Algorithmic Complexity project: Classification of n points in d-sized Euclidean space

Luc Blassel, Romain Gautron

5 mars 2018

1 Introduction

1.1 What is classification ?

The problem we are trying to resolve is the following :
— We have n points in a d-dimensional Euclidean space
— each of these points is one of two colors : Blue or Red
— Some of the points have no assigned colors

The goal of this excercise os to accurately guess the color of the points for which we don't know it. This is a very common problem in machine learning.

1.2 A simple example

2 What is used today ?

2.1 A lot of different classification methods

There are hundreds of different methods to be able to classify these points into labels.

2.2 KNN

The principle behind the k nearest neighbours is very simple we try to find the k nearest points to the one we are trying to classify according to some distance function.

2.2.1 Finding the nearest neighbour

The naïve approach is very simple to implement, we choose a distance function, calculate the distance of the point we want to classify to all the points in the training set and we return the point for which that distance is minimal.

The distance used in this nearest neighbour search can be any of a number of functions (x being our known point, q the point we want to classify and d the number of dimensions) :

Euclidean distance : $d(x,q) = \sqrt{\sum_{i=1}^{d}(x_i - q_i)^2}$

Manhattan distance : $d(x,q) = \sum_{i=1}^{d} |x_i - q_i|$

Chebychev distance : $d(x,q) = \max_i(|x_i - q_i|)$

In a lot of cases the Euclidean distance is used, so in our implementation we will use it as well.

2.2.2 Transposing to k nearest neighbours

To select the k nearest neighbours we can simply run the nearest neighbour algorithm while keeping a priority queue of points sorted according to their distance and keep the k points with the lowest distances. this can also be acheived by keeping the points in a list along with their distance (rather thatn a priority queue) and sort the list according to distances after calculating all distances.

Once we have the k nearest neighbours to assign a label to our unclasssified point it is a simple majority vote situation where the label that is the most present in the k nearest neighbours is assigned to the point. We must of course take in to account the value of k if $k = 1$ then we are in the nearest neighbour problem, if k is big then what happens to size complexity, what happens if there is a tie in our k nearest neighbours ?

2.2.3 How can we lower the number of calculations ?

In the naïve approach we calculate the ditance of the unknown point to all the other points of the dataset. This is of course very costly computation-wise it would be best to eliminate some possibilities

2.3 Exact KNN versus approximate KNN

3 What did we implement ?

Of we course, as we wanted to be as efficient as possible, we did not implement a naive versoin of the k-NN method. We used k-dimensional trees *(ie. k-d trees)* to help us prune the search space and be able to be more efficient with our computing. k-d trees allow us to partition the k-dimensional Euclidean space into subspaces and organize our known points into a data structure similar to binary-search trees.

3.1 k-d trees

To be show how this structure is created we will use a simple 2-dimensional example, it is simpler to represent and visualize but higher dimensional k-d trees work in the exact same way.

The dataset we have is the following :

$$\{(1,3),(1,8),(2,2),(2,10),(3,6),(4,1),(5,4),(6,8),(7,4),(7,7),(8,2),(8,5),(9,9)\}$$

