Predicting heart disease

Alejandro González

27 de abril, 2021

Contents

Step 1: collecting data **2**

Step 2 - exploring and preparing the data **2**
Shuffled the data . . . 2
Transformation - normalizing numeric data . . . 2
Data preparation - creating training and test datasets . . . 3

Step 3 - Explore the different algorithms **4**
KNN . . . 4
Naïve Bayes classifier . . . 9
Neural Networks . . . 12
Support vector machines . . . 14
Decision trees . . . 17

Step 4: Conclusion section **21**
KNN . . . 21
Neural Network . . . 21
Support Vector Machine . . . 21
Decision trees . . . 21
Final conclusion . . . 21

Step 5: References **22**

Step 1: collecting data

Data in this exercise is a subset of the complete dataset available at http://archive.ics.uci.edu/ml. The dataset was originally scraped from UCI reposritory, you can find it in this link.

```
# import the CSV file
myfile <-"heart.dat"
data <- read.csv(myfile,header = FALSE, sep = " ")
```

Step 2 - exploring and preparing the data

Let's explore the data and see if we can shine some light on the relationships. At the same time, we will prepare the data for use with the kNN learning method.

Using the command `str(data)`, we can confirm that the data is structured with 14 examples and 270 features as we expected. The first several lines of output are as follows:

```
# examine the structure of the data data frame
str(data, list.len = 4)
```

```
## 'data.frame':    270 obs. of  14 variables:
##  $ V1 : num  70 67 57 64 74 65 56 59 60 63 ...
##  $ V2 : num  1 0 1 1 0 1 1 1 1 0 ...
##  $ V3 : num  4 3 2 4 2 4 3 4 4 4 ...
##  $ V4 : num  130 115 124 128 120 120 130 110 140 150 ...
##   [list output truncated]
```

Shuffled the data

In this step I mix the data so that when I randomly select the data is not all the same type of data, for that I use the function `sample ()`

```
n <- nrow(data)
shuffled_rows <- sample(n)
shuffled_data <- data[shuffled_rows, ]
```

Transformation - normalizing numeric data

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`, subtract the minimum value in `x` and divide by the range of values in `x`. Finally, the resulting vector is returned. The code for the function is as follows:

```
# create normalization function
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
```

Next I normalize the data stored in the variable `shuffled_data` and save it in a new variable called`normal_data`, when normalizing the data I make sure that they are all on the same scale, and not that there will be problems when using the algorithms. When normalizing the data, you have to pay attention to the fact that there are columns like two, for example that the range of the values is already between 0 and 1, so in the columns in which this case occurs, they do not have to be normalized. To normalize the data I have used the `lapply ()` function, lapply returns a list of the same length as x, each element of which is the result of applying `normalize ()` to the corresponding element of x.
I have also used the `data.frame ()` function to create data frames, tightly coupled collections of variables that share many of the properties of arrays and lists, used as a fundamental data structure by most R modeling programs .

```
normal_data <- data.frame(lapply(shuffled_data, normalize))
```

we can see that the data has been normalized correctly by using the `str ()` function on the already normalized data:

```
str(normal_data)
```

```
## 'data.frame':    270 obs. of  14 variables:
##  $ V1 : num  0.521 0.312 0.833 0.604 0.688 ...
##  $ V2 : num  1 1 0 1 1 1 1 0 0 1 ...
##  $ V3 : num  0.667 1 0 1 0.333 ...
##  $ V4 : num  0.528 0.17 0.434 0.321 0.321 ...
##  $ V5 : num  0.242 0.374 0.258 0.205 0.187 ...
##  $ V6 : num  0 0 0 0 1 1 0 0 0 0 ...
##  $ V7 : num  1 1 0 1 1 0 0 0 0 0 ...
##  $ V8 : num  0.718 0.626 0.611 0.458 0.527 ...
##  $ V9 : num  0 0 0 1 0 0 0 0 0 0 ...
##  $ V10: num  0.258 0 0.29 0.355 0 ...
##  $ V11: num  0 0 0 0.5 0 0 0 0 0 0 ...
##  $ V12: num  0 0.333 0.667 1 0 ...
##  $ V13: num  1 0 0 1 0 0 0 0 1 0 ...
##  $ V14: num  0 1 0 1 0 0 0 0 0 0 ...
```

Data preparation - creating training and test datasets

To use all the algorithms we need to have the data separated into train and test, we differentiate them into these two types because the trian are the data that we use to train a model and the tests are the data that we reserve to check if the model that we have generated from the training data it works. That is, whether the responses predicted by the model for a totally new case are correct or not.

In this exercise the statement specifies that the data must be divided into a proportion training (67%) and test (33%), to do the division we use the data found in the variable `noraml_data`

```
train_set <- normal_data[1:round(0.67 * n),]
test_set <- normal_data[(round(0.67 * n)+1):n,]
```

To know which column I have to take as a label I went to the web page, where the data appears, since the objective of the work is to know if a patient has a heart disease or not, the column that tells you is 14, in this column there are two values, the number 1 corresponds to the absence of a heart disease and the number 2 means that the patient does have a disease

```
train_set_labels <- train_set[,14]
test_set_labels <- test_set[,14]
```

Step 3 - Explore the different algorithms

To check the correct operation of the algorithms, I will check three factors:
1.- data transformation (if necessary)
2.- model training and evaluation
3.- model improvement (parameter tuning)

KNN

To be able to use the KNN algorithm, all the variables have to be numeric, and although they are expressed with numbers, they are categorical variables, and these are columns 3, 7, 13, I have obtained this information from the web where the data comes from.

