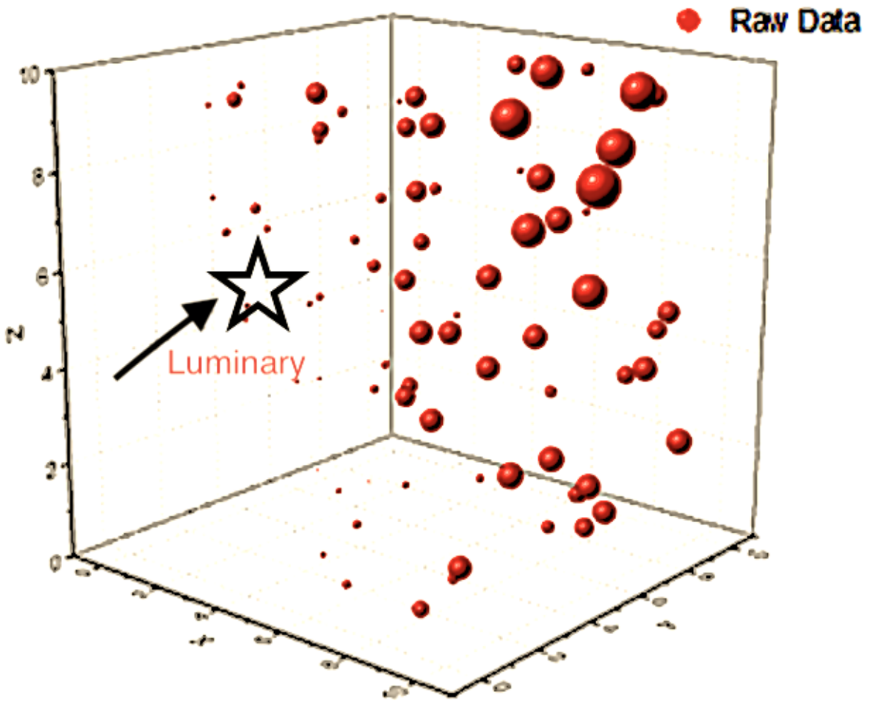


Implementation of K Nearest Neighbor in C++

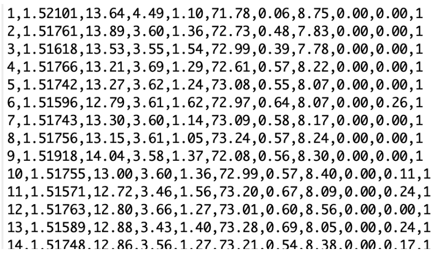
CS106B Final Project, Nov 2020

Minakshi Mukherjee

Stanford University

1. Problem description

Hubble Space Telescope has identified yet another new planet, NASA scientists has named it Luminary and they want to find out the nearest neighbors of Luminary in Solar System. There are different types of planets in planetary system, scientists measured different features of 196 planetary objects and labeled them into six categories(Label 1,Label 2, Label 3, Label 5, Label 6, Label 7). Here is a snippet of the data, where the last column shows the Label category. We need a simple algorithm to find out the K nearest neighbors



```
1,1.52101,13.64,4.49,1.10,71.78,0.06,8.75,0.00,0.00,1
2,1.51761,13.89,3.60,1.36,72.73,0.48,7.83,0.00,0.00,1
3,1.51618,13.53,3.55,1.54,72.99,0.39,7.78,0.00,0.00,1
4,1.51766,13.21,3.69,1.29,72.61,0.57,8.22,0.00,0.00,1
5,1.51742,13.27,3.62,1.24,73.08,0.55,8.07,0.00,0.00,1
6,1.51596,12.79,3.61,1.62,72.97,0.64,8.07,0.00,0.26,1
7,1.51743,13.30,3.60,1.14,73.09,0.58,8.17,0.00,0.00,1
8,1.51756,13.15,3.61,1.05,73.24,0.57,8.24,0.00,0.00,1
9,1.51918,14.04,3.58,1.37,72.08,0.56,8.30,0.00,0.00,1
10,1.51755,13.00,3.60,1.36,72.99,0.57,8.40,0.00,0.11,1
11,1.51571,12.72,3.46,1.56,73.20,0.67,8.09,0.00,0.24,1
12,1.51763,12.80,3.66,1.27,73.01,0.60,8.56,0.00,0.00,1
13,1.51589,12.88,3.43,1.40,73.28,0.69,8.05,0.00,0.24,1
14,1.51748,12.86,3.56,1.27,73.21,0.54,8.38,0.00,0.17,1
```