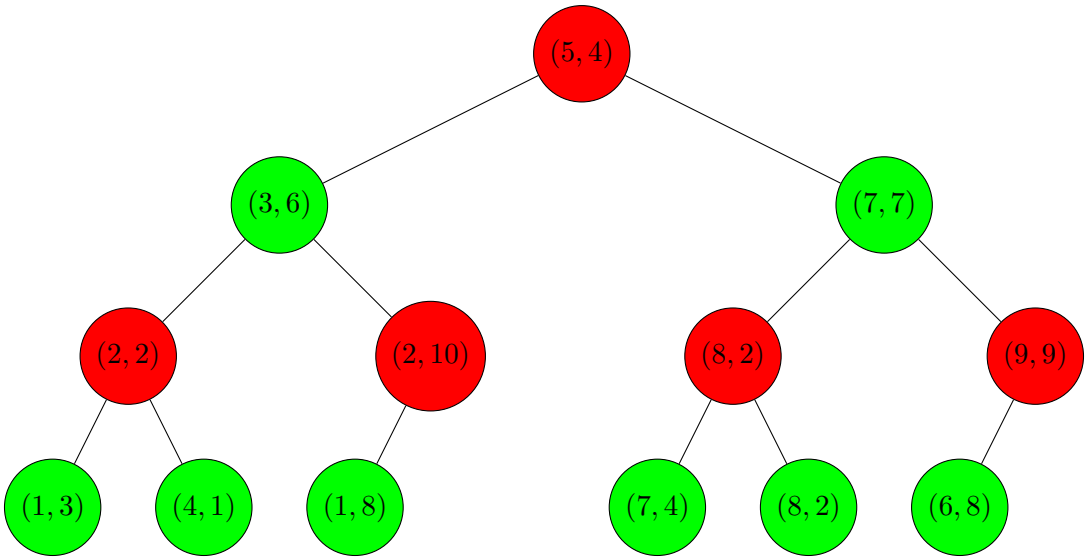
To construct the tree, we will alternatively split on the median of the first dimension and then the second at each level. So on level on we will split on the median of dimension 1, on the second level we will split on the median of the second dimension, on the third level we will split on the first dimension again and son on and so forth until all our points are placed in the tree. Since we always split on the median according to the selected dimesion, we can assure that the tree will be as balanced as it can be, which will speed up any subsequent searching operations in that tree.

So if we implement this method to our dataset we will have :

1. the median on the first dimension is $(5,4)$, so this point will be our first node of the tree. On the left subtree we will have all the points for which $x_1 < 5$ and on the right subtree all the points for which $x_2 >= 5$.
2. We now place ourselves in the left subtree of $(5,4)$. We will now split the sub-space according to the secon dimension. The median according to the second dimension is now $(3,6)$ so in the left subtree of $(3,6)$ will be all the points of the sub-space for which $x_2 < 6$ and on the right subtree all the points for which $x_2 >= 6$.

3. We build the tree in a recursive manner building the left subtree first and then the right subtree.

In the end we end up with the following k-d tree :



The red nodes are splitted on the first dimension, and the green nodes on the second.

It is also easy to see how this can be generalized to higher dimensions, the process is identical except that instead of looping on 2 dimensions, we loop on d dimensions. For instance if we have 3 dimensions, the first 3 levels are splitted on their corresponding dimensions and then the subsequent levels of the tree are splitted according to the remainder of the euclidean division $\frac{level}{d}$, so for example the 4^{th} level of the tree will be split along the 1^{st} dimension.

So in order to build the k-d tree given a set of points we can follow this algorithm :

```
createTree(list points, int dimensions, int depth, node parent) :
  Data: For the first iteration depth = 0 and parent = none
if point list is empty then
    return ;
end

axis ← depth%dimensions;
sort(points according to axis);
median ← length(points)/2;

root ← new node(value = points[median], parent=parent,axis=axis,vsited=False);
root.left ← createTree(points[ :median],dimensions,depth+1,root);
root.right ← createTree(point[median :],dimesions,depth+1,root);

return root;
```

The node object having a value element (the coordinates of the point), a left subtree, a right subtree and the axis bring the dimension along which it was split. The node object also has a visited boolean attribute that will be useful for search functions later on.

It is also interesting to note that on line 6 of the algorithm, the points list is sorted, depending on the sorting method the time complexity will not be the same. We use either shellSort or quicksort in our implementation.

3.2 k nearest neighbour search

Now that we have our data organized as a k-d tree, we must use this data-structure to our avantage as we search for the k nearest neighbours.

3.3 cross-validation

3.4 best case complexity

3.5 worst case complexity

3.6 results

4 Future works to be done

4.1 Change sort for tree creation

Median of medians

4.2 Use approximate KNN for selecting hyper-parameters

LSH in CV to select k