Data transformation

To convert them into vectors I use the `factor ()` function, at this point I had a problem, the data was already separated into ** train ** and ** test **, so I repeated the operation of converting them to factor for the two types

```
train_set_KNN <- train_set
test_set_KNN <- test_set
```

```
train_set_KNN$V3 <- factor(train_set_KNN$V3)
train_set_KNN$V7 <- factor(train_set_KNN$V7)
train_set_KNN$V13 <- factor(train_set_KNN$V13)
```

```
test_set_KNN$V3 <- factor(test_set_KNN$V3)
test_set_KNN$V7 <- factor(test_set_KNN$V7)
test_set_KNN$V13 <- factor(test_set_KNN$V13)
```

To check that the change you have made before has worked and is correct, I check it with the `str ()` function

```
str(train_set_KNN)
```

```
## 'data.frame':    181 obs. of  14 variables:
##  $ V1 : num  0.521 0.312 0.833 0.604 0.688 ...
##  $ V2 : num  1 1 0 1 1 1 1 0 0 1 ...
##  $ V3 : Factor w/ 4 levels "0","0.333333333333333",..: 3 4 1 4 2 3 3 4 3 3 ...
##  $ V4 : num  0.528 0.17 0.434 0.321 0.321 ...
##  $ V5 : num  0.242 0.374 0.258 0.205 0.187 ...
##  $ V6 : num  0 0 0 0 1 1 0 0 0 0 ...
##  $ V7 : Factor w/ 2 levels "0","1": 2 2 1 2 2 1 1 1 1 1 ...
##  $ V8 : num  0.718 0.626 0.611 0.458 0.527 ...
##  $ V9 : num  0 0 0 1 0 0 0 0 0 0 ...
##  $ V10: num  0.258 0 0.29 0.355 0 ...
##  $ V11: num  0 0 0 0.5 0 0 0 0 0 0 ...
##  $ V12: num  0 0.333 0.667 1 0 ...
##  $ V13: Factor w/ 3 levels "0","0.75","1": 3 1 1 3 1 1 1 1 3 1 ...
##  $ V14: num  0 1 0 1 0 0 0 0 0 0 ...
```

```
str(test_set_KNN)
```

```
## 'data.frame':    89 obs. of  14 variables:
##  $ V1 : num  0.458 0.208 0.729 0.708 0.312 ...
##  $ V2 : num  1 1 1 1 1 1 1 1 0 0 ...
##  $ V3 : Factor w/ 4 levels "0","0.333333333333333",..: 4 4 1 1 3 2 3 3 1 3 ...
##  $ V4 : num  0.434 0.226 0.151 0.481 0.434 ...
##  $ V5 : num  0.308 0.212 0.194 0.244 0.249 ...
##  $ V6 : num  0 0 0 1 0 0 0 0 0 0 ...
##  $ V7 : Factor w/ 3 levels "0","0.5","1": 3 1 3 3 3 1 1 1 1 2 ...
##  $ V8 : num  0.878 0.527 0.557 0.603 0.832 ...
##  $ V9 : num  1 0 1 0 0 0 0 0 0 0 ...
##  $ V10: num  0 0.194 0.29 0.371 0 ...
##  $ V11: num  0 0.5 0.5 1 0 0 0 1 1 0.5 ...
##  $ V12: num  0 0 0 0 0 0 ...
##  $ V13: Factor w/ 3 levels "0","0.75","1": 1 3 1 2 1 1 3 1 1 1 ...
##  $ V14: num  0 1 0 0 0 0 0 0 0 0 ...
```

Model training and evaluation

To classify our test instances, I will use a kNN implementation from the `class` package, which provides a set of basic R functions for classification. If this package is not already installed on your system, you can install it by typing:

```
# load the "class" library
#install.packages(class)
```

To load the package during any session in which you wish to use the functions, simply enter the command ...

```
library(class)
```

Now I can use the `knn()` function to classify the test data:

```
data_test_KNN <- knn(train = train_set_KNN, test = test_set_KNN, cl = train_set_labels, k = 13)
```

As our training data includes 181 instances, I chose `k = 13`, an odd number roughly equal to the square root of 181 With a two-category outcome, using an odd number eliminates the chance of ending with a tie vote. The `knn()` function returns a factor vector of predicted labels for each of the examples in the test dataset, which I have assigned to `data_test_KNN`.

