcohol, for example, is a variable in the input dataset (wine\_random). Read more about ggvis at <http://ggvis.rstudio.com/ggvis-basics.html>. Try visualizing different pairs as well and see if you can spot any correlation.

The next step is to normalize the data so that all the features are on the same scale. As seen from the data exploration step, the values of all the attributes are more or less in a comparable range. This is necessary because without a common scale for each variable, then it will not be possible for kNN to meaningfully classify the variable of interest. We normalize as per the below code:

normalize <- function(x) {

return ((x - min(x)) / (max(x) - min(x)))

}

norm\_wine <- as.data.frame(lapply(wine\_random[2:14], normalize)) # start from 2, as column one is associated to class (i.e. desired output).

The following code shows a summary of the normalized data frame:

summary(norm\_wine)

summary(norm\_wine$Alcohol)

Now that we have our data normalized, we can divide it into training and test datasets. We will follow the usual two-third one-third rule of splitting the data into two.

#Training and Test Data

trainset <- norm\_wine[1:100, ]

testset <- norm\_wine[101:178, ]

#Labels

trainset\_labels <- wine\_random[1:100, 1]

testset\_labels <- wine\_random[101:178, 1]

Once we have the data ready in our training and test data sets, we can proceed to the next step and learn from the data using KNN. The KNN implementation in R is present in the class library. The KNN function takes the following inputs:

• train: The data frame containing the training data.

• test: The data frame containing the test data.

• class: A vector containing the class labels. Also called as the factor vector.

• k: The value of k-nearest neighbours.

For the current case, let us assume the value of k to be 15. Odd numbers are usually good at breaking ties. KNN is executed as:

library(class)

knn\_prediction <- knn(train = trainset, test = testset,cl = trainset\_labels, k=15)

library(gmodels)

CrossTable(x=testset\_labels, y=knn\_prediction,prop.chisq=FALSE)

Cell Contents

|-------------------------|

| N |

| N / Row Total |

| N / Col Total |

| N / Table Total |

|-------------------------|

Total Observations in Table: 78

| knn\_prediction

testset\_labels | 1 | 2 | 3 | Row Total |

---------------|-----------|-----------|-----------|-----------|

1 | 26 | 1 | 0 | 27 |

| 0.963 | 0.037 | 0.000 | 0.346 |

| 0.929 | 0.036 | 0.000 | |

| 0.333 | 0.013 | 0.000 | |

---------------|-----------|-----------|-----------|-----------|

2 | 2 | 27 | 2 | 31 |

| 0.065 | 0.871 | 0.065 | 0.397 |

| 0.071 | 0.964 | 0.091 | |

| 0.026 | 0.346 | 0.026 | |

---------------|-----------|-----------|-----------|-----------|

3 | 0 | 0 | 20 | 20 |

| 0.000 | 0.000 | 1.000 | 0.256 |

| 0.000 | 0.000 | 0.909 | |

| 0.000 | 0.000 | 0.256 | |

---------------|-----------|-----------|-----------|-----------|

Column Total | 28 | 28 | 22 | 78 |

| 0.359 | 0.359 | 0.282 | |

---------------|-----------|-----------|-----------|-----------|

KNN is a simple yet powerful algorithm which makes no assumptions about the underlying data distribution and hence can be used in cases where relationships between features and classes are complex or difficult to understand.

**Alternative case for wine dataset (see the CV issue)**

The R programming machine learning caret package (Classification and Regression Training) holds tons of functions that helps to build predictive models. It holds tools for data splitting, pre-processing, feature selection, tuning and [supervised – unsupervised learning](https://dataaspirant.com/2014/09/19/supervised-and-unsupervised-learning/) algorithms. Caret package provides us direct access to various functions for training our model with various [machine learning algorithms](https://dataaspirant.com/2016/09/24/classification-clustering-alogrithms/) like K-NN, SVM, decision tree, [linear regression](https://dataaspirant.com/2014/10/02/linear-regression/), etc. For implementing K-NN in r, we only need to import caret package.