of Luminary and which Label(out of the six labels), Luminary primarily belongs to. We can build the algorithm on the dataset of 196 planetary objects and verify our algorithm on another dataset with 14 planetary objects with known Labels, which is not used to train the algorithm. All the cool ADTs learned in CS106B will be very handy to solve this problem.

2. Solutions

2.1 Function: buildDataGrid

The function ‘buildDataGrid‘ fulfills the following requirements:
1. If no valid file is available based on the input filename, it errors out.
2. It resizes the grid first based on the dimension from the input file to avoid index out of bound error.
3. It reads all the data from the input file and builds a multidimensional grid object.

2.2 Function: pointToDistance

The function ‘pointToDistance‘ fulfills the following requirements:
1. It takes a grid object and a vector point and calls another helper function ‘square2D‘ to calculate Euclidean distance.
2. Two variables are passed as reference to build a ‘distanceMap‘ and to keep the distances in a vector ‘allDistances‘.

2.3 Function: kNearestClassify

The function ‘kNearestClassify‘ fulfills the following requirements:
1. If the input value of K exceeds the number of rows in the training grid, it errors out.

```
Grid<double> buildDataGrid(string& filename){
    ifstream in;
    if (!openFile(in, filename))
        error("cannot open file named "+ filename);

    Vector<string> lines;
    Grid<double> resultGrid;
    readEntireFile(in, lines);
    int numRows = lines.size();
    int numCols = stringSplit(lines[0], ",").size();
    // resize the grid with proper dimensions before populating it from the file
    resultGrid.resize(numRows, numCols);
    for ( int i = 0; i < lines.size() ; i++){
        string trainRow = lines[i];
        Vector<string> row = stringSplit(trainRow, ",");
        for (int c = 0; c <  row.size()  ; c++){
            resultGrid[i][c] = stringToReal(row[c]);
        }
    }
    return resultGrid;
}
```

```
void pointToDistance(Grid<double>& grid, Vector<double> point,
                     Map<int, Set<int>>& distanceMap, Vector<int>& allDistances){
    //Map<int, Set<double>> distanceMap;
    for (int r = 0; r < grid.numRows() ; r++){
        double intermediate = 0.0;
        for (int c = 0; c < grid.numCols() - 1; c++) {
            // Calculate Euclidean distance between the test point and all
            // training data points
            intermediate += square2D(grid[r][c], point[c]);
        }
        double finalDis = sqrt(intermediate);
        // Add the distances to a vector
        allDistances.add(finalDis);
       // Add the distances to a map with map key denoting label
        distanceMap[grid[r][grid.numCols()-1]].add(finalDis);
    }
}
```