The next step of the process is to evaluate how well the predicted classes in the `data_test_KNN` vector match up with the known values in the `test_labels` vector. To do this, I will use the `CrossTable()` function in the `gmodels` package. If you haven't done so already, please install this package using the command `install.packages("gmodels")`. After loading the package with ...

```
# load the "gmodels" library
library(gmodels)
```

we can create a cross tabulation indicating the agreement between the two vectors. Specifying `prop.chisq = FALSE` will remove the chi-square values that are not needed, from the output:

```
# Create the cross tabulation of predicted vs. actual
conf.mat <- CrossTable(x = test_set_labels, y = data_test_KNN, prop.chisq = FALSE)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  89
##
##
##                  | data_test_KNN
## test_set_labels |         0 |         1 | Row Total |
## ----------------|-----------|-----------|-----------|
##               0 |        52 |         3 |        55 |
##                 |     0.945 |     0.055 |     0.618 |
##                 |     0.945 |     0.088 |           |
##                 |     0.584 |     0.034 |           |
## ----------------|-----------|-----------|-----------|
##               1 |         3 |        31 |        34 |
##                 |     0.088 |     0.912 |     0.382 |
##                 |     0.055 |     0.912 |           |
##                 |     0.034 |     0.348 |           |
## ----------------|-----------|-----------|-----------|
##    Column Total |        55 |        34 |        89 |
##                 |     0.618 |     0.382 |           |
## ----------------|-----------|-----------|-----------|
##
##
```

The cell percentages in the table indicate the proportion of values that fall into four categories. In the top-left cell (labeled **TN**), are the true negative results. These 52 of 89 values indicate cases where the patient have or not a heart disease, and the kNN algorithm correctly identified it as such. The bottom-right cell (labeled **TP**), indicates the true positive results, where the classifier and the clinically determined label agree that the patient has a disease . A total of 31 of 89 predictions were true positives.

The cells that fall on the other diagonal contain counts of examples where the kNN approach did not agree with the true etiquette. The examples 3 in the lower left cell ** FN ** are false negative results; in this case, the predicted value was that the patient was healthy when he really was not. Errors in this direction can be extremely costly, as It can lead a patient to believe that he is not sick when in fact he does have heart disease.The cell labeled ** FP ** will contain the false positive results, if any. These values occur when the model classifies a healthy patient with one with heart disease.

A total of 3 of 89 masses were incorrectly classified by the kNN approach. While the 98 percent accuracy seems impressive for a few lines of R code, we could try another iteration of the model to see if we can improve the performance and reduce the number of values that have been misclassified, particularly since the errors were dangerous false negatives. Although such errors are less dangerous than a false negative result should also be avoided as they could lead to burden on the health care system, or additional stress on the patient, such as Testing or treatment may need to be provided.

Model improvement

I will attempt two simple variations on our previous classifier. First, we will employ an alternative method for rescaling our numeric features. Second, we will try several different values for k.

Transformation - z-score standardization

Although normalization is traditionally used for kNN classification, it is not always be the most suitable way to scale functions. Because standardized z-score values have no predefined minimums and maximums, extreme values are not compressed toward the center. One might suspect that with heart disease, we could see some very extreme outliers, as some are more serious than others. So it could be reasonable to allow outliers to be weighted more in the distance calculation. Let's see if z-score standardization can improve our predictive accuracy.

To standardize a vector, we can use R's built-in `scale ()` function, which by default rescale the values using z-score standardization. The `scale ()` function offers the added benefit that it can be applied directly to a data frame, so we can avoid using of the `lapply ()` function. To create a standardized z-score version of the data, I'm going to use the following command, which scales all the features with the exception of `V14`, which is the column that says if the patient has heart disease or not

```
# use the scale() function to z-score standardize a data frame
data_z <- as.data.frame(scale(shuffled_data[-14]))
```

To confirm that the transformation was applied correctly, we can look at the summary statistics:

```
# confirm that the transformation was applied correctly
summary(data_z)
```

```
##        V1                 V2                 V3                 V4
##  Min.   :-2.79209   Min.   :-1.4476   Min.   :-2.2883   Min.   :-2.09077
##  1st Qu.:-0.70626   1st Qu.:-1.4476   1st Qu.:-0.1832   1st Qu.:-0.63513
##  Median : 0.06221   Median : 0.6882   Median :-0.1832   Median :-0.07527
##  Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.00000
##  3rd Qu.: 0.72089   3rd Qu.: 0.6882   3rd Qu.: 0.8693   3rd Qu.: 0.48459
##  Max.   : 2.47739   Max.   : 0.6882   Max.   : 0.8693   Max.   : 3.84375
##        V5                 V6                 V7                 V8
##  Min.   :-2.39250   Min.   :-0.4163   Min.   :-1.0244   Min.   :-3.3963
##  1st Qu.:-0.70927   1st Qu.:-0.4163   1st Qu.:-1.0244   1st Qu.:-0.7199
##  Median :-0.09015   Median :-0.4163   Median : 0.9798   Median : 0.1650
##  Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000
##  3rd Qu.: 0.58702   3rd Qu.:-0.4163   3rd Qu.: 0.9798   3rd Qu.: 0.7046
##  Max.   : 6.08171   Max.   : 2.3935   Max.   : 0.9798   Max.   : 2.2586
##        V9                V10                V11                V12
##  Min.   :-0.6999   Min.   :-0.9169   Min.   :-0.9525   Min.   :-0.7102
##  1st Qu.:-0.6999   1st Qu.:-0.9169   1st Qu.:-0.9525   1st Qu.:-0.7102
##  Median :-0.6999   Median :-0.2183   Median : 0.6752   Median :-0.7102
##  Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000
##  3rd Qu.: 1.4234   3rd Qu.: 0.4803   3rd Qu.: 0.6752   3rd Qu.: 0.3492
##  Max.   : 1.4234   Max.   : 4.4970   Max.   : 2.3028   Max.   : 2.4681
##       V13
```

```
##  Min.   :-0.8741
##  1st Qu.:-0.8741
##  Median :-0.8741
##  Mean   : 0.0000
##  3rd Qu.: 1.1871
##  Max.   : 1.1871
```

```
train_set_z <- data_z[1:round(0.67 * n),]
test_set_z <- data_z[(round(0.67 * n)+1):n,]
```