library(caret)

dataurl <- "https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data"

download.file(url = dataurl, destfile = "wine.data") # here we download data from a website

wine\_df <- read.csv("wine.data", header = FALSE)

In dataurl vector, we are putting URL of our wine data. Using download.file() method, we can download the data file from URL. For downloading, URL of data and destination file name should be mentioned as the parameters of download.file() method. Here, our destfile parameter is set with value “wine.data”. For importing data into an R data frame, we can use read.csv() method with parameters as a file name and whether our dataset consists o the 1st row with a header or not. If a header row exists then, the header should be set TRUE else header should set to FALSE. For checking the structure of data frame we can call the function **str** over wine\_df

str(wine\_df)

'data.frame': 178 obs. of  14 variables:

$ V1 : int  1 1 1 1 1 1 1 1 1 1 ...

$ V2 : num  14.2 13.2 13.2 14.4 13.2 ...

$ V3 : num  1.71 1.78 2.36 1.95 2.59 1.76 1.87 2.15 1.64 1.35 ...

$ V4 : num  2.43 2.14 2.67 2.5 2.87 2.45 2.45 2.61 2.17 2.27 ...

$ V5 : num  15.6 11.2 18.6 16.8 21 15.2 14.6 17.6 14 16 ...

$ V6 : int  127 100 101 113 118 112 96 121 97 98 ...

$ V7 : num  2.8 2.65 2.8 3.85 2.8 3.27 2.5 2.6 2.8 2.98 ...

$ V8 : num  3.06 2.76 3.24 3.49 2.69 3.39 2.52 2.51 2.98 3.15 ...

$ V9 : num  0.28 0.26 0.3 0.24 0.39 0.34 0.3 0.31 0.29 0.22 ...

$ V10: num  2.29 1.28 2.81 2.18 1.82 1.97 1.98 1.25 1.98 1.85 ...

$ V11: num  5.64 4.38 5.68 7.8 4.32 6.75 5.25 5.05 5.2 7.22 ...

$ V12: num  1.04 1.05 1.03 0.86 1.04 1.05 1.02 1.06 1.08 1.01 ...

$ V13: num  3.92 3.4 3.17 3.45 2.93 2.85 3.58 3.58 2.85 3.55 ...

$ V14: int  1065 1050 1185 1480 735 1450 1290 1295 1045 1045 ...

It shows that our data consists of **178** observations and **14** columns. Value ranges of all attributes from V2-V14 are varying, so we will have to standardize the data before training our classifier

**Data Slicing**

Data slicing is a step to split data into train and test set. Training data set can be used specifically for our model building. Test dataset should not be mixed up while building model.

#set.seed(3333)

intrain <- createDataPartition(y = wine\_df$V1, p= 0.7, list = FALSE)

training <- wine\_df[intrain,]

testing <- wine\_df[-intrain,]

Theset.seed() method is used to make the work replicable. The caret package provides a method createDataPartition() for partitioning our data into train and test set. We are passing 3 parameters. The “y” parameter takes the value of variable according to which data needs to be partitioned. In our case, target variable is at V1, so we are passing wine\_df$V1 (wine data frame’s V1 column). The “p” parameter holds a decimal value in the range of 0-1. It’s to show that percentage of the split. We are using p=0.7. It means that data split should be done in 70:30 ratio. The “list” parameter is for whether to return a list or matrix. We are passing FALSE for not returning a list. The createDataPartition() method is returning a matrix “intrain” with record’s indices. By passing values of intrain, we are splitting training data and testing data.

dim(training)

dim(testing)

[1] 125 14

[1] 53 14

**Preprocessing & Training**

Pre-processing is all about correcting the problems in data before building a machine learning model using that data. Problems can be of many types like missing values, attributes with a different range, etc. To check whether our data contains missing values or not, we can use **anyNA()** method. Here, NA means Not Available.

anyNA(wine\_df)