2. For each point in test grid, it calls the function ‘pointToDistance‘ to build Map object ‘distanceMap‘ and Vector ‘vecDistance‘.
3. It uses Vector sort to sort the array and takes the K smallest elements. 4. It calls the function getLabel to get the Label types(1,2,3,5,6,7).
5. It outputs the predicted label and the actual label for each point in test grid.

```
void kNearestClassify(Grid<double>& grid, Grid<double> testGrid, int k){
    if (k > grid.numRows()){
        error(" k is bigger than grid row size");
    }
    for (int r = 0; r < testGrid.numRows(); r++){
        Vector<double> testRow;
        for (int c = 0; c < testGrid.numCols() - 1; c++){
            testRow.add(testGrid[r][c]);
        }
        Vector<int> vecDistance;
        Vector<int> final;
        Map<int, Set<int>> distanceMap;
        pointToDistance(grid, testRow, distanceMap, vecDistance);
        // sort the array
        vecDistance.sort();
        // Take top k nearest
        for (int i =0; i< k ; i++){
            final.add(vecDistance[i]);
        }
        cout << r << ":-> " << "predicted-> " << getLabel(distanceMap, final) << " Actual-> " <<
                testGrid[r][testGrid.numCols() - 1] << endl;

    }
}
```

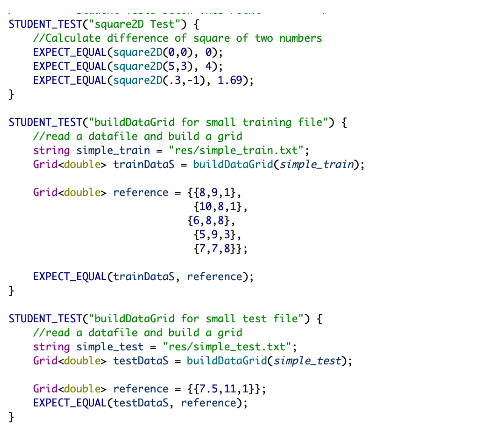
2.4 Function: kNearestClassifyPQ

This function is very similar to ‘kNearestClassify‘, except it uses Priority Queue to enqueue and dequeue the K nearest neighbors. Here is a snippet of the function that is different from ‘kNearestClassify‘ function.

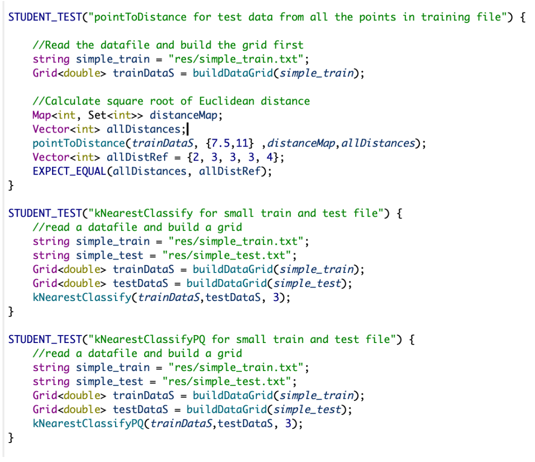
```
Vector<int> vecDistance;
Vector<int> final;
Map<int, Set<int>> distanceMap;
pointToDistance(grid, testRow, distanceMap, vecDistance);
PriorityQueue<int> pq;
// Once we enqueue priority queue is already sorted.
for (int i =0; i < vecDistance.size(); i++){
    pq.enqueue(vecDistance[i], vecDistance[i]);
}
// Take top k nearest
for (int i =0; i< k; i++){
    final.add(pq.dequeue());
}
cout << r << ":-> " << "predicted-> " << getLabel(distanceMap, final) << " Actual-> " <<
        testGrid[r][testGrid.numCols() - 1] << endl;
```

2.5 Test Cases

Following test cases are used to test the individual functions.