The mean of a z-score standardized variable should always be zero, and the range should be fairly compact. A z-score greater than 3 or less than -3 indicates an extremely rare value. The previous summary seems reasonable. As we had done before, we need to divide the data into training and test sets, then classify the test instances using the `knn()` function. We'll then compare the predicted labels to the actual labels using `CrossTable()`:

```
# re-classify test cases
standardized_KNN <- knn(train = train_set_z, test = test_set_z,
                      cl = train_set_labels, k = 13)
# Create the cross tabulation of predicted vs. actual
conf.mat1 <- CrossTable(x = test_set_labels, y = standardized_KNN,
           prop.chisq = FALSE)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  89
##
##
##                  | standardized_KNN
## test_set_labels |         0 |         1 | Row Total |
## ----------------|-----------|-----------|-----------|
##               0 |        45 |        10 |        55 |
##                 |     0.818 |     0.182 |     0.618 |
##                 |     0.849 |     0.278 |           |
##                 |     0.506 |     0.112 |           |
## ----------------|-----------|-----------|-----------|
##               1 |         8 |        26 |        34 |
##                 |     0.235 |     0.765 |     0.382 |
##                 |     0.151 |     0.722 |           |
##                 |     0.090 |     0.292 |           |
## ----------------|-----------|-----------|-----------|
##    Column Total |        53 |        36 |        89 |
##                 |     0.596 |     0.404 |           |
## ----------------|-----------|-----------|-----------|
##
##
```

Unfortunately, in the following table, the results of our new transformation show a slight decline in accuracy. The instances where I had correctly classified 83 percent of examples previously, I classified only 71 percent correctly this time. Making matters worse, I did no better at classifying the dangerous false negatives.

Testing alternative values of k

I may be able do even better by examining performance across various values of k. Using the normalized training and test datasets, the records were classified using several different `k` values. The number of false negatives and false positives are shown for each iteration:

```
library(knitr)
kable(resum, col.names=c("k value", "# false negatives",
      "# false positives", "% classified Incorrectly"),
      align= c("l","c","c","c"))
```

k value	# false negatives	# false positives	% classified Incorrectly
1	0	0	0.000000
5	1	3	4.494382
11	2	2	4.494382
15	3	3	6.741573
21	3	3	6.741573
27	3	2	5.617978

Naïve Bayes classifier

Data transformation

For the naive bayes algorithm I also have to do the transformation of the data as I have done previously for the knn algorithm, so I repeat the lines of code using the `factor ()` function, converting the categorical variables into numeric variables. In the previous section, I saved the value already converted into a factor over the variables `test_set` and`train_set`.

```
train_set_NB <- train_set
test_set_NB <- test_set
```

```
train_set_NB$V3 <- factor(train_set_NB$V3)
train_set_NB$V7 <- factor(train_set_NB$V7)
train_set_NB$V13 <- factor(train_set_NB$V13)
train_set_NB$V14 <- factor(train_set_NB$V14)
```

```
test_set_NB$V3 <- factor(test_set_NB$V3)
test_set_NB$V7 <- factor(test_set_NB$V7)
test_set_NB$V13 <- factor(test_set_NB$V13)
test_set_NB$V14 <- factor(test_set_NB$V14)
```

To check that the change you have made before has worked and is correct, I check it with the `str ()` function

```
str(train_set_NB)
```

```
## 'data.frame':    181 obs. of  14 variables:
##  $ V1 : num  0.521 0.312 0.833 0.604 0.688 ...
##  $ V2 : num  1 1 0 1 1 1 1 0 0 1 ...
##  $ V3 : Factor w/ 4 levels "0","0.333333333333333",..: 3 4 1 4 2 3 3 4 3 3 ...
##  $ V4 : num  0.528 0.17 0.434 0.321 0.321 ...
##  $ V5 : num  0.242 0.374 0.258 0.205 0.187 ...
##  $ V6 : num  0 0 0 0 1 1 0 0 0 0 ...
```

```
##  $ V7 : Factor w/ 2 levels "0","1": 2 2 1 2 2 1 1 1 1 1 ...
##  $ V8 : num  0.718 0.626 0.611 0.458 0.527 ...
##  $ V9 : num  0 0 0 1 0 0 0 0 0 0 ...
##  $ V10: num  0.258 0 0.29 0.355 0 ...
##  $ V11: num  0 0 0 0.5 0 0 0 0 0 0 ...
##  $ V12: num  0 0.333 0.667 1 0 ...
##  $ V13: Factor w/ 3 levels "0","0.75","1": 3 1 1 3 1 1 1 1 3 1 ...
##  $ V14: Factor w/ 2 levels "0","1": 1 2 1 2 1 1 1 1 1 1 ...
```

Model training and evaluation

The Naive Bayes implementation I will employ is in the `e1071` package. First I will obtain a Naive Bayes model object:

```
library(e1071)
samples_classifier <- naiveBayes(train_set_NB[-14], train_set_NB[,14], laplace = 1)
```

and then I will use it to make predictions on the test data:

```
samples_test_NB <- predict(samples_classifier, test_set_NB)
```

