CMSC 35900 (Spring 2009) Large Scale Learning **Lecture: 7**

Metric trees for fast NN search

Instructors: Sham Kakade and Greg Shakhnarovich

Today we will focus on the problem of efficiently finding k-NN of a point in $\mathbb{R}^d$, with respect to an L_p metric

$$L_p(\mathbf{x}_i, \mathbf{x}_j) \;=\; \left(\sum_{q=1}^{d} |x_{i,q} - x_{j,q}|^p \right)^{1/p}.$$

The cost of the naïve search by linear scan in a database of n examples is $O(nd)$. In many applications this is unacceptable, and we would like to have a search algorithm with query time sublinear in n. Note however, that we also should worry about the dependence on dimension d.

1 kd-trees: construction and search

The basic intuition behind kd-trees (k-dimensional trees) is to partition the space hierarchically into a tree, splitting along one dimension at a time. If we manage to do this in such a way that points that are far from each other end up in different leaves of the tree, and the depth of the tree is not too large, we can hope to be able to find neighbors fast by traversing the tree.

1.1 Building kd-tree

Each node of the kd-tree represents an example from the database, a d-dimensional hyperrectangle containing that example, and information on how the node partitions (splits) that hyperrectangle.

The input to the tree-building algorithm is a hyperrectangle and a set of examples. To start, we call the algorithm BUILDKD(R, X) where R is the smallest hyperrectangle R containing all the examples, and X is the full example set $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$.

Each branching of the tree is created by selecting a *pivot* $\mathbf{r} \in X$ (the example that will serve as the root of the subtree), along with the splitting dimension $j \in \{1, \ldots, d\}$. We will discuss the details of this selection a bit later. Given $\mathbf{r}$ and j, the current node contents is set to $\langle R, \mathbf{r}, j \rangle$. The example set $X \backslash \{\mathbf{r}\}$ is partitioned into two subtrees, "top" and "bottom", as follows.

$$\begin{aligned} X_{\text{top}} &= \{\mathbf{x}_t \in X \backslash \{\mathbf{r}\} \;:\; x_{ij} \geq r_j\}, \\ X_{\text{bottom}} &= \{\mathbf{x}_t \in X \backslash \{\mathbf{r}\} \;:\; x_{ij} < r_j\}. \end{aligned}$$

Similarly, the hyperrectangle R is partitioned into R_{top} and R_{bottom} by splitting the j-th dimension at r_j. The subtrees rooted at the current node are created recursively by calling BUILDKD(R_{top}, X_{top}) and BUILDKD(R_{bottom}, X_{bottom}).

The recursion call BUILDKD(R, X) terminates when X contains a single example. This example is assigned to hyperrectangle R, which becomes a leaf in the tree associated with that example. We could also have an empty leaf, in case X_{top} or X_{bottom} (but not both) is empty. Then, we will associate the leaf with $\mathbf{r}$. In the end, the algorithm stops when each node is either in a non-terminal node, or in a leaf.

1.2 Search in kd-tree

Gibe the kd-tree with depth m, we can use it to search for a NN of a query $\mathbf{x}_0$. The first step is to find the leaf that contains $\mathbf{x}_0$. It is easy to do, and takes $O(m)$ comparisons. This yields *a* neighbor: the example $\mathbf{x}'$ associated with that leaf. The algorithm maintains the identity of the nearest neighbor found so far $\mathbf{x}^*$ and the distance to this neighbor, ρ^*. At this point, $\mathbf{x}^* := \mathbf{x}'$, and $\rho^* := D(\mathbf{x}_0, \mathbf{x}')$. Note that we always have $D(\mathbf{x}_{(1)}(\mathbf{x}_0), \mathbf{x}_0) \leq \rho^*$, so that ρ^* provides an upper bound on the distance to the true NN.

The search then proceeds by backtracking and branch-and-bound approach, depth-first, as follows. We backtrack to the parent of the current node. Suppose this parent is $\langle R, \mathbf{r}, j\rangle$, and let $\rho = D(\mathbf{x}_0, \mathbf{r})$. If $\rho < \rho^*$, we set $\mathbf{x}^* := \mathbf{r}$ and $\rho^* := \rho$.

Next, we consider descending to the other subtree of the current node (not the one we just came from). Let the root of this subtree (child of $\langle R, \mathbf{r}, j\rangle$) be $\langle \tilde{R}, \tilde{\mathbf{r}}, \tilde{j}\rangle$. This descent only makes sense if there is a chance that the true NN could be there. To find out, we compute the intersection between a hyperrectangle $\tilde{R}$ and the ball $B_{\rho^*}(\mathbf{x}_0)$. If the intersection is empty, we abandon the entire subtree rooted at $\langle \tilde{R}, \tilde{\mathbf{r}}, \tilde{j}\rangle$, and continue backtracking to the parent of $\langle R, \mathbf{r}, j\rangle$.

