Advanced Programming

RCPP ASSIGNMENT, PROGRAMMING NEAREST NEIGHBOUR IN C++

Errasti Domínguez, Nuria and Levenfeld Sabau, Gabriela

2023-12-07

Contents

Preparing the data

First of all, we must prepare the data we are going to work with.

```r
set.seed(123) # for reproducibility

# X contains the inputs as a matrix of real numbers
data("iris")
iris_shuffled <- iris[sample(nrow(iris)), ]

# X contains the input attributes (excluding the class)
X <- iris_shuffled[,-5]
# y contains the response variable (named medv, a numeric value)
y <- iris_shuffled[,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)
```

Preliminaries: my_knn_R

R code

Here is attached the knn function programmed in R language by the professor for the assignment that we will use during the task.

```
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 (just one instance)

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

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

  distance = sqrt(distance)

  closest_distance = distance
  closest_output = y[1]
  closest_neighbor = 1

  for(i in 2: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
}
```

Compile R code

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

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

Task 1: my_knn_c

The file for this first task can be found over the named: *task1_my_knn_c.cpp*.

Translate R code into C++

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

// [[Rcpp::export]]
int my_knn_c(NumericMatrix X, NumericVector X0, IntegerVector y){
  /*
   - X data matrix with input attributes. Each row represent an observation and columns
  are refer to the different variables (features) we are having into account.
   - y response variable values of instances in X. Vector that contain the
   response varibles (3 possible values: setosa, versicolor or virginica)
  - X0 vector of input attributes for prediction (just one instance)
  */

  int nrows = X.nrow(); //number of observations
  int ncols = X.ncol(); //features or attributes that are being considered

  // Initialize variables for distance calculation
  double distance = 0;
  double difference = 0;
  int j;

  /* Compute Euclidean distance between the input instance X0 and the first row
  (observation) of X in order to initialize the parameter distance. */
  for (j=0; j<ncols; ++j){
    difference = X(1,j)-X0[j];
    distance = distance + difference * difference;
  }
  distance = sqrt(distance);

  /* Next step is to compute every distance between X0 and the observations of
   matrix X and store the lowest value in the closest_distance parameter. */
  double closest_distance = distance;
  double closest_output = y[1];
  int closest_neighbor = 1;
  int i;

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

    distance = sqrt(distance);

      /* Finally, we will return the output value of the closest neighbor based
```

```
       on the shortest distance. */
      if(distance < closest_distance){
        closest_distance = distance;
        closest_output = y[i];
        closest_neighbor = i;
      }
  }
  return closest_output;
}
```

Compile C++ code with sourceCpp

```
library(Rcpp)
sourceCpp("C:/Users/gabri/advancedProg/Assignment1/task1_my_knn_c.cpp")
print(my_knn_c(X, X0, y))
```

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

We can observe that same result as before is achieved.

Verify Results against FNN/class knn

```
# Using class:knn to predict point X0
library(class)
print(class::knn(X, X0, y, k=1))
```

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

As we can observe the result is the same as with R code and C++ code, 2.

Use microbenchmark to compare performance

```
library(microbenchmark)
pander(microbenchmark(my_knn_c(X, X0, y), my_knn_R(X, X0, y), knn(X, X0, y, k=1),
               times = 1000))
```

Table 1: Unit: microseconds

expr	min	lq	mean	median	uq	max	neval
my_knn_c(X, X0, y)	1.8	2.4	5.056	3.701	4.201	1150	1000
my_knn_R(X, X0, y)	317.6	350.6	412.5	362.8	396.9	4904	1000
knn(X, X0, y, k = 1)	84.8	98.7	115.9	105.1	118.6	341.4	1000

This information refers to the execution time of each knn function (R, C++ and class) expressed in microseconds. We can verify that the C++ implementation is much faster than the other two options, with a mean time of just 4.225 microseconds (which is significantly faster). To do so, we have set up the evaluation parameter as 1000 times, so we can estimate the execution time in a more reliable way.

Disclaimer

From this point onwards, we have created and loaded our own package (*rcppAssignment.Rproj*) which contains all the needed functions. The C++ code for all functions can be found inside rcppAssignment/src/my_knn_c_function.cpp

Task 2: my_knn_c_euclidean

C++ code

In order to do this task we have created two different functions:

- **my_knn_c_euclidean**; it computes the euclidean distance between two numeric vectors.
- **my_knn_c_task2**; a specific function to perform the knn algorithm for Task 2. It calculates the euclidean distance between the observation X0 and each row of the X matrix using the my_knn_c_euclidean function. Finally, it returns the predicted class for X0.

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

// [[Rcpp::export]]
double my_knn_c_euclidean(NumericVector X, NumericVector X0){
  int ncols = X.size(); //number of columns
  double distance = 0; //initialize the variable
  // Compute the euclidean distance
  for (int j=0; j<ncols; ++j){
    double difference = X(j)-X0[j];
    distance += difference * difference;
  }
  return sqrt(distance);
}

// [[Rcpp::export]]
int my_knn_c_task2(NumericMatrix X, NumericVector X0, IntegerVector y){
  int nrows = X.nrow(); //number of observations
  // Initialize the variables
  double closest_distance = my_knn_c_euclidean(X(0,_), X0);
  double closest_output = y[1];
  int closest_neighbor = 1;
  /* Compute the euclidean distance from each observation to the input instance
  and choose the closest one */
  for (int i=1; i<nrows; ++i){
    double distance = my_knn_c_euclidean(X(i,_), X0);
    if(distance < closest_distance){
      closest_distance = distance;
      closest_output = y[i];
      closest_neighbor = i;
    }
  }
  return closest_output;
}
```