Finally, I will evaluate the performance of our algorithm:

```
library(gmodels)
conf.mat_NB <- CrossTable(samples_test_NB, test_set_labels,
                          prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
                          dnn = c('predicted', 'actual'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  89
##
##
##              | actual
##    predicted |         0 |         1 | Row Total |
## -------------|-----------|-----------|-----------|
##            0 |        43 |         9 |        52 |
##              |     0.483 |     0.101 |           |
## -------------|-----------|-----------|-----------|
##            1 |        12 |        25 |        37 |
##              |     0.135 |     0.281 |           |
## -------------|-----------|-----------|-----------|
## Column Total |        55 |        34 |        89 |
## -------------|-----------|-----------|-----------|
##
##
```

```
accuracy <- sum(diag(conf.mat$t)) / sum(conf.mat$t)
accuracy
```

```
## [1] 0.9438202
```

Model improvement

To improve the ** Naive Bayes ** algorithm, I am going to change the value of `laplace`, this argument of the`naivebayes` function indicates the positive double controlling Laplace smoothing. The default (0) disables Laplace smoothing.

In the confusion matrix you can see that the number of false positives and false negatives is greater than that of the **KNN** algorithm, the comparison of the different algorithms will be done later by comparing all the algorithms to see which one is better to use for this case

```
samples_classifier_1 <- naiveBayes(train_set_NB[ -14], train_set_NB[,14], laplace = 0)
```

```
samples_test_NB_1 <- predict(samples_classifier, test_set_NB)
```

```
library(gmodels)
conf.mat <- CrossTable(samples_test_NB_1, test_set_labels,
                       prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
                       dnn = c('predicted', 'actual'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  89
##
##
##              | actual
##    predicted |         0 |         1 | Row Total |
## -------------|-----------|-----------|-----------|
##            0 |        43 |         9 |        52 |
##              |     0.483 |     0.101 |           |
## -------------|-----------|-----------|-----------|
##            1 |        12 |        25 |        37 |
##              |     0.135 |     0.281 |           |
## -------------|-----------|-----------|-----------|
## Column Total |        55 |        34 |        89 |
## -------------|-----------|-----------|-----------|
##
##
```

```
accuracy <- sum(diag(conf.mat$t)) / sum(conf.mat$t)
accuracy
```

```
## [1] 0.7640449
```

Despite changing the value of `laplace`, it can be seen that the data does not vary, the number of false positives and false negatives is the same and the accuracy does not change either.

Neural Networks

Data transformation

For neural networks, the data have to be previously normalized, I have done this step in the first section, since the data were not on the same scale, and that although they are all numbers, not all the variables are numeric, Also for neural networks, the best way to use the data is if it is normalized, as I have already done in previous sections, it is not necessary to repeat it:

```
train_set_NN <- train_set
test_set_NN <- test_set
```

Model training and evaluation

To model the relationship between molecular descriptors and chemical toxicity, I am going to use a multi-layered feeding neural network. Stefan Fritsch and Frauke Guenther's Neural Networks suite provides a standard, easy-to-use implementation of such networks. It also offers a function for mapping the network topology.

```
#install.packages("neuralnet")
library(neuralnet)
```

The `neuralnet()` syntax is explained in the help page:

```
#?neuralnet
```

I'll begin by training the simplest multilayer feedforward network with the default settings using only a single hidden node:

```
NN_model=neuralnet(V14~ ., data=train_set_NN, hidden = c(3,2),act.fct = "logistic",
                   linear.output = FALSE)
plot(NN_model)
test = data.frame(test_set_NN,test_set_labels)
Predict = compute(NN_model,test)
predicted_model_NN <- Predict$net.result
```

In this simple model, there is one input node for each of the eight features, followed by a single hidden node and a single output node that predicts the concrete strength. The weights for each of the connections are also depicted, as are the bias terms indicated by the nodes labeled with the number 1. The bias terms are numeric constants that allow the value at the indicated nodes to be shifted upward or downward, much like the intercept in a linear equation.

At the bottom of the figure, R reports the number of training steps and an error measure called the sum of squared errors (SSE), which, as you might expect, is the sum of the squared differences between the predicted and actual values. The lower the SSE, the more closely the model conforms to the training data, which tells us about performance on the training data but little about how it will perform on unseen data.

```r
library(gmodels)
conf.mat_NN <- CrossTable(pred, test_set_labels,
                          prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
                          dnn = c('predicted', 'actual'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  89
##
##
##              | actual
##    predicted |         0 |         1 | Row Total |
## -------------|-----------|-----------|-----------|
##            0 |        37 |         9 |        46 |
##              |     0.416 |     0.101 |           |
## -------------|-----------|-----------|-----------|
##            1 |        18 |        25 |        43 |
##              |     0.202 |     0.281 |           |
## -------------|-----------|-----------|-----------|
## Column Total |        55 |        34 |        89 |
## -------------|-----------|-----------|-----------|
##
##
```