The intersection can be tested very efficiently: The hyperrectangles are represented by the two vectors $\mathbf{l} = [l_1, \ldots, l_d], \mathbf{u} = [u_1, \ldots, u_d] \in \mathbf{R}^d$ that specify the "corners" with the lowest and the highest coordinates. We can find the point $\mathbf{p}$ in the (closed) hyperrectangle that is closest to $\mathbf{x}$. For each coordinate q, we set

$$p_q(\mathbf{x}) = \begin{cases} l_q & \text{if} \quad x_q \leq l_q, \\ x_q & \text{if} \quad l_q \leq x_q \leq h_q \\ h_q & \text{if} \quad x_q \geq h_q. \end{cases}$$

Given a ball $B_\rho(\mathbf{x}_0)$ we find the point $\mathbf{p}(\mathbf{x}_0)$. If $D(\mathbf{p}(\mathbf{x}_0), \mathbf{x}_0) \leq \rho$ then the hyperectangle and $B_\rho(\mathbf{x}_0)$ intersect; otherwise, they do not.

It is easy to modify the algorithm slightly to allow for a slightly different search task. If we need to find the k nearest neighbors, we maintain, during search, the list of k nearest neighbors found so far $\mathbf{x}_{(1)}*, \ldots, \mathbf{x}_{(k)}*$, and use $\rho^* = D(\mathbf{x}_0, \mathbf{x}_{(k)}*)$ when considering whether to examine a tree branch.

Similarly, for a range search, when we are given a specific distance r and asked to find neighbors within r from $\mathbf{x}_0$, we maintain a fixed sphere of radius r when testing for intersections with leaves, and also record all the points within r from $\mathbf{x}_0$ that we encounter. Note that in both of these modifications, we do not need to modify the data structure, only the search algorithm!

In the standard search algorithm, we essentially examine leafs in the order determined by their distance from the initial leaf in the tree. Instead, we can consider ordering them according to their distance from the initial leaf in $\mathbb{R}^d$. This can be done by using a priority queue. This algorithm is called Best Bin First (BBF) search, and has been shown to work somewhat better in practice.

1.3 Splitting stategy

Two procedures are a popular choice of the pivot and the splitting dimension.

Median of highest variance Select the dimension in which the variance of the data in the subtree is highest, and split at the median value. Variance is normally estimated robustly, after removing outliers on the extreme tails of the empirical distribution. This produces a balanced tree, but for some distributions the resulting bins are over-extended in few dimensions and very compact in others ("long skinny" bins).

Middle of max-spread Find the dimension with the largest spread range, and choose the pivot closest to the middle of this range. The balance may be less perfect, but the shape of the bins may be more even. We will assume that this strategy is used in practice.

2 Analysis

The following analysis is from [1]. It is useful to study primarily because it offers some insights into the behavior of the tree and the search algorithm. However, it relies on some assumptions which are questionable in practice, as we will see.

We consider $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n$ drawn i.i.d. from $p(\mathbf{x})$. Let $S(\mathbf{x}, k)$ be the smallest ball centered at $\mathbf{x}$ and containing exactly k NN of $\mathbf{x}_0$,

$$S(\mathbf{x}, k) \;=\; \{\mathbf{y} \in \mathbb{R}^d \;:\; D(\mathbf{x}, \mathbf{y}) \;\leq\; D(\mathbf{x}, \mathbf{x}_{(k)}(\mathbf{x})\}.$$

We denote the volume of this ball by

$$v_k(\mathbf{x}) \;=\; \int_{S(\mathbf{x},k)} d\mathbf{y}$$

and its probability mass by

$$\pi_k(\mathbf{x}) \;=\; \int_{S(\mathbf{x},k)} p(\mathbf{y}) d\mathbf{y}.$$

The latter is a random variable (due to dependence on the k-NN which in turn depends on the sample). Its probability can be found by considering the event that exactly k points fall inside:

$$P(\pi_k(\mathbf{x})) \;=\; \frac{n!}{(k-1)!(n-k)!} \left(\pi_k(\mathbf{x})\right)^{k-1} \left(1 - \pi_k(\mathbf{x})\right)^{n-k} \;=\; B(k, n+1-k), \tag{1}$$

the beta distribution. This is independent of the underlying $p(\mathbf{x})$.[1] Taking the expectation we get

$$E\left[\pi_k(\mathbf{x})\right] \;=\; \frac{k}{n+1}.$$

We assume that n is large enough so that the ball $S(\mathbf{x}_0, k)$ is small enough, so that $p(\mathbf{x})$ is approximately flat inside the ball, with average probability density $\bar{p}(\mathbf{x}_0)$. Then, we can approximate

$$\pi_k(\mathbf{x}_0) \;\approx\; \bar{p}(\mathbf{x}_0) v_k(\mathbf{x}_0),$$

and thus

$$E\left[v_k(\mathbf{x}_0)\right] \;\approx\; \frac{k}{(n+1)\bar{p}(\mathbf{x}_0)}. \tag{2}$$

