Advanced Programming: Assignmment 1

Daniel Alonso

November 27th, 2020

Importing libraries

```
library(Rcpp)
library(microbenchmark)
library(FNN)
```

Importing all the Rcpp function from the .cpp file in ./assignment1knn/src/knn.cpp using sourceCpp:

This will import the knn translation from the class example and the 2 functions made to work (the one with the normal voting system, and the other one with the inverse of distance voting system) with K = 3

```
sourceCpp('./assignment1knn/src/knn.cpp')
```

Exercise 1

Class example (this code is NOT mine)

```
my_knn_R = function(X, X0, y){
  # X data matrix with input attributes
  # y response variable values of instances in X
  # X0 vector of input attributes for prediction

  nrows = nrow(X)
  ncols = ncol(X)

  # One of the instances is going to be the closest one:
  # closest_distance: it is the distance , min_output
  closest_distance = 99999999
  closest_output = -1
  closest_neighbor = -1

  for (i in 1:nrows) {

    distance = 0
    for (j in 1:ncols) {
      difference = X[i,j]-X0[j]
      distance = distance + difference * difference
    }

    distance = sqrt(distance)
```

```
    if (distance < closest_distance) {
      closest_distance = distance
      closest_output = y[i]
      closest_neighbor = i
    }
  }
  closest_output
}
```

Testing class example (This code is NOT mine)

```
# X contains the inputs as a matrix of real numbers
data("iris")
# X contains the input attributes (excluding the class)
X <- iris[,-5]
# y contains the response variable (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)
# Using my_knn and FNN:knn to predict point X0
# Using the same number of neighbors, it should be similar (k=1)
my_knn_R(X, X0, y)
```

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

```
FNN::knn(X, matrix(X0, nrow = 1), y, k=1)
```

```
## [1] 2
## attr(,"nn.index")
##      [,1]
## [1,]   96
## attr(,"nn.dist")
##           [,1]
## [1,] 0.2061553
## Levels: 2
```

Translating the teacher's code into an Rccp function

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int knn_1(NumericMatrix X, NumericVector X0, NumericVector y) {
    int nrows = X.nrow();
    int ncols = X.ncol();

    double closest_distance = 99999999;
    double closest_output = -1;
    double closest_neighbor = -1;
    double difference = 0;

    int i;
    int j;

    for (i = 0; i < nrows; i++) {

        double distance = 0;
        for (j = 0; j < ncols; j++) {
            difference = X(i,j) - X0(j);
            distance = distance + difference * difference;
        }

        distance = sqrt(distance);

        if (distance < closest_distance) {
            closest_distance = distance;
            closest_output = y(i);
            closest_neighbor = i;
        }
    }
    return closest_output;
}
```

Testing Rcpp translation

```
knn_1(X, X0, y)
```

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

Benchmarking differences in runtime between R version and Rcpp version

R version

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

```
## Unit: microseconds
##                expr     min       lq     mean  median      uq      max neval
##  my_knn_R(X, X0, y) 722.171 754.4265 859.5091 876.926 904.281 2154.892   100
```

We can see that our mean runtime for the R version is more than 800 microseconds

FNN version

```r
microbenchmark(FNN::knn(X, matrix(X0, nrow = 1), y, k=1))
```

```
## Unit: microseconds
##                                          expr     min      lq     mean  median
##  FNN::knn(X, matrix(X0, nrow = 1), y, k = 1) 229.501 236.851 259.9678 241.966
##       uq     max neval
##  252.911 602.391   100
```

We can see that our mean runtime for the Rcpp version is of under 250 microseconds

Rcpp version

```r
microbenchmark(knn_1(X, X0, y))
```

```
## Unit: microseconds
##             expr   min    lq    mean median    uq     max neval
##  knn_1(X, X0, y) 5.221 5.321 15.1948  5.381 5.506 980.301   100
```

We can see that our mean runtime for the Rcpp version is of under 14 microseconds

Exercise 2

Implementation for k>1 (but works well for k=3)

```
knn_more_R = function(X, X0, y, K){
  # X data matrix with input attributes
  # y response variable values of instances in X
  # X0 vector of input attributes for prediction

  nrows = nrow(X)
  ncols = ncol(X)

  # One of the instances is going to be the closest one:
  # closest_distance: it is the distance , min_output
  distances = c()
  closest_distance = 1e99
  closest_neighbor = -1
  closest_classif = -1

  # get distances
  for (i in 1:nrows) {

    distance = 0
    for (j in 1:ncols) {
      difference = X[i,j]-X0[j]
      distance = distance + difference * difference
    }

    distance = sqrt(distance)