Support vector machines

Data transformation

To do the SVM I have to pass column 14, which is the one that indicates whether the patient has a heart disease or not, to a factor, I do that using the `factor ()` function

```
train_set_SVM <- train_set
test_set_SVM <- test_set
```

```
train_set_SVM$V14 <- factor(train_set$V14)
test_set_SVM$V14 <- factor(test_set$V14)
str(train_set_SVM)
```

```
## 'data.frame':    181 obs. of  14 variables:
##  $ V1 : num  0.521 0.312 0.833 0.604 0.688 ...
##  $ V2 : num  1 1 0 1 1 1 1 0 0 1 ...
##  $ V3 : num  0.667 1 0 1 0.333 ...
##  $ V4 : num  0.528 0.17 0.434 0.321 0.321 ...
##  $ V5 : num  0.242 0.374 0.258 0.205 0.187 ...
##  $ V6 : num  0 0 0 0 1 1 0 0 0 0 ...
##  $ V7 : num  1 1 0 1 1 0 0 0 0 0 ...
##  $ V8 : num  0.718 0.626 0.611 0.458 0.527 ...
##  $ V9 : num  0 0 0 1 0 0 0 0 0 0 ...
##  $ V10: num  0.258 0 0.29 0.355 0 ...
##  $ V11: num  0 0 0 0.5 0 0 0 0 0 0 ...
##  $ V12: num  0 0.333 0.667 1 0 ...
##  $ V13: num  1 0 0 1 0 0 0 0 1 0 ...
##  $ V14: Factor w/ 2 levels "0","1": 1 2 1 2 1 1 1 1 1 1 ...
```

Model training and evaluation

I'll use the SVM functions in the `kernlab` package. After you install the package, you can look at the documentation by typing `?ksvm`.

```
#install.packages("kernlab")
library(kernlab)
#?ksvm
```

```
letter_classifier <- ksvm(V14 ~ ., data = train_set_SVM,  kernel = "vanilladot")
```

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

```
letter_classifier_1 <- ksvm(V14 ~ ., data = train_set_SVM,  kernel = "rbfdot")
```

When the training finishes (it can take some time depending on your computer), I can inspect some basic information about the training parameters and the fit of the model:

```
letter_classifier
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Linear (vanilla) kernel function.
##
## Number of Support Vectors : 64
##
```

```
## Objective Function Value : -55.2217
## Training error : 0.127072
```

```
letter_classifier_1
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.0511954278740874
##
## Number of Support Vectors : 97
##
## Objective Function Value : -60.4026
## Training error : 0.077348
```

This information tells us very little about how well the model will perform in the real world. I'll need to examine its performance on the testing dataset to know whether it generalizes well to unseen data.

Model improvement

he `predict()` function allows us to use the letter classification model to make predictions on the testing dataset:

```
letter_predictions <- predict(letter_classifier, test_set_SVM)
```

```
letter_predictions_1 <- predict(letter_classifier_1, test_set_SVM)
```

This returns a vector containing a predicted letter for each row of values in the testing data. Using the head() function, we can see that the first predicted letters:

```
head(letter_predictions)
```

```
## [1] 0 1 0 0 0 0
## Levels: 0 1
```

```
head(letter_predictions_1)
```

```
## [1] 0 1 0 1 0 0
## Levels: 0 1
```

```
conf.mat_SVM <- table(letter_predictions, test_set_SVM$V14)
conf.mat_SVM
```

```
##
## letter_predictions  0  1
##                  0 46  6
##                  1  9 28
```

```
table(letter_predictions_1, test_set_SVM$V14)
```

```
##
## letter_predictions_1  0  1
##                    0 44  7
##                    1 11 27
```

Looking at each type of mistake individually may reveal some interesting patterns about the specific types of letters the model has trouble with, but this is time consuming. I can simplify our evaluation by instead

calculating the overall accuracy.

The following command returns a vector of TRUE or FALSE values indicating whether the model's predicted letter agrees with (that is, matches) the actual letter in the test dataset:

```
agreement <- letter_predictions == test_set_SVM$V14
```

```
table(agreement)
```

```
## agreement
## FALSE  TRUE
##    15    74
```

```
prop.table(table(agreement))
```

```
## agreement
##     FALSE      TRUE
## 0.1685393 0.8314607
```

Try an RBF kernel

```
set.seed(12345)
letter_classifier_rbf <- ksvm(V14 ~ ., data = train_set_SVM, kernel = "rbfdot")
letter_predictions_rbf <- predict(letter_classifier_rbf, test_set_SVM)

agreement_rbf <- letter_predictions_rbf == test_set_SVM$V14
table(agreement_rbf)
```

```
## agreement_rbf
## FALSE  TRUE
##    17    72
```

```
prop.table(table(agreement_rbf))
```

```
## agreement_rbf
##     FALSE      TRUE
## 0.1910112 0.8089888
```

Test various values of the cost parameter

Now we will examine how the model performs for various values of C, the cost parameter. Rather than repeating the training and evaluation process repeatedly, we can use the `sapply()` function to apply a custom function to a vector of potential cost values.

