PROGRAMMING KNN WITH RCCP

REPORT

JAVIER FONG GUZMAN & MENGTING CHEN

We have created a package with three functions, that compute different variation of a useful machine learning method called K-nearest neighbor.

Function: my_knn_cpp

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

  double closest_distance = 99999999;
  int closest_output = -1, closest_neighbor = -1;
  double distance = 0, difference = 0;

  for (int i = 0; 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 receives three parameters:

- X: a numeric matrix of the existing observation with all the feature, except the response variable. Every column is a different variable and every row is different observation.
- X0: A new observation for classification.
- Y: the response variable (class).

This function calculates the Euclidean distance between X0 and all the observation on X row by row, and compare the new distance with the shortest one until now and choose

the shortest distance. At the end, it is left with the class of the nearest neighbor based on the shortest distance.

In this case, we have compiled the C++ code with *sourceCpp*. And we compare our function with the knn that belongs to library *FNN*, obtaining the same result. This new observation belongs to the second class.

```
> sourceCpp("cpp_functions.cpp")
> my_knn_cpp(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,] 0.05616121
Levels: 2
```

Using the library microbenchmark in order to determine whether the C++ version is faster than: the R version of the code and the knn of FNN.

```
> microbenchmark(
+   my_knn_R(X, X0, y)
+   ,my_knn_cpp(X,X0,y)
+   ,FNN::knn(X, matrix(X0, nrow = 1), y, k=1)
+ )
Unit: microseconds
                                      expr     min       lq      mean   median       uq      max neval
                        my_knn_R(X, X0, y) 595.584 602.3545 641.00936 607.3545 619.0010 3552.584   100
                       my_knn_cpp(X, X0, y)   2.209   2.6670  11.30693   3.1255   3.5635  804.084   100
 FNN::knn(X, matrix(X0, nrow = 1), y, k = 1) 211.001 217.8135 233.94354 223.9590 235.6045  509.751   100
>
```

As we can observe, the performance of the my_knn_cpp function is almost 58 (mean run time) time faster than the same function in R, and 21 time faster than the function of the FNN library.

Function: my_knn_mink_cpp

```
double my_knn_mink_cpp(NumericMatrix X, NumericVector X0, NumericVector y, double p) {
  int nrows = X.nrow();
  int ncols = X.ncol();

  double closest_distance = 99999999;
  int closest_output = -1, closest_neighbor = -1;
  double distance = 0, difference = 0;

  for (int i = 0; i < nrows; i++) {
    distance = 0;
    for (int j = 0; j < ncols; j++) {
      difference = abs(X(i,j) - X0[j]);
      if (p > 0) {
        distance = distance + pow(difference, p);
      } else {
        if (difference > distance) {
          distance = difference;
        }}}
    if (p > 0) {
      distance = pow(distance, 1/p);
    }

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

  return closest_output;
}
```

This function has the same logic as the first function, but with two changes. It uses the Minkowsky distance instead of the Euclidean distance and has one more parameter call *p*, if positive, then it is exponent *p*, and if negative, then it means $L\infty$.

To show that our function calculates the Minkowsky distance, we change the function return from closest_output to closest_distance, which give us the smallest Minkowsky distance.

```
> #KNN with Euclidean Distance
> my_knn_cpp(X,X0,y)
[1] 0.2061553

> #KNN with Minkowsky Distance (p = 2, should be the same as the Euclidean)
> my_knn_mink_cpp(X,X0,y,2)
[1] 0.2061553

> #KNN with Minkowsky Distance (p = 4)
> my_knn_mink_cpp(X,X0,y,4)
[1] 0.1630195

> #KNN with Minkowsky Distance (p = -2, which uses a different formula)
> my_knn_mink_cpp(X,X0,y,-2)
[1] 0.15
```

Function: my_knn_mink_scaled_cpp

```cpp
// [[Rcpp::export]]
double my_knn_mink_scaled_cpp(NumericMatrix X, NumericVector X0, NumericVector y, double p, double s) {
  NumericMatrix newX = X;
  NumericVector newX0 = X0;

  int nrows = X.nrow();
  int ncols = X.ncol();

  if (s == 0) {
    for (int i = 0; i < ncols; i++) {
      double mean_val = mean(X(_,i));
      double sd_val = sd(X(_,i));
      for (int j = 0; j < nrows; j++) {
        newX(j,i) = (newX(j,i) - mean_val)/sd_val;
      }
      newX0[i] = (newX0[i] - mean_val)/ sd_val;
    }
  } else if (s == 1) {
    for (int i = 0; i < ncols; i++) {
      double max_val = max(X(_,i));
      double min_val = min(X(_,i));
      for (int j = 0; j < nrows; j++) {
        newX(j,i) = (newX(j,i) - min_val)/(max_val - min_val);
      }
      newX0[i] = (newX0[i] - min_val)/(max_val - min_val);
    }
  }
  return(my_knn_mink_cpp(newX, newX0, y, p));
}
```

This function uses my_knn_mink_cpp function in their function. But instead using the original data matrix, we have another parameter *s*, if *s* = 0 then we standardized our data matrix (mean = 0 and standard deviation = 1) and if *s* = 1, we normalized the data matrix to range 0-1.

To check that in our function the data is standardized when we use *s* = 0 and normalized when *s* = 1, we change the function return from closest_output to newX, which give us the new data matrix. As we can see in the following example:

```
> #Call of my_knn_mink_scaled_cpp with s = 1 (Normalized data)
> newx = my_knn_mink_scaled_cpp(X,X0,y,4,1)
> newx %>% head()
     Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]   0.22222222   0.6250000   0.06779661  0.04166667
[2,]   0.16666667   0.4166667   0.06779661  0.04166667
[3,]   0.11111111   0.5000000   0.05084746  0.04166667
[4,]   0.08333333   0.4583333   0.08474576  0.04166667
[5,]   0.19444444   0.6666667   0.06779661  0.04166667
[6,]   0.30555556   0.7916667   0.11864407  0.12500000
> #If the data is correctly normalized, all the values should be between 0-1
> summary(newx)
  Sepal.Length     Sepal.Width      Petal.Length     Petal.Width
 Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   Min.   :0.00000
 1st Qu.:0.2222   1st Qu.:0.3333   1st Qu.:0.1017   1st Qu.:0.08333
 Median :0.4167   Median :0.4167   Median :0.5678   Median :0.50000
 Mean   :0.4287   Mean   :0.4406   Mean   :0.4675   Mean   :0.45806
 3rd Qu.:0.5833   3rd Qu.:0.5417   3rd Qu.:0.6949   3rd Qu.:0.70833
 Max.   :1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :1.00000
>
```

```
> #Call of my_knn_mink_scaled_cpp with s = 0 (Standarized data)
> newx = my_knn_mink_scaled_cpp(X,X0,y,4,0)
> newx %>% head()
     Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]   -0.8976739  1.01560199    -1.335752   -1.311052
[2,]   -1.1392005 -0.13153881    -1.335752   -1.311052
[3,]   -1.3807271  0.32731751    -1.392399   -1.311052
[4,]   -1.5014904  0.09788935    -1.279104   -1.311052
[5,]   -1.0184372  1.24503015    -1.335752   -1.311052
[6,]   -0.5353840  1.93331463    -1.165809   -1.048667
> #If the data is correctly standardized, all the means should be 0 and all the sd should be 1
> round(colMeans(newx),12)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
           0            0            0            0
> colSds(newx)
[1] 1 1 1 1
```