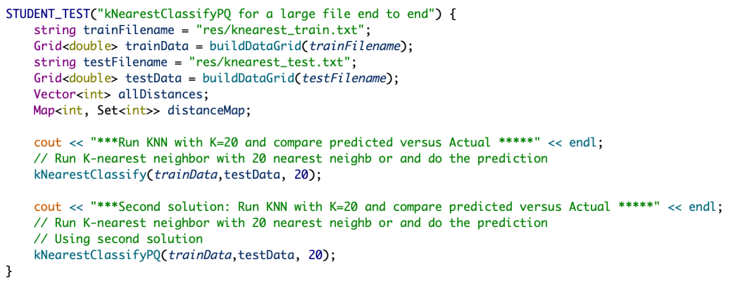
```
STUDENT_TEST("square2D Test") {
    //Calculate difference of square of two numbers
    EXPECT_EQUAL(square2D(0,0), 0);
    EXPECT_EQUAL(square2D(5,3), 4);
    EXPECT_EQUAL(square2D(.3,-1), 1.69);
}

STUDENT_TEST("buildDataGrid for small training file") {
    //read a datafile and build a grid
    string simple_train = "res/simple_train.txt";
    Grid<double> trainDataS = buildDataGrid(simple_train);

    Grid<double> reference = {{8,9,1},
                               {10,8,1},
                              {6,8,8},
                               {5,9,3},
                               {7,7,8}};

    EXPECT_EQUAL(trainDataS, reference);
}

STUDENT_TEST("buildDataGrid for small test file") {
    //read a datafile and build a grid
    string simple_test = "res/simple_test.txt";
    Grid<double> testDataS = buildDataGrid(simple_test);

    Grid<double> reference = {{7.5,11,1}};
    EXPECT_EQUAL(testDataS, reference);
}
```



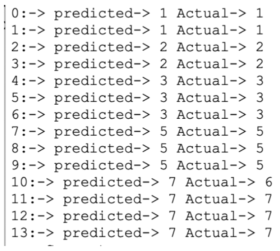
```
STUDENT_TEST("pointToDistance for test data from all the points in training file") {

    //Read the datafile and build the grid first
    string simple_train = "res/simple_train.txt";
    Grid<double> trainDataS = buildDataGrid(simple_train);

    //Calculate square root of Euclidean distance
    Map<int, Set<int>> distanceMap;
    Vector<int> allDistances;
    pointToDistance(trainDataS, {7.5,11} ,distanceMap,allDistances);
    Vector<int> allDistRef = {2, 3, 3, 3, 4};
    EXPECT_EQUAL(allDistances, allDistRef);
}

STUDENT_TEST("kNearestClassify for small train and test file") {
    //read a datafile and build a grid
    string simple_train = "res/simple_train.txt";
    string simple_test = "res/simple_test.txt";
    Grid<double> trainDataS = buildDataGrid(simple_train);
    Grid<double> testDataS = buildDataGrid(simple_test);
    kNearestClassify(trainDataS,testDataS, 3);
}

STUDENT_TEST("kNearestClassifyPQ for small train and test file") {
    //read a datafile and build a grid
    string simple_train = "res/simple_train.txt";
    string simple_test = "res/simple_test.txt";
    Grid<double> trainDataS = buildDataGrid(simple_train);
    Grid<double> testDataS = buildDataGrid(simple_test);
    kNearestClassifyPQ(trainDataS,testDataS, 3);
}
```

2.6 End to end run on bigger files

This TEST shows the complete implementation of all the functions to solve the problem that NASA scientists wanted to get a rough idea on. It will iterate on all the multidimensional test points from a validation dataset with known labels and calculates the Euclidean distance of all those test points from the training grid. It then finds K nearest neighbors, label them and find the majority vote label; at the end it verifies the predicted label vs actual label to see how accurate this algorithm is. This algorithm will help to predict the label on the new planet Luminary.