Consider now the leaf of kd-tree that contains one example, $\mathbf{x}_i$. By derivation identical of that leading to (2), the expected volume of this (assumed small) region is

$$E\left[v_1(\mathbf{x}_i)\right] \;\approx\; \frac{1}{(n+1)\bar{p}(\mathbf{x}_i)}. \tag{3}$$

[1] There is nothing special about the ball shape, or even with the identity of the points as nearest neighbors of a query. This result holds for any compact volume enclosing exactly k points from $p(\mathbf{x})$.

Now consider the hypercube $C(\mathbf{x}_0)$ circumscribed around $S(\mathbf{x}_0, k)$, and let $V_k(\mathbf{x}_0)$ be its volume. We have

$$V_k(\mathbf{x}_0) \;=\; G(d)v_k(\mathbf{x}_0),$$

with constant G that depends on the dimension and the metric D.[2] The expected volume of $C(\mathbf{x}_0)$ is

$$E\left[V_k(\mathbf{x}_0)\right] \;=\; \frac{kG(d)}{(n+1)\bar{p}(\mathbf{x}_0)}. \tag{4}$$

We are ready now to bound the expected number of comparisons in the kd-tree search for $\mathbf{x}_0$. This number is bounded from above by the expected number of leaves that intersect $S(\mathbf{x}_0, k)$, and that in turn is bounded by the expected number of leaves that intersect $C(\mathbf{x}_0)$. The expected edge length of $C(\mathbf{x}_0)$ is

$$\left(\frac{kG(d)}{(n+1)\bar{p}(\mathbf{x}_0)}\right)^{1/d},$$

We will make the simplifying[3] assumption that the expected shape of the leaf hyperrectangle is a hypercube (that is, all sides are equal). From (3), the expected length of its edge is

$$\left(\frac{1}{(n+1)\bar{p}(\mathbf{x}_i)}\right)^{1/d}.$$

Furthermore, we assume that if the $p(\mathbf{x}_0) \approx p(\mathbf{x}_i)$. Therefore the expected number of leaves overlapping $C(\mathbf{x}_0)$ is

$$\Big([kG(d)]^{1/d} + 1\Big)^d$$

and that provides an approximate bound on the expected number of comparisons in the search when the query is $\mathbf{x}_0$.

For L_2 norm in $\mathbb{R}^d$, the constant $G(d)$ is, for any r, the ratio of the volume of a hypercube with side $2r$ to that of a ball with radius r:

$$G(d) \;=\; \frac{(2r)^d\Gamma(d/2+1)}{\pi^{d/2}r^d} \;=\; \frac{2^d\Gamma(d/2+1)}{\pi^{d/2}}.$$

The bound on the number of comparisons is then

$$\left(k^{1/d}\frac{2\Gamma(d/2+1)^{1/d}}{\sqrt{\pi}} + 1\right)^d.$$

One could suspect that leaving $b > 1$ example per leaf could increase performance. However, if we plug in b instead of 1 in the analysis above, it can be shown [1] that $b = 1$ is in fact optimal for minimizing the expected number of comparisons.

A big problem with this analysis is that it relies on a number of approximations and assumptions which may not hold in practice. Therefore, it is important to investigate the behavior of the algorithm on a particular data set empirically. Andrew Moore's tutorial [2] is an example of such empirical analysis. In general, a widely accepted "rule of thumb" is that kd-trees work well when dimension d is no higher than 12-15, and deteriorate rapidly to linear time when d grows to 20 and above. Furthermore, kd-trees do not take full advantage of low-dimensional structure in the data, such as when the data, while embedded in a nominally high-dimensional space, conform to a low-dimensional subspace.

[2]Note that $G(d)$ is monotonically increasing with d.

[3]and not quite realistic!

References

[1] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[2] Andrew Moore. A tutorial on kd-trees. Extract from PhD Thesis, 1991. Available from http://www.cs.cmu.edu/∼awm/papers.html.