Assignment 1

Lúa Arconada & Alejandro Macías

14/12/2023

Assignment 1

Before doing anything, we have to load our created package with all our functions to be able to use them to show that they work and do what they have to do.

```
library(myknn)
library(Rcpp)
```

Exercise 1

In this exercise, we have to write in C++ language a function called 'my__knn__c', the already coded KNN algorithm in R language called 'my__knn__R'. We will compile it using sourceCpp, which we do by loading library(Rcpp) and, then, writing cppFunction('my__knn__c' in the C++ language).

```
cppFunction('
int my_knn_c ( NumericMatrix X, NumericVector X0, NumericVector y){
  // X data matrix with input attributes
  // y response variable values of instances in X
  // X0 vector of input attributes for prediction (just one instance)
  int nrows=X.nrow();
  int ncols=X.ncol();
  double distance=0;
  int j;
  double difference;
  for (j=0;j<ncols;j++){
    difference = X(1,j)-X0[j];
    distance+=difference*difference;
  }
  distance = sqrt(distance);
  double closest_distance=distance;
  double closest_output = y[1];
  int closest_neighbor=1;
  for (int i=1;i<nrows;i++){
    distance=0;
    for (int j=0;j<ncols;j++){
      difference = X(i,j)-X0[j];
      distance+= difference*difference;
    }
    distance=sqrt(distance);
    if (distance<closest_distance){
      closest_distance = distance;
      closest_output = y[i];
      closest_neighbor = i;
    }
  }
  return closest_output;
}
')
```

This function has three arguments:

1. A data matrix X with input attributes.
2. A response vector y which contains the values of the response variable of the instances of X.
3. A vector X0 of input attributes for prediction (just one instance). Its output will be a number indicating the closest category, in other words, the predicted category for our instance input X0.

Firstly, through a loop, we calculate the distance between the first observation of the data in X and X0 and we create a variable 'distance' which will be equal to the sum of the second power of these differences. After that, we compute its square root and we save the result in a variable called 'closest_distance'. We know that the first coordinate of the prediction vector y is the predicted category for the first observation of X and we save that prediction in another variable as 'closest_output'. Also, we keep track that this is (at the moment) the closest neighbour by saving the number of this observation in another variable called 'closest_neighbor'. So, at the moment, closest_neighbor=1 and closest_output=y[1] (first coordinate of vector y). However, we have to see if this is in fact the closest neighbour or if it is one of the other observations we have in X, so we will compare them one by one to this one.

We implement another loop that will consider each of the rows of X from the second to the last one by one. The loop calculates the same distance as before for the row selected and it checks if that distance is smaller than the computed using the first row (stored in 'closest_distance'). If that condition happens, then the value of 'closest_distance' will be updated to this new one, the value of 'closest_output' will be the coordinate of the vector y corresponding to the prediction of the row selected and the 'closest_neighbour' will be the number of the row used to compute the distance. Lastly, the function will return the value of the variable 'closest_neighbour', which will be the predicted category for our input X0.

After coding this function, we are asked to check that we obtain the same results as with the function 'knn' contained in library 'class'. First, we get our input matrix X and vector y and X:

```r
# X contains the inputs as a matrix of real numbers
data("iris")
# X contains the input attributes (excluding the class which is the prediction)
X <- iris[,-5]
# y contains the response variable, the prediction (named medv, a numeric value)
y <- iris[,5]

# From dataframe to matrix
X <- as.matrix(X)
# From factor to integer
y <- as.integer(y)

# This is the point we want to predict
X0 <- c(5.80, 3.00, 4.35, 1.30)
```

```r
# We use my_knn_R, my_knn_c and class:knn to predict point X0; and to check that
# the results are the same to see if our function 'my_knn_c' is right.

print(my_knn_R(X, X0, y))
```

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

```
print(my_knn_c(X, X0, y))
```

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

```
library(class)
print(class::knn(X, X0, y, k=1))
```

```
## [1] 2
## Levels: 1 2 3
```

We have computed the prediction of X0's category using the given function 'my_knn_R', our C++ function 'my_knn_c' and the 'knn' function in the class library. We can see that the three outcomes are the same, which is what we wanted.