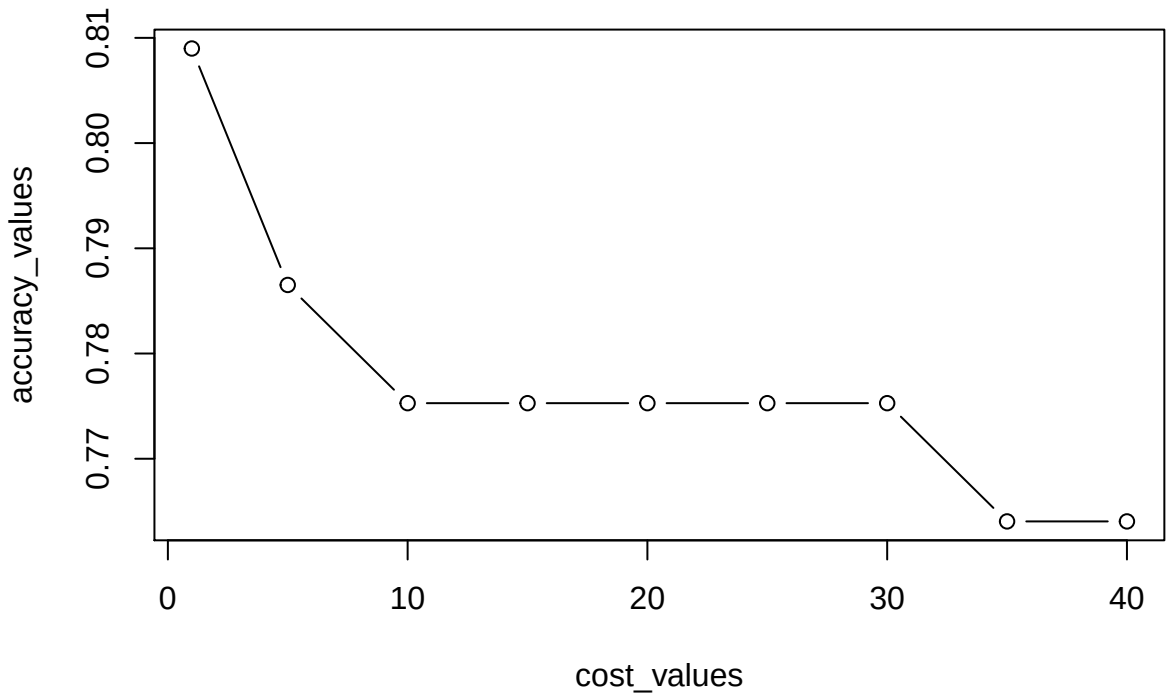
We begin by using the `seq()` function to generate this vector as a sequence counting from five to 40 by five. Then, as shown in the following code, the custom function trains the model as before, each time using the cost value and making predictions on the test dataset. Each model's accuracy is computed as the number of predictions that match the actual values divided by the total number of predictions. The result is visualized using the `plot()` function:

```
cost_values <- c(1, seq(from = 5, to = 40, by = 5))

accuracy_values <- sapply(cost_values, function(x) {
  set.seed(12345)
  m <- ksvm(V14 ~ ., data = train_set_SVM,
            kernel = "rbfdot", C = x)
  pred <- predict(m, test_set_SVM)
  agree <- ifelse(pred == test_set_SVM$V14, 1, 0)
  accuracy <- sum(agree) / nrow(test_set_SVM)
```

```
  return (accuracy)
})

plot(cost_values, accuracy_values, type = "b")
```



Decision trees

In order to use the decision tree classification I need to download these libraries:

Data transformation

```
train_set_DT <- train_set
test_set_DT <- test_set
```

```
train_set_DT$V14 <- factor(train_set_DT$V14)
test_set_DT$V14 <- factor(test_set_DT$V14)
```

Model training and evaluation

The package for the C5.0 desicion tree algorithm must be loaded into the session:

```
# install.packages("C50")
require(C50)
```

Let's create the decision treee model and predict the outcome of the target feature for our test data.

```
tree_model <- C5.0(train_set_DT[,-14], train_set_DT[,14])

tree_model
```

```
##
## Call:
## C5.0.default(x = train_set_DT[, -14], y = train_set_DT[, 14])
##
## Classification Tree
## Number of samples: 181
## Number of predictors: 13
##
## Tree size: 16
##
## Non-standard options: attempt to group attributes
```

By using the function `summary()` you can see the decisions that determine the splits of our data:

```
summary(tree_model)
```

```
##
## Call:
## C5.0.default(x = train_set_DT[, -14], y = train_set_DT[, 14])
##
##
## C5.0 [Release 2.07 GPL Edition]      Tue Apr 27 20:02:06 2021
## -------------------------------
##
## Class specified by attribute `outcome'
##
## Read 181 cases (14 attributes) from undefined.data
##
## Decision tree:
##
## V13 > 0:
## :...V9 > 0:
## :   :...V4 <= 0.1698113: 0 (3/1)
## :   :   V4 > 0.1698113: 1 (39)
## :   V9 <= 0:
## :   :...V12 > 0: 1 (22/2)
## :       V12 <= 0:
## :       :...V4 <= 0.1698113: 1 (2)
## :           V4 > 0.1698113:
## :           :...V3 > 0.3333333: 0 (9)
## :               V3 <= 0.3333333:
## :               :...V1 <= 0.5625: 0 (3)
## :                   V1 > 0.5625: 1 (2)
## V13 <= 0:
## :...V10 > 0.2580645:
##     :...V11 <= 0: 0 (2)
##     :   V11 > 0: 1 (9/1)
##     V10 <= 0.2580645:
##     :...V3 <= 0.6666667: 0 (58/2)
##         V3 > 0.6666667:
##         :...V12 > 0:
```

```
##             :...V2 <= 0: 0 (2)
##             :   V2 > 0: 1 (7)
##             V12 <= 0:
##             :...V1 <= 0.6041667: 0 (13)
##                 V1 > 0.6041667:
##                 :...V2 > 0: 0 (3/1)
##                     V2 <= 0:
##                     :...V1 <= 0.6666667: 1 (3)
##                         V1 > 0.6666667: 0 (4/1)
##
##
## Evaluation on training data (181 cases):
##
##      Decision Tree
##    ----------------
##    Size      Errors
##
##      16    8( 4.4%)   <<
##
##
##     (a)   (b)    <-classified as
##    ----  ----
##      92     3    (a): class 0
##       5    81    (b): class 1
##
##
##  Attribute usage:
##
##  100.00% V13
##   57.46% V3
##   55.80% V10
##   44.20% V9
##   38.67% V12
##   32.04% V4
##   15.47% V1
##   10.50% V2
##    6.08% V11
##
##
## Time: 0.0 secs
```