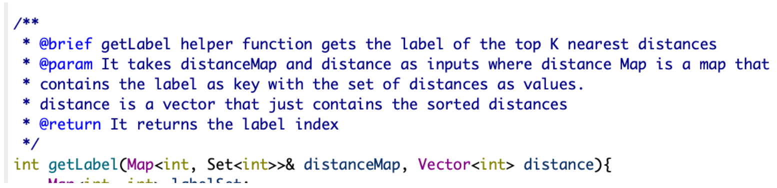
```
STUDENT_TEST("kNearestClassifyPQ for a large file end to end") {
    string trainFilename = "res/knearest_train.txt";
    Grid<double> trainData = buildDataGrid(trainFilename);
    string testFilename = "res/knearest_test.txt";
    Grid<double> testData = buildDataGrid(testFilename);
    Vector<int> allDistances;
    Map<int, Set<int>> distanceMap;

    cout << "***Run KNN with K=20 and compare predicted versus Actual *****" << endl;
    // Run K-nearest neighbor with 20 nearest neighb or and do the prediction
    kNearestClassify(trainData,testData, 20);

    cout << "***Second solution: Run KNN with K=20 and compare predicted versus Actual *****" << endl;
    // Run K-nearest neighbor with 20 nearest neighb or and do the prediction
    // Using second solution
    kNearestClassifyPQ(trainData,testData, 20);
}
```



```
0:-> predicted-> 1 Actual-> 1
1:-> predicted-> 1 Actual-> 1
2:-> predicted-> 2 Actual-> 2
3:-> predicted-> 2 Actual-> 2
4:-> predicted-> 3 Actual-> 3
5:-> predicted-> 3 Actual-> 3
6:-> predicted-> 3 Actual-> 3
7:-> predicted-> 5 Actual-> 5
8:-> predicted-> 5 Actual-> 5
9:-> predicted-> 5 Actual-> 5
10:-> predicted-> 7 Actual-> 6
11:-> predicted-> 7 Actual-> 7
12:-> predicted-> 7 Actual-> 7
13:-> predicted-> 7 Actual-> 7
```

2.7 getlabel helper function for final labeling

The program uses the helper function ‘getLabel‘ to put a label type on the distance values.



```
/**
 * @brief getLabel helper function gets the label of the top K nearest distances
 * @param It takes distanceMap and distance as inputs where distance Map is a map that
 * contains the label as key with the set of distances as values.
 * distance is a vector that just contains the sorted distances
 * @return It returns the label index
 */
int getLabel(Map<int, Set<int>>& distanceMap, Vector<int> distance){
```

2.8 Reflect on TEST cases

Each of the above TEST cases were extremely useful for unit testing.
It helped to achieve good decomposition and verification of Mathematical formulas.
It helped to test multiple solutions and come up with a neat and succinct strategy to move forward.

3. Two alternatives for sorting: Vector sort vs use of Priority Queue

1. Since, we are using integers, enqueue in priority queue is Big O(n) and dequeue is Big O(1), so sorting takes a constant time, this is faster for our algorithm as we need to provide K nearest neighbors for all the test points, so once, the distances are enqueued in a priority queue, dequeuing them in a sorted manner is O(1)

2. For vectors, insert is O(n), but sorting is O(nlogn).
3. Hence use of Priority Queue improvers performance for large dataset.

4. Problem Motivation

K nearest neighbor algorithm has a wide variety of real world applications, as this is a very simple and useful algorithm. In the past, I have used Python ML libraries to get the nearest neighbor and label classification, but I found that Python libraries run much slower as the value of K increases. I realized, it will be very fulfilling if I can implement a simple version of KNN using the knowledge learnt in CS106B.

4.1 Concept Coverage

Playing and iterating with different container and collection types(Vector, Grid, Map and Set)
Priority Queue, file read, complex nesting and manipulation through multiple data structure
Passing variables by reference