[1] FALSE

Since it’s returning **FALSE**, it means we don’t have any missing values

**Wine Dataset summarized details**

For checking the summarized details of our data, we can use summary**()** method. It will give us a basic idea about our dataset’s attributes range.

summary(wine\_df)

  V1              V2              V3              V4              V5              V6

Min.   :1.000   Min.   :11.03   Min.   :0.740   Min.   :1.360   Min.   :10.60   Min.   : 70.00

1st Qu.:1.000   1st Qu.:12.36   1st Qu.:1.603   1st Qu.:2.210   1st Qu.:17.20   1st Qu.: 88.00

Median :2.000   Median :13.05   Median :1.865   Median :2.360   Median :19.50   Median : 98.00

Mean   :1.938   Mean   :13.00   Mean   :2.336   Mean   :2.367   Mean   :19.49   Mean   : 99.74

3rd Qu.:3.000   3rd Qu.:13.68   3rd Qu.:3.083   3rd Qu.:2.558   3rd Qu.:21.50   3rd Qu.:107.00

Max.   :3.000   Max.   :14.83   Max.   :5.800   Max.   :3.230   Max.   :30.00   Max.   :162.00

       V7              V8              V9              V10             V11              V12

Min.   :0.980   Min.   :0.340   Min.   :0.1300   Min.   :0.410   Min.   : 1.280   Min.   :0.4800

1st Qu.:1.742   1st Qu.:1.205   1st Qu.:0.2700   1st Qu.:1.250   1st Qu.: 3.220   1st Qu.:0.7825

Median :2.355   Median :2.135   Median :0.3400   Median :1.555   Median : 4.690   Median :0.9650

Mean   :2.295   Mean   :2.029   Mean   :0.3619   Mean   :1.591   Mean   : 5.058   Mean   :0.9574

3rd Qu.:2.800   3rd Qu.:2.875   3rd Qu.:0.4375   3rd Qu.:1.950   3rd Qu.: 6.200   3rd Qu.:1.1200

Max.   :3.880   Max.   :5.080   Max.   :0.6600   Max.   :3.580   Max.   :13.000   Max.   :1.7100

      V13             V14

Min.   :1.270   Min.   : 278.0

1st Qu.:1.938   1st Qu.: 500.5

Median :2.780   Median : 673.5

Mean   :2.612   Mean   : 746.9

3rd Qu.:3.170   3rd Qu.: 985.0

Max.   :4.000   Max.   :1680.0

From above summary statistics, it shows us that all the attributes have a different range. So, we need to standardize our data. We can also standardize data using caret’s preProcess**()** method. Our target variable consists of 3 values 1, 2, 3. These should be considered as categorical variables. To convert these to categorical variables, we can convert them to factors.

training[["V1"]] = factor(training[["V1"]])

The above line of code will convert training data frame’s “V1” column to factor variable.

**Training the K-nn model**

Caret package provides the train() method for training our data for various algorithms. Before applying the train() method, we need to use the trainControl() method. It controls the computational details of the train() method.

trctrl <- trainControl(method = "repeatedcv", number = 10, repeats = 3)

#set.seed(3333)

knn\_fit <- train(V1 ~., data = training, method = "knn",

trControl=trctrl,

preProcess = c("center", "scale"),

tuneLength = 10)

We are setting 3 parameters of **trainControl()** method. The “method” parameter holds the details about re-sampling method. We can set “method” with many values like “boot”, “boot632”, “cv”, “repeatedcv”, “LOOCV”, “LGOCV” etc. Here we are using the repeatedcv, i.e. repeated cross-validation. The “number” parameter holds the number of re-sampling iterations. The “repeats” parameter contains the complete sets of folds to compute for our repeated cross-validation. We are using setting number =10 and repeats =3. This trainControl() methods returns a list. We are going to pass this on our train() method.  
Before training our knn classifier, we set the set.seed().