Code compilation

```
print(my_knn_c_task2(X, X0, y))
```

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

Code works correctly, the same result as before is obtained.

Task 3: my_knn_c_minkowsky

C++ code

In this section, the procedure followed was similar as before, where two new function have been created:

- **my_knn_c_minkowsky**; it computes the Minkowsky distance between two numeric vectors. This function considers all possible options for the parameter p, including positive, negative, and zero values.
- **my_knn_c_task3**; specific function to perform the knn algorithm for Task 3.

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

static bool warningDisplayed = false;


// [[Rcpp::export]]
double my_knn_c_minkowsky(NumericVector X, NumericVector X0, double p){
  // Computes the minkowsky distance
  int ncols = X.size(); //number of columns
  //initialize the variables
  double distance = 0;
  double difference = 0;
  int j;
  // Separate the different cases according to the value of p
  if (p>0){
    for(int j=0; j<ncols; ++j){
      difference = abs(X[j] - X0[j]);
      distance += pow(difference,p);
    }
    distance = pow(distance, 1/p);
  }  else if (p==0){
    if (!warningDisplayed) {
      Rcpp::Rcerr << "Warning: p is 0; consider handling this case appropriately." << std::endl;
      warningDisplayed = true;
    } distance = NA_REAL;
  } else{
    for(j=0; j<ncols; ++j){
      difference = abs(X[j] - X0[j]);
      distance = std::max(distance, difference);
    }
  }
  return distance;
}


// [[Rcpp::export]]
int my_knn_c_task3(NumericMatrix X, NumericVector X0, IntegerVector y, double p){
  int nrows = X.nrow(); //number of observations
  // Initialize the variables
  double closest_distance = R_PosInf;
  double closest_output = NA_INTEGER;
```

```
  int closest_neighbor = -1;
  // Compute the distance from each observation to the input instance X0
  for (int i=1; i<nrows; ++i){
    double distance = my_knn_c_minkowsky(X(i,_), X0, p);
    // Ignore the case of p=0
    if (distance != NA_REAL) {
      if (distance < closest_distance) {
        closest_distance = distance;
        closest_output = y[i];
        closest_neighbor = i;
      }
    }
  }
  /* Finally, we will return the output value of the closest neighbor based
  on the shortest distance */
  return closest_output;
}
```

Code compilation

To verify the code works properly, we are going to consider every possible case for parameter p.

1. **p positive**

```
print(my_knn_c_task3(X, X0, y, 2))
```

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

```
my_knn_c_minkowsky(X[1,], X0, 2)
```

```
## [1] 3.775248
```

```
my_knn_c_euclidean(X[1,], X0)
```

```
## [1] 3.775248
```

For this case, we set up $p = 2$. Hence, we are computing the Euclidean distance and the result is the same as our previous computation.

To further validate our code and ensure consistency between the different distance metrics we are working with, we can specifically compute the Minkowsky distance for the first observation in our dataset. And compare it with the Euclidean distance which will lead us into the same results.

For simplicity, we just focus on computing distances between X0 and the first observation in the dataset.

2. **p negative**

```
print(my_knn_c_task3(X, X0, y, -7))
```

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

In this case, negative p value is used to prove that knn algorithm still works.

3. **p equal to zero**