    # add distance to vector
    distances = c(distances, distance)

    if (distance < closest_distance) {
      closest_distance = distance
      closest_classif = y[i]
      closest_neighbor = i
    }
  }

  # eliminating closest distance
  NN_distances = c(closest_distance)
  NN_classif = c(closest_classif)
  distances[closest_neighbor] = 1e99
  distances = unname(distances)

  # We already got the closest so remove one from K
  K = K - 1

  # because we can't sort, we just manually pull out the minimum value K times
  # by subtracting each distance to the previous closest distance
  for (i in 1:K) {

    # placeholder variables for loop
```

```
  diff = 0
  min_diff = 1e99
  index = 0

  # calculate diffs between distances and closest distance
  # the lowest is saved in placeholder variable min_diff
  # then the index is saved in the index variable
  for (idx in 1:nrows) {
    diff = distances[idx] - NN_distances[i]
    if (diff < min_diff) {
      min_diff = diff
      index = idx
    }
  }

  # add the corresponding distance to NN distances
  # add the corresponding classif to NN classif
  NN_distances = c(NN_distances, distances[index])
  NN_classif = c(NN_classif, y[index])
  distances[index] = 1e99
}

# different classifications
classifs = unique(y)

# loop through classifications to count
cnts = matrix(rep(0,6), nrow=length(classifs), byrow=TRUE)
cnts[,1] = classifs;
for (g in NN_classif) {
  cnts[g,2] = cnts[g,2] + 1
}

# check if there's identical counts
count_vector = cnts[,2]
group_normally = 0
if (K %% 2 == 0) {
  if (length(count_vector[count_vector == max(count_vector)]) > 1) {
    for (i in 1:K) {
      if (NN_distances[i] == min(NN_distances)) {
        group = NN_classif[i]
      }
    }
  } else {
    group_normally = 1
  }
}
else {
  group_normally = 1
}

# select maximum value
if (group_normally == 1) {
  group = 0
```

```r
    for (i in cnts[,1]) {
      if (count_vector[i] == max(count_vector)) {
        group = i
      }
    }
  }

  group
}
```

Testing R implementation for k=3

```r
knn_more_R(X, X0, y, 3)
```

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

Translating our knn implementation for k=3 into Rcpp

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int knn_more(NumericMatrix X, NumericVector X0, NumericVector y, int K) {
    int nrows = X.nrow();
    int ncols = X.ncol();

    NumericVector distances(nrows);
    NumericVector NN_distances(K);
    NumericVector NN_classif(K);
    double closest_distance = 99999999999999999;
    double closest_output = -1;
    double closest_neighbor = -1;
    double difference;

    int i;
    int j;


    for (i = 0; i < nrows; i++) {

        double distance = 0;
        for (j = 0; j < ncols; j++) {
            difference = X(i,j) - X0(j);
            distance = distance + difference * difference;
        }

        distance = sqrt(distance);
        distances[i] = distance;

        if (distance < closest_distance) {
            closest_distance = distance;
            closest_output = y(i);
            closest_neighbor = i;
        }
    }

    K = K - 1;
    NN_distances(0) = closest_distance;
    NN_classif(0) = closest_output;
    distances(closest_neighbor) = 99999999999999999;

    int idx;
    for (i = 0; i < K; i++) {
      double diff = 0;
      double min_diff = 99999999999999999;
      int index = 0;
      for (idx = 0; idx < nrows; idx++) {
        diff = distances(idx) - NN_distances(i);
        if (diff < min_diff) {
          min_diff = diff;
```

```
          index = idx;
        }
      }
      NN_distances(i+1) = distances(index);
      NN_classif(i+1) = y(index);
      distances(index) = 9999999999999999;
    }

    NumericVector classifs(unique(y).size());
    for (i = 0; i < unique(y).size(); i++) {
      classifs[i] = i+1;
    }

    NumericMatrix cnt(classifs.size(), 2);
    for (i = 0; i < classifs.size(); i++) {
      cnt(i,0) = classifs(i);
      cnt(i,1) = 0;
    }
    for (i = 0; i < NN_classif.size(); i++) {
      cnt(NN_classif(i)-1,1) = cnt(NN_classif(i)-1,1) + 1;
    }

    NumericVector count_vector = cnt(_,1);
    NumericVector maxes = count_vector[count_vector == max(count_vector)];
    int group = 0;
    int group_normally = 0;
    int maxes_size = maxes.size();
    if (K % 2 == 0) {
      if (maxes_size > 1) {
        for (i = 0; i < K; i++) {
          if (NN_distances(i) == min(NN_distances)) {
            group = NN_classif(i);
          }
        }
      } else {
        group_normally = 1;
      }
    } else {
      group_normally = 1;
    }

    if (group_normally == 1) {
      for (i = 0; i < classifs.size(); i++) {
        if (count_vector(i) == max(count_vector)) {
          group = i+1;
        }
      }
    }

  return group;
}
```

