Chapter 3

Nearest neighbors

We discussed a simple document retrieval system using TF-IDF as the feature representation of the data, and cosine similarity as the measure of "goodness". This lecture, we will generalize this algorithm into something that we call as the *Nearest Neighbor* method.

The algorithm

Suppose that we have a database of data points $\{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^d$, and a query data point $x_0 \in \mathbb{R}^d$. Also suppose that we are given a distance measure $d(\cdot,\cdot)$ that measures closeness between data points. Typical choice of distances $d(\cdot,\cdot)$ include the ℓ_2-distance (or ℓ_1-distance).

The nearest neighbor (NN) method advocates the following (intuitive) method for finding the closest point in the database to the query.

1. Compute distances $d_i = d(x_i, x_0)$ for $i = 1, 2, \ldots, n$.
2. Output $i^* = \arg\min_{i \in [n]} d(x_0, x_0)$.

Particularly simple! The above two steps form a core building block of several, more complicated techniques.

Efficiency

The NN method makes no assumptions on the distribution of the data points and can be generically applied; that's part of the reason why it is so powerful.

To process each given query point, the algorithm needs to compute distances from the query to n-points in d dimensions; for ℓ_1 or ℓ_2 distances, each distance calculation has a running time of $O(d)$, giving rise to an overall running time of $O(nd)$.

This is generally OK for small n (number of samples) or small d (dimension), but quickly becomes very large. Think of n being in the $10^8 - 10^9$ range, and d being of a similar order of magnitude. This is very common in image retrieval and similar applications.

Moreover: this is the running time incurred for *each* new query. It is typical to possess one large (training) database of points and make repeated queries to this set. The question now is whether we can improve running time if we were allowed to do some type of preprocessing to the database to speed up running time of finding the nearest neighbor.

Improving running time: the 1D case

Consider the one-dimensional case ($d = 1$). Here, the data points (and query) are all scalars. The running time of naive nearest neighbors is O(n). Can we improve upon this?

Yes! The idea is to use a divide-and-conquer strategy.

Suppose the data points are given by $\{x_1, x_2, \ldots, x_n\}$. We *sort* the data points in increasing order to obtain the (permuted version of the) data set $\{x_{\pi_1}, x_{\pi_2}, \ldots, x_{\pi_n}\}$. This takes $O(n \log n)$ time using MergeSort, etc.

Now, for each query point x_0, we simply perform *binary search*. More concretely: assuming that n is even, we compare x_0 to the median point. $x_{\pi_{n/2}}$. If $x_0 > x_{\pi_{n/2}}$, then the nearest neighbor to x_0 cannot belong to the bottom half $\{x_1, \ldots, x_{\pi_{n/2-1}}\}$. Else, if $x_0 < x_{\pi_{n/2}}$, then the nearest neighbor cannot belong to the top half $\{x_{\pi_{n/2+1}}, \ldots, x_{\pi_n}\}$. Either way, this discards half the number of points from the database. We recursively apply this procedure on the remaining data points.

Eventually, we will be left with a single data point x_j. We output $i^* = \pi^{-1} j$ as the index of the nearest neighbor in the (original) database.

Since the dataset size decreases by a factor 2 at each recursive step, the number of iterations is at most $\log_2 n$. Therefore, the running time of nearest neighbors for each query is $O(\log n)$.

So by paying a small additional factor ($O(\log n)$) in terms of pre-processing time, we can dramatically speed up running time per query. This kind of trick will often be used in several techniques that we will encounter later.

Extension to higher dimensions: kd-trees

The "binary search" idea works well in one dimension ($d = 1$). For $d > 1$, a similar idea called *kd-trees* can be developed. (The nomenclature is a bit strange, since "kd" here is short for "k-dimensional". But to be consistent, we will use the symbol d to represent dimension.) It's a bit more complicated since there is no canonical way to define "binary search" in more than one dimension.

Preprocessing

For concreteness, consider two dimensions ($d = 2$). Then, all data points (and the query point) can be represented within some bounded rectangle in the XY-plane.

We first sort all data points according to the X-dimension, as well as according to the Y-dimension.

Next, we arbitrarily choose a *splitting direction* along one of the two axes. (Without loss of generality, suppose we choose the X-direction.) Then, we divide the data set into two subsets according to the X-values of the data points by drawing a line perpendicular to the X-axis through the *median* value of the (sorted) X-coordinates of the points. Each subset will contain the same number of points if n is even, or differ by 1 if n is odd.

We recursively apply the above splitting procedure for the two subsets of points. At every recursive step, each of the two subsets will induce two halves of approximately the same size. One can imagine this process as an approximately balanced binary tree, where the root is the given input dataset, each node represents a subsets of points contained in a sub-rectangle of the original rectangle we started with, and the leaves correspond to singleton subsets containing the individual data points.

After $O(\log n)$ levels of this tree, this process terminates. Done!

There is some flexibility in choosing the splitting direction in each recursive call. One option is to choose the axis where the coordinates of the data points have the maximum variance. The other option is to simply alternate between the axes for $d = 2$ (or cycle through the axes in a round-robin manner for $d > 2$). The third option is to choose the splitting direction at random. Either way, the objective is to make sure the overall tree is balanced.

The overall running time is the same as sorting the n data points along each dimension, which is given by $O(dn \log n)$.

Making queries

Like in the 1D case, nearest neighbor queries in kd-trees can be made using a divide-and-conquer strategy. We leverage the pre-processed data to quickly discard large portions of the dataset as candidate nearest neighbors to a given query point.

Identical to the 1D case, the nearest neighbor algorithm starts at the root, looks at the splitting direction (in our above example, the initial split is along the X-direction), and moves left or right depending on whether the query point has its corresponding coordinate (in our above example, its X-coordinate) smaller than or greater than the median value. Recursively do this to traverse all the way down to one of the leaves of the binary tree.

This traversal takes $O(\log n)$ comparisons. Now, unlike the 1D case, there is no guarantee that the leaf data point is the true nearest neighbors (since we have been fairly myopic and only looked at each co-ordinate in order to make our decisions while traversing the tree.) So, we need to do some additional calculations to refine our estimate.

Declare the data point in the leaf as the *current estimate* of the nearest neighbor, and calculate the distance, Δ, between the current estimate and the query point. It is certainly true that the *actual* nearest neighbor cannot be further away than the current estimate. In other words, the true nearest neighbor lies within a circle of radius Δ centered at the query point.

Therefore, we only need to examine the sub-rectangles (i.e., nodes of the binary tree) that *intersect this circle*! All other rectangles (and points within those rectangles) are irrelevant.

Checking whether the circle intersects a given sub-rectangle is simple: since each rectangle is specified by a split that occurred along some X- (or some Y-) coordinate (say at value α), we only have to check whether the difference between the splitting coordinate of the query point and α exceeds Δ. If yes, then there is no intersection; we can safely discard the sub-rectangle (and all its children). This intuition supports the divide-and-conquer strategy.

One can prove that the *expected* number of additional nodes that need to visit is given by $O(\log n)$ if the original data is generated from certain random distributions. However, in the worst case we may still need to perform $O(n)$ distance calculations, which implies a worst-case running time of $O(nd)$ (i.e., no better than vanilla nearest neighbors.)

Nevertheless, kd-trees are often used in practice, particularly for moderate dimensional problems. Most modern machine learning software packages (such as scikit-learn for Python) have robust implementations of kd-trees that offer considerable speedups over nearest neighbors.