Programmeren (FEB22012)

6. Exercise

Deadline for submission: 2012-10-12 23:59 CET

Introduction

In this assignment you will implement a binary tree structure, k-d tree, and use it for classifying flowers based on four attributes using the k nearest neighbor algorithm (k-NN). Note that the k in the two names have no relation with each other.

K-d trees are a space-partitioning data structure used to store k-dimensional data points. Every non-leaf node of the tree represents a hyperplane cutting the space in two parts (i.e. in two half-spaces). The leaf nodes represent the partitioned areas in space. For example, a k-d tree with a root node and two children would split the space in 2 parts based on a single attribute. For more information and graphical representation of a k-d tree, see `http://en.wikipedia.org/wiki/K-d_tree`.

In this exercise we will use the k nearest neighbor (k-NN) algorithm for classifying flowers based on their sepal length, sepal width, petal length, and petal width in three species: setosa, versicolor, and virginica. K-NN classifies unseen data points by looking at the points in the *training set*, that is, a set of flowers for which we know the species already. Based on the euclidean distances between the training points and the new point, k nearest ones are located. The new point (flower) will be classified to be of the species the majority of these k points are of.

A naive solution for finding a closest point would be to compute the distance from the new point to each point in the training data set. If you have 100 new points to classify and a data set of 1000 points, you would have to do $100 \times 1000 = 100000$ distance computations. With a k-d tree this can be reduced (on average) to $100 \times \log 1000 = 996$.

Exercise part 1: implement a k-d tree

A k-d tree is created by recursively splitting on each dimension of the data. So for example, at the root node (Point) of the tree, all points on the left will have values smaller and all all points on the right will have values larger than the root (for the first dimension). There are several ways to decide on the splitting value. The method you are going to implement is to use the median of the training data set to choose the optimal cut-off value for the splits, leading to reasonably balanced trees.

Creating a k-d tree

First, you should implement an algorithm for constructing a k-d tree. The pseudo-code for this part of the assignment is given in Algorithm 1. In the end, what you should have (at least), is one function called *kdtree*, taking one input: the data set matrix D (where the rows represent the points and the columns represent the variables). The algorithm starts with dimension 1 (the first column) and sorts the matrix on this column. Next, the median index is determined. You should implement this in such a way that for a length of one, the first element is chosen as the median, for a length of two, the second element, for a length of three, also the second element, for a length four, the third element, etc. Then, the next axis for splitting should be selected. This is done by shuffling the axis. This means if you have four dimensions (columns), you first start with the first, then second, then third, then forth, then back to first, etc. Finally, the return variable is initialized. You should make a class Point with necessary fields. On lines 8 and 9 the recursion takes places. Here we build in the same way the tree for the left and right child of our root node with data up until and after from the median index, respectively.

Algorithm 1 Creating a k-d tree

Input: a $n \times k$ data set D, where n is the number of points and k the dimensionality (i.e., number of variables)
Input: $axisIndex$, index of the axis to do the split (default: 1)
Output: node ret which represents the root of the k-d tree
1: sort matrix D on column $axisIndex$
2: determine $medianIndex$ for column $axisIndex$ in D
3: determine $nextAxis$
4: initialize empty point ret
5: $ret.axisSplit = axisIndex$
6: $ret.val =$ the $medianIndex^{\text{th}}$ row of D
7: $ret.left =$ build recursively tree with input $D(1 : (medianIndex - 1), :)$ and $axisIndex = nextAxis$
8: $ret.right =$ build recursively tree with input $D((medianIndex + 1) : end, :)$ and $axisIndex = nextAxis$

Searching in a k-d tree

Now that you are able to create a kd-tree, you need to implement search functionality. For this part, you have to implement a function that finds the nearest neighbour in a data set for a given query using a k-d tree as a data structure.

The search procedure is recursive in the sense that it traverses the tree downwards and remembers the best point. Algorithm 2 shows the pseudo-code for the search procedure. It requires the starting Point *head* (e.g. the root) and the query point q. The result is the best point (closest) from the data set and its squared distance to the query point. The algorithm first starts by computing the distance between the query and the root of the tree (line 2). Then, it does a check whether or not the variables that track the 'best' point should be updated (lines 3–4). Obviously, the first time this will always happen as the variable *bd* is initialized to be Inf (infinite). Next, the difference between the head and q is computed to check whether the left node or right node is closer by (only considering the dimension which *head* uses to split), this is done on lines 6–13. The next step is to perform a recursive search where *head* becomes the node that is closer (node *close*). This recursive search should be skipped if *close* is empty (line 14). Finally, if *away* is not empty and the previously computed difference is smaller than the best distance, then another recursive search has to be performed with $head = away$. This last line is where the magic happens in the k-d tree search (why?).

Algorithm 2 Searching in a k-d tree

Input: a k-d tree, represented by its root node *head*
Input: a query point q
Input: the best distance *bd* (initially Inf)
Output: the best point *bp*
Output: the best distance *bd*
1: $headSD =$ compute Euclidean distance between *head* and q
2: **if** $headSD < bd$ **then**
3: $bp \leftarrow head$
4: $bd \leftarrow headSD$
5: **end if**
6: $d \leftarrow$ difference between *head* and q $(= q - head)$ in dimension $head.axisSplit$
7: **if** $d \leq 0$ **then**
8: $close = head.left$
9: $away = head.right$
10: **else**
11: $close = head.right$
12: $away = head.left$
13: **end if**
14: **if** *close* is not empty **then**
15: recursive search with $head = close$
16: **end if**
17: **if** *away* is not empty and $d^2 < bd$ **then**
18: recursive search with $head = away$
19: **end if**

Exercise part 2: classify new points based on their nearest neighbor

The k-NN algorithm requires the value for k to be specified. We will now use $k = 1$, meaning that an unseen data point will be classified to the same class as its nearest neighbor in the training data set.

Download the data set from `http://smaa.fi/tommi/courses/prog2/data/iris.data` into a directory accessible by Matlab. For more information on the data set, see `http://en.wikipedia.org/wiki/Iris_flower_data_set`. The data set has the following columns:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
   - Iris Setosa
   - Iris Versicolour
   - Iris Virginica

Make a script file, that:

1. Loads the data from iris.data and normalizes it (see function `zscore`)
2. Constructs a k-d tree with the iris.data using the four first attributes for the $k = 4$ dimensions on which the splitting is done. The actual values stored in the tree should contain the class as well.
3. Constructs 1000 points with 4 attributes having uniformly distributed values between the minimum- and maximum ones in the data (sepal and petal lengths and widths). Leave the class of these empty.
4. Uses the k-d tree searching procedure to find the closest point in the k-d tree for each of these. Assign each random point to the class of its nearest neighbor.
5. Makes two scatterplots where x-axis have the sepal length, y-axis the petal length, and the point itself color (blue, green, or red) indicating its class (setosa, versicolour, or virginica). For the first plot use the points from iris.data, and for the second one your recently classified random points. Plot according to the original vector space, not in the normalized space (i.e. the plots should have x- and y-axes in centimeters, not in z-values).