Testing our Rcpp implementation for k=3

```
knn_more(X, X0, y, 3)
```

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

Benchmarking the implementation for k=3

Benchmarking our R implementation for k=3

```
microbenchmark(knn_more_R(X, X0, y, 3))
```

```
## Unit: milliseconds
##                     expr      min       lq     mean   median       uq      max
##  knn_more_R(X, X0, y, 3) 1.116211 1.160662 1.287584 1.213246 1.276252 3.933092
##  neval
##    100
```

We can see that the R code takes a bit under 1.5 seconds to finish

Benchmarking our Rcpp implementation for k=3

```
microbenchmark(knn_more(X, X0, y, 3))
```

```
## Unit: microseconds
##                   expr    min     lq    mean median     uq      max neval
##  knn_more(X, X0, y, 3) 15.481 15.871 26.5342 16.116 16.341 1055.862   100
```

Our Rcpp implementation takes under 24 microseconds on average

Benchmarking the FNN knn function for k=3

```
microbenchmark(FNN::knn(X, matrix(X0, nrow = 1), y, k=3))
```

```
## Unit: microseconds
##                                          expr     min       lq     mean   median
##  FNN::knn(X, matrix(X0, nrow = 1), y, k = 3) 228.221 233.1715 246.9397 235.8765
##       uq     max neval
##  244.096 557.451   100
```

The knn function from the FNN library takes under 250 microseconds on average

Exercise 3

Modifying distances voting system to use 1/distance

```cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int knn_inv(NumericMatrix X, NumericVector X0, NumericVector y, int K) {
    int nrows = X.nrow();
    int ncols = X.ncol();

    NumericVector distances(nrows);
    NumericVector NN_distances(K);
    NumericVector NN_classif(K);
    double closest_distance = 9999999999999999;
    double closest_output = -1;
    double closest_neighbor = -1;
    double difference;

    int i;
    int j;

    for (i = 0; i < nrows; i++) {

        double distance = 0;
        for (j = 0; j < ncols; j++) {
            difference = X(i,j) - X0(j);
            distance = distance + difference * difference;
        }

        distance = sqrt(distance);
        distances[i] = distance;

        if (distance < closest_distance) {
            closest_distance = distance;
            closest_output = y(i);
            closest_neighbor = i;
        }
    }

    K = K - 1;
    NN_distances(0) = closest_distance;
    NN_classif(0) = closest_output;
    distances(closest_neighbor) = 9999999999999999;

    int idx;
    for (i = 0; i < K; i++) {
      double diff = 0;
      double min_diff = 9999999999999999;
      int index = 0;
      for (idx = 0; idx < nrows; idx++) {
        diff = distances(idx) - NN_distances(i);
        if (diff < min_diff) {
```

```
          min_diff = diff;
          index = idx;
        }
      }
      NN_distances(i+1) = distances(index);
      NN_classif(i+1) = y(index);
      distances(index) = 9999999999999999;
    }

    NumericVector classifs(unique(y).size());
    for (i = 0; i < unique(y).size(); i++) {
      classifs[i] = i+1;
    }

    NumericMatrix cnt(classifs.size(), 2);
    for (i = 0; i < classifs.size(); i++) {
      cnt(i,0) = classifs(i);
      cnt(i,1) = 0;
    }
    for (i = 0; i < NN_distances.size(); i++) {
      cnt(NN_classif(i)-1,1) = cnt(NN_classif(i)-1,1) + 1/NN_distances(i);
    }

    int group;
    for (i = 0; i < cnt(_,1).size(); i++) {
      if (cnt(i,1) == max(cnt(_,1))) {
        group = cnt(i,0);
      }
    }

  return group;
}
```

Testing the Rcpp implementation using the inverse voting system

```
knn_inv(X, X0, y, 3)
```

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

Testing the functions after importing from the library

```
library('assignment1knn')
assignment1knn::knn_1(X, X0, y)
```

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

```
assignment1knn::knn_more(X, X0, y, 3)
```

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

```
assignment1knn::knn_inv(X, X0, y, 3)
```

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