Finally, we are asked to use library 'microbenchmark' to compare the time it takes each function to compute the prediction. We want to see if our C++ function is faster than either, none or both of the other two.

```
library(microbenchmark)
comparison=microbenchmark(my_knn_R(X,X0,y),my_knn_c(X,X0,y),class::knn(X,X0,y))
comparison
```

```
## Unit: microseconds
##                   expr    min      lq     mean  median      uq    max neval cld
##     my_knn_R(X, X0, y) 1304.5 1317.90 1541.450 1482.80 1574.00 4219.2   100 a
##     my_knn_c(X, X0, y)    4.4    5.55   31.543    7.35   10.85 2192.3   100  b
##   class::knn(X, X0, y)  103.9  116.85  149.595  127.10  165.65  490.5   100   c
```

```
#Microbenchmark creates a dataframe of 100 iterations of each function and it
#gives its time
```

We can see in the mean column, that the C++ function is much faster that the other two.

Exercise 2

Now we are asked to write a C++ function that computes the Euclidean distance and implement it in our main function, which we will call 'my_knn_c_euclidean'.

Our function 'euclidean' takes two vectors x and y and computes the square of the sum of the power of their difference (which is the Euclidean distance of x and y).

Then, our function 'my_knn_c_euclidean' does exactly the same as 'my_knn_R' and 'my_knn_c', but the difference is the way it computes de distance: 'my_knn_c_euclidean', makes use of a previously implemented function to compute the Euclidean distance within its loop.

```
my_knn_c_euclidean(X,X0,y)
```

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

We can see that it predicts the same category for our X0 as the functions in Exercise 1.

Exercise 3

This exercise is like the previous one, but in this one we first have to code a function that computes the Minkowsky distance and then, the main function 'my_knn_c_minkowsky' will implement it.

Our function 'minkowsky' takes an extra argument p (it still needs X, X0 and y). If p is negative or zero, then the Minkowsky distance is computed as the maximum of the absolute values of the difference (which is the L_infinity distance). However, if p>0, then the distance is computed as the 1/p power of the sum of the difference between x and y, each raised to the power of p.

Moreover, our function 'my_knn_c_minkowsky' does, once again, the same as the function 'my_knn_c' but takes one more argument p (the exponent) and computes the distance as the Minkowsky distance using our 'minkowksi' function.

```r
my_knn_c_minkowsky(X,X0,y,p=5)
```

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

It should be noted that, in this example where $p = 5$ has been used, the function predicts a different label for X0.

Exercise 4

In this last exercise, we are asked to write a function called 'my_knn_tuningp' that instead of using a given p, it carries out hyper-parameter tuning of the Minkowsky exponent p. We will do this by adding a new argument to the function, called 'possible_p', which is a vector of possible values of p.

Firstly, we will split the data X with 2/3-proportion for training and 1/3-proportion for testing. To do so, we define four auxiliary functions. Two functions, 'keepFirstTwoThirds' and 'keepLastOneThird', will have a matrix X as the input and will give back matrices consisting of the first two thirds and the last third respectively (in terms of rows). The other two, 'keepFirstTwoThirdsVector' and 'keepLastOneThirdVector', will do the same but the inputs and outputs will be vectors.

After doing this, we write our function which will go through all possible given values of p. Then, it will compute the accuracy and choose the parameter corresponding to the highest accuracy. And, finally, it will make a prediction using this chosen p. Now, we are going to describe in more detail what this function does.

First of all, it creates two variables called 'best' and 'best_p' to keep track of the best accuracy and the corresponding p and it uses our four additional previously defined functions to split X and y in train-test. Then, it takes a possible value of p from the input vector that contains them and uses our function 'my_knn_c_minkowsky' to obtain the prediction and if it is equal to the true value stored in y. Then, we add one to a counter we call 'right' which keeps track of the accurate predictions. Afterwards, we calculate the accuracy of that p as the number of right predictions (counter 'rights') divided by the total number of observations we have predicted. In addition, we check if the accuracy with this p is bigger than the one stored in the variable 'best', and if it is, we save that exponent as the best yet in our variable 'best_p'.

We do all this work with a p inside a loop that goes through all the values of the input vector that contains all its possible values. We are checking if each value has a bigger accuracy than the previous p and only if it is we update the variable 'best_p' to that better p. And lastly, we ask our function to tell us what the best value of p ('best_p') was and to compute the knn prediction using the Minkowsky distance and that p.

```r
my_knn_tuningp(X,X0,y,seq(0,1,0.2))
```

```
## The best value for p is:0
```

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

Lastly, we can see that it makes the same prediction for X0 as all our previous functions (except 'my_knn_minkowsky for $p = 5$'), and the best $p = 0$.