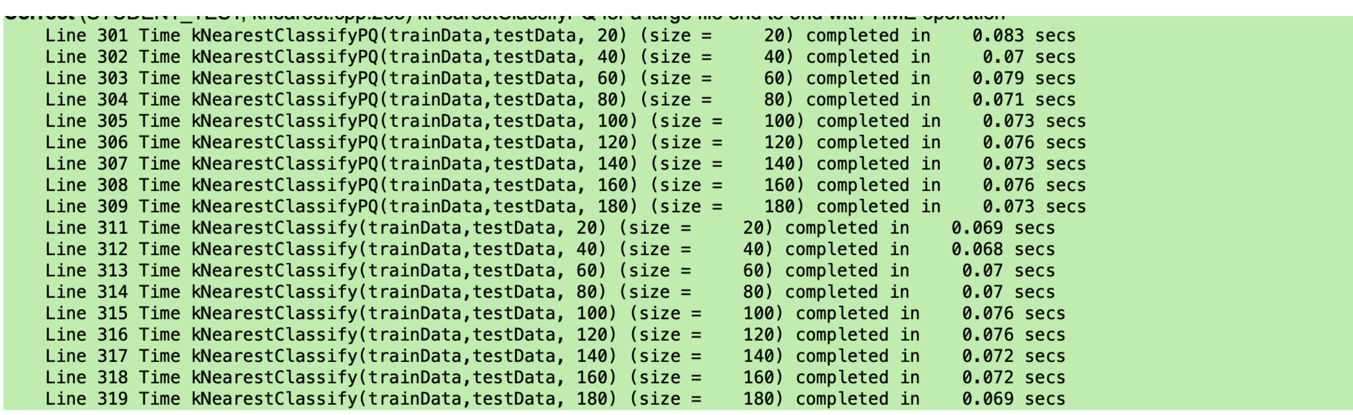
4.2 Personal Significance

I wanted to enrich my understanding in C++ by implementing an algorithm end-to-end where I apply good decomposition and appropriate ADT. I picked KNN algorithm as it is simple yet powerful. This project enhanced my programming skill and understanding as I progressed through writing multiple functions to solve a problem in an elegant manner.

In addition to that, I find my implementation to be much faster than Python and there is still a lot of scope for improvement which I will explore further in future.

4.3 TIME OPERATION using different values of K

It shows that as we increase the values of K, the timing does not increase, so the C++ implementation almost runs in linear time.



```
    Line 301 Time kNearestClassifyPQ(trainData,testData, 20) (size =     20) completed in    0.083 secs
    Line 302 Time kNearestClassifyPQ(trainData,testData, 40) (size =     40) completed in     0.07 secs
    Line 303 Time kNearestClassifyPQ(trainData,testData, 60) (size =     60) completed in    0.079 secs
    Line 304 Time kNearestClassifyPQ(trainData,testData, 80) (size =     80) completed in    0.071 secs
    Line 305 Time kNearestClassifyPQ(trainData,testData, 100) (size =    100) completed in    0.073 secs
    Line 306 Time kNearestClassifyPQ(trainData,testData, 120) (size =    120) completed in    0.076 secs
    Line 307 Time kNearestClassifyPQ(trainData,testData, 140) (size =    140) completed in    0.073 secs
    Line 308 Time kNearestClassifyPQ(trainData,testData, 160) (size =    160) completed in    0.076 secs
    Line 309 Time kNearestClassifyPQ(trainData,testData, 180) (size =    180) completed in    0.073 secs
    Line 311 Time kNearestClassify(trainData,testData, 20) (size =     20) completed in    0.069 secs
    Line 312 Time kNearestClassify(trainData,testData, 40) (size =     40) completed in    0.068 secs
    Line 313 Time kNearestClassify(trainData,testData, 60) (size =     60) completed in     0.07 secs
    Line 314 Time kNearestClassify(trainData,testData, 80) (size =     80) completed in     0.07 secs
    Line 315 Time kNearestClassify(trainData,testData, 100) (size =    100) completed in    0.076 secs
    Line 316 Time kNearestClassify(trainData,testData, 120) (size =    120) completed in    0.076 secs
    Line 317 Time kNearestClassify(trainData,testData, 140) (size =    140) completed in    0.072 secs
    Line 318 Time kNearestClassify(trainData,testData, 160) (size =    160) completed in    0.072 secs
    Line 319 Time kNearestClassify(trainData,testData, 180) (size =    180) completed in    0.069 secs
```