Machine Learning (CS 419/519)
Assignment 02 (50 points)
Due 5:00 PM, Wednesday, 17 October 2018

In this assignment, you will implement the algorithm for constructing K-D trees (covered in lecture 10), and then find the neighbor sets for some data, along with the nearest neighbor in that set. Note that we will not be doing the more complex task of finding the nearest neighbor in the entire data-set, which involves back-tracking up the tree and using bounding-box information. Instead, we will simply be classifying new data according to the median values used in building the tree.

For full points, your program will work as follows:

1. (*5 pts.*) The program will execute from the command line, and handle all input/output using standard system IO (in Java, e.g., this is `System.in` and `System.out`, and in C++ you would use `cin` and `cout`, etc.).

   When the program is executed, it is given two command-line arguments, in order:

   (a) A data-file from which to read. You can assume the file exists in the same directory as the executing program.

   (b) A minimal set-size (this is the parameter S used by the algorithm to decide when to stop splitting data). You can assume this will be a positive integer value.

   Note that these arguments are supplied when we execute the code; unlike the prior program, they are not something that the program prompts the user for after it begins to run. As an example, in a Java context, we may execute the program as follows:

   ```
   java KD_Builder 5d.txt 10
   ```

   This would cause the program to build a tree using the 5-dimensional data set found in the file `5d.txt`, and splitting that data up until each set of points had 10 or fewer members.

   Note: The assignment folder contains a sample runs of the program so you can see what is expected in this step and later steps; your output should be much the same, if not identical.

2. (*44 pts.*) The program will then execute as follows:

   (a) The program will read in its data-set from the file given in the first command-line argument. Each data-set consists of:

      i. An integer value d that gives the dimensionality of the data.

      ii. One or more lines, each consisting of d floating-point values v, $0 \leq v < 1$, that define data-points. No two data-points will be identical, although some may have the same values on some dimensions.

   Sample data-files have been given for data with between 2 and 5 dimensions; your code should be able to handle any positive number of dimensions up to 100-dimensional data. (You can write your own code to produce such data-sets if you want to test this.)

   (b) Once your program has read in the data, it will construct a K-D tree, using the input parameter S to determine when it has divided the data into small enough subsets. When building this tree, you will follow the algorithm covered in lecture, placing elements that fall at or below a median value on the left-hand branch below a dividing node, and those that fall above the median on the right-hand branch.

Note 01: While you may assume that the user gives you a minimal set-size value that is positive ($S \geq 1$), that value may actually be *higher* than the number of data-points in your input set. When this happens, the algorithm produces a "tree" consisting of a single root-node, containing all of the data; your code should account for this possibility. An example of where this occurs is in the file `testRun_2d_small-0.txt`.

Note 02: Depending upon the data, and the value S, it is possible that all data-points in a given subset have the same value for the dimension δ that is being used to divide that subset. When this occurs, one of the sets X^- or X^+ will be empty; your code should account for this possibility, too.

(c) After constructing the tree, the program will prompt the user as to whether or not it should display a print-out of the final subsets of data, stored in the leaves of the tree. If the user chooses to display the leaves, each will be printed with the following information:

- The path of left (`L`) or right (`R`) branches that need to be followed to reach the corresponding leaf-node.
- The minimal and maximal points that define the bounding-box for all data in that set (that is the tightest points that define opposite corners of the region surrounding each data-point).
- A listing of each data-point in the set.

Leaves should be printed from left-most to right-most across the tree. See the samples of this output in the four `testRun_2d_small-x.txt` files.

(d) After printing the tree (or not, if the user decides not to do so), the program will prompt the user as to whether or not to use the tree to classify some data-set. If the user chooses to do so, the program will prompt them for the name of a file. Again, you can assume that the file exists in the same directory as the executing code, and that it contains data of the same dimensionality as that used to build the tree. The program will then take each data-point from the testing file, and find the root-node containing the subset of its possible nearest neighbors, by going left or right according to the median values.

When that subset is found, it will be printed out. The program will also find the member of that set that is closest to the given data-point, identifying that neighbor, and printing out the distance to it. (This distance is the usual Euclidean, i.e., the L^2 norm.) See the various sample runs for examples of how this ought to be displayed.

Note 01: Because some leaf-nodes may be empty, your code may find that a new data-point ends up with no nearest neighbors in its set. This must be handled, and indicated. An example of where this occurs is in the file `testRun_2d_small-3.txt`.

Note 02: While you can assume that there are no duplicate data-points in the input (otherwise some implementations and runs of the algorithm would never terminate correctly), it *is possible* for the testing data to contain a point identical to one found in the data used to build the tree. When this happens, a point will be identical in value to its nearest neighbor, and the distance between them will be 0. An example of where this occurs is in the file `testRun_2d_small-3.txt`.

3. (*1 pt.*) When done finding nearest neighbors for the testing data, the program should wish the user goodbye, since is always important to be polite.

You will hand in a single compressed archive containing the following:

1. Complete source-code for your program. This can be written in any language you choose, but should follow proper software engineering principles for said language, including meaningful comments and appropriate code style. Code should be in its own folder within the archive, to keep it organized and separate from the rest of the submission.

2. A `README` file that provides instructions for how the program can be run. It is expected that this will consist of the command-line instructions necessarily. If you have completed the final part of the assignment (the optional verbose mode), then your instructions should include the commands needed to activate that mode and see the tree print-out at the end.

Your code will be tested using input files distinct from the ones provided, although they will be in the same basic format.