```
print(my_knn_c_task3(X, X0, y, 0))
```

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

Last possible situation is when $p = 0$, where a warning message will be raised indicating that p can not accept this value.

Task 4: my_knn_c_tuningp

C++ code

- **my_knn_c_tuningp**; Function which performance a search over a set of possible p values provided by the user. And, after, with the selected value makes the prediction for X0. It also return the corresponding accuracy of the model.

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

static bool warningDisplayed = false;


// [[Rcpp::export]]
double my_knn_c_minkowsky(NumericVector X, NumericVector X0, double p){
  // Computes the Minkowsky distance
  int ncols = X.size(); //number of columns
  // Initialize the variables
  double distance = 0;
  double difference = 0;
  int j;
  // Separate the different cases according to the value of p
  if (p>0){
    for(int j=0; j<ncols; ++j){
      difference = abs(X[j] - X0[j]);
      distance += pow(difference,p);
    }
    distance = pow(distance, 1/p);
  }  else if (p==0){
    if (!warningDisplayed) {
      Rcpp::Rcerr << "Warning: p is 0; consider handling this case appropriately." << std::endl;
      warningDisplayed = true;
    } distance= NA_REAL;
  } else{
    for(j=0; j<ncols; ++j){
      difference = abs(X[j] - X0[j]);
      distance = std::max(distance, difference);
    }
  }
  return distance;
}


// [[Rcpp::export]]
int my_knn_c_task3(NumericMatrix X, NumericVector X0, IntegerVector y, double p){
  int nrows = X.nrow(); //number of observations
  // Initialize the variables
  double closest_distance = R_PosInf;
  double closest_output = NA_INTEGER;
  int closest_neighbor = -1;
  // Compute the distance from each observation to the input instance X0
  for (int i=1; i<nrows; ++i){
```

```
    double distance = my_knn_c_minkowsky(X(i,_), X0, p);
    // Ignore the case of p=0
    if (distance != NA_REAL) {
      if (distance < closest_distance) {
        closest_distance = distance;
        closest_output = y[i];
        closest_neighbor = i;
      }
    }
  }
  /* Finally, we will return the output value of the closest neighbor based
   on the shortest distance */
  return closest_output;
}

// [[Rcpp::export]]
List my_knn_c_tuningp(NumericMatrix X, NumericVector X0, IntegerVector y,
                      NumericVector possible_p){

  int nrows = X.nrow();
  // Calculate the number of observations for the training set
  int train_size = floor((2.0 / 3.0) * nrows);

  // Split dataset into training (2/3) and validation (1/3)
  NumericMatrix X_train = X(Range(0, train_size-1), _);
  NumericMatrix X_val = X(Range(train_size, nrows-1), _);
  IntegerVector y_train = y[Range(0, train_size-1)];
  IntegerVector y_val = y[Range(train_size, nrows-1)];

  // Initialize the variables
  double best_accuracy = 0.0;
  double best_p = 0.0;

  // Iterate for all possible_p in order to find the best value
  for (int i=0; i<possible_p.size(); ++i){
    double p  = possible_p[i];
    int correct_predictions = 0;
    for (int j = 0; j < nrows - train_size; ++j) {
      // Reused code from task3
      int closest_output = my_knn_c_task3(X_train, X_val(j, _), y_train, p);
      // Check if prediction is correct
      if (closest_output == y_val[j]) {
        correct_predictions++;
      }
    }

    // Choose the best_p in relation to the accuracy
    double accuracy = static_cast<double>(correct_predictions)/(nrows-train_size);
    if (accuracy > best_accuracy) {
      best_accuracy = accuracy;
      best_p = p;
    }
  }
```

```
  // Return: final prediction, optimal value of p and the accuracy
  List final_params;
  final_params["best_p"] = best_p;
  // Use the best_p to make the final prediction for X0
  final_params["final_output"] = my_knn_c_task3(X, X0, y, best_p);
  final_params["best_accuracy"] = best_accuracy;
  return final_params;
}
```

Code compilation

In order to show the functionality of **my__knn__c__tuningp** function, two demos are executed.

- **Demo 1**:

```
possible_p <- c(0.5,6,-1,0)
print(my_knn_c_tuningp(X, X0, y, possible_p))
```

```
## $best_p
## [1] 6
##
## $final_output
## [1] 2
##
## $best_accuracy
## [1] 0.96
```

- **Demo 2**:

```
possible_p <- c(5,2,7,6)
print(my_knn_c_tuningp(X, X0, y, possible_p))
```

```
## $best_p
## [1] 5
##
## $final_output
## [1] 2
##
## $best_accuracy
## [1] 0.96
```

In both demos, the model reaches an accuracy of 96% and the selected prediction class is the second one. However, the best p value is different.

For this exercise it is worth mentioning that for positive values greater than 1, the model obtains always the same accuracy (96%) and, as a consequence, it keeps the value that occupies the first position in the vector (possible_p) created by the user.