`stalk_root`, indicated at the end of the output. Let's predict the outcome of the target feature for our test set:

```
DT_prediction <- predict(tree_model, test_set_DT[,-14])
```

By creating a cross table, we will see how accurate our model has been:

```
library(gmodels)
conf.matrix_DT <- CrossTable(test_set_DT[,14], DT_prediction, prop.chisq = FALSE,
prop.c = FALSE, prop.r = FALSE, dnn = c('actual', 'predicted'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  89
##
##
##              | predicted
##       actual |         0 |         1 | Row Total |
## -------------|-----------|-----------|-----------|
##            0 |        38 |        17 |        55 |
##              |     0.427 |     0.191 |           |
## -------------|-----------|-----------|-----------|
##            1 |        11 |        23 |        34 |
##              |     0.124 |     0.258 |           |
## -------------|-----------|-----------|-----------|
## Column Total |        49 |        40 |        89 |
## -------------|-----------|-----------|-----------|
##
##
```

```
accuracy <- (conf.matrix_DT$t[1,1] + conf.matrix_DT$t[2,2])/nrow(test_set_DT)
print(accuracy)
```

```
## [1] 0.6853933
```

Model improvement

There are several techniques that can be used to improve a model using decision trees, these are among others:

Bagging: generateseveral training datasets by bootstrap sampling the original training data.These datasets are then used to generate a set of models using a single learning algorithm.

Random Forests: Ensemble of decision trees: bagging + feature subsets Each tree is trained with only a random subset of features

Boosting: Sequential production of classifiers Each classifier is dependent on the previous one, and focuses on the previous one's errors

Step 4: Conclusion section

In this last section I am going to compare the confusion matrices of all the algorithms made previously to see which one has the least errors, the algorithm that has the least errors, will be the best at the time of predicting whether or not patients may have a heart disease, I understand ** FN ** and ** FP ** as failures or errors.

Another thing that I will compare is the time it takes to process the data, in addition to whether you have to modify the initial data to be able to do the algorithm, since having to do it adds extra time

KNN

The good thing about KNN is that it is very simple and effective, this can be seen by the number of badly classified predictions, which is very low, more than the rest of the algorithms, it is also very fast in the training phase, the problems What the KNN has is that you have to do tests to check which is the optimal value of k, you also have to modify the data so as not to make errors, so the classification phase is a little slower than other algorithms and requires work additional

Naive Bayes

Like the KNN, the naive bayes classifier, it is very simple, fast and efficient, an advantage that this algorithm has that others do not have is that it is good at dealing with noisy data, which is data that is not necessary at the time of classify them, but one of the problems it has is that it is not good if we have a lot of numerical data, and also Estimated probabilities are less reliable than the predicted classes

Neural Network

The neural network is not the best algorithm when it comes to predicting data, although it is used to classify and numerical predictions, it is very slow to train, and it is more complex to program than others and is very prone to overfitting. and the results are difficult to interpret

Support Vector Machine

The support vector machine like the neural network can be used for classification or numerical predictions, what differentiates it from neural networks is that it is not due to overfitting and is not influenced by noise data, and it is easier to program than Neural networks, the bad thing about it is that it is slow to train and the results are difficult to interpret, and due to the kernels that exist to find the optimal solution we will have to try all the combinations, some being very slow at the time of processing and loading the data by creating a model

Decision trees

Finally, the decision trees algorithm, the good thing it has is that it is more efficient than other more complex models, it can be used for very extensive and smaller data, in addition to that it can be used with categorical as well as numerical data, the problem it has is which is easy to overfit, and if there is a small initial change it may change the result a lot, and if the tree is very long it can be very difficult to interpret

Final conclusion

The final conclusion that I come to is that the best algorithm that has come out to me to classify and predict the presence or absence of cardiac disease is the KNN algorithm, since it is easy and fast to use, and although some data has to be modified for the correct realization of the algorithm it is the most reliable in terms of the number of misclassified data

Step 5: References

Most of the information I have taken from the pdu, as well as some parts of code, the data with which I carry out the study as I have already specified at the beginning of the work I took them from:
http: //archive.ics.uci .edu / ml, also in order to solve the neural network confusion matrix, look at this link: https://www.datacamp.com/community/tutorials/neural-network-models-r, the rest of the information by asking the teacher