For training the knn classifier, train() method should be passed with “method” parameter as “knn”. We are passing our target variable V1. The V1~. denotes a formula for using all attributes in our classifier and V1 as the target variable. The “trControl” parameter should be passed with results from our trianControl() method. The “preProcess” parameter is for pre-processing our training data.

As discussed earlier for our data, pre-processing is a mandatory task. We are passing 2 values in our “preProcess” parameter “center” & “scale”. These two help for centering and scaling the data. The “tuneLength” parameter holds an integer value. This is for tuning our algorithm.

**Trained Knn model result**

You can check the result of our train() method. We are saving its results in a knn\_fit variable.

knn\_fit #knn classifier

k-Nearest Neighbors

125 samples

13 predictor

3 classes: '1', '2', '3'

Pre-processing: centered (13), scaled (13)

Resampling: Cross-Validated (10 fold, repeated 3 times)

Summary of sample sizes: 114, 112, 113, 112, 114, 112, ...

Resampling results across tuning parameters:

k Accuracy Kappa

5 0.9543790 0.9317929

7 0.9404512 0.9109657

9 0.9379260 0.9073292

11 0.9374598 0.9067419

13 0.9396270 0.9099077

15 0.9482129 0.9225977

17 0.9479604 0.9222815

19 0.9516706 0.9276711

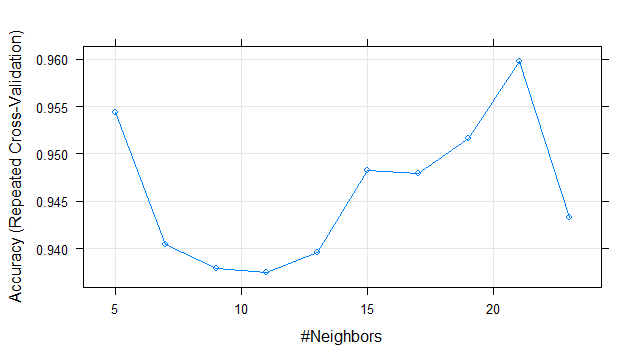
21 0.9597597 0.9401666

23 0.9432678 0.9152521

Accuracy was used to select the optimal model using the largest value.

The final value used for the model was k = 21.

Results are showing Accuracy and Kappa metrics result for different k value. From the results, method, automatically selects best k-value. Here, our training model is choosing k = 21 for its final value. We can see the above variations, also if we plot these results by any graphical programme.



**Test Set Prediction**

Now, our model is trained with K value as 21. We are ready to predict classes for our test set. We can use predict() method.

test\_pred <- predict(knn\_fit, newdata = testing)

test\_pred

[1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

[40] 3 3 3 3 3 3 3 3 3 3 3 3 3 3

Levels: 1 2 3

caret package provides predict() method for predicting results. We are passing 2 arguments. The first parameter is our trained model and second parameter “newdata” holds our testing data frame. The predict() method returns a list, we are saving it in a test\_pred variable.

Using confusion matrix, we can print statistics of our results. It shows that our model accuracy for test set is 96.23%.

desired\_test = factor(testing$V1) # why I am doing this? What will happen without such change?

confusionMatrix(test\_pred, desired\_test )

Confusion Matrix and Statistics

Reference

Prediction 1 2 3

1 15 0 0

2 0 22 0

3 0 2 14

Overall Statistics

Accuracy : 0.9623

95% CI : (0.8702, 0.9954)

No Information Rate : 0.4528

P-Value [Acc > NIR] : 1.208e-15

Kappa : 0.9421

Mcnemar's Test P-Value : NA

Statistics by Class:

Class: 1 Class: 2 Class: 3

Sensitivity 1.000 0.9167 1.0000

Specificity 1.000 1.0000 0.9487

Pos Pred Value 1.000 1.0000 0.8750

Neg Pred Value 1.000 0.9355 1.0000

Prevalence 0.283 0.4528 0.2642

Detection Rate 0.283 0.4151 0.2642

Detection Prevalence 0.283 0.4151 0.3019

Balanced Accuracy 1.000 0.9583 0.9744