CPS290: Algorithmic Foundations of Data Science **February 3, 2017**

Lecture 4 : Similarity Search

Lecturer: Kamesh Munagala *Scribe: Kamesh Munagala*

In the similarity search problem, we are given a database D consisting of n items. Given a query q, the goal is to report all $x \in D$ that are "near" the query point q. Such a problem is of fundamental importance to recommendation systems: Given an image, what images are similar; given a movie and a measure of similarity between movies, what are other movies to recommend to the user; given a product purchased, what are similar products to recommend; and so on. In typical settings, n is massive, say all movies on Netflix, or all products on Amazon.

Solving the near-neighbor problem requires two intertwined ingredients:

1. A *similarity measure* between the items. The similarity measure usually depends on the context – what works for movies may not work for images. We survey some commonly used measures below.

2. A fast algorithm that given a query, finds the most similar item(s). Of course, this is easy to solve by brute-force search – for each $x \in D$, compute the similarity between q and x, and report the items that are most similar. This takes n similarity computations. The question we ask is – can we beat this? If so, how are the algorithms dependent on the similarity measure?

Similarity Measures

Before proceeding further, it makes sense to survey the commonly used similarity measures. The notion of similarity is strongly tied to applications, so we only present the definitions without too much practical motivation.

Jaccard Measure. This measure is most commonly used when objects are represented as sets. For instance, a webpage can be represented as a collection of words (with frequencies), or a collection of phrases (shingles). Given two sets S and T, the Jaccard measure is

$$J(S,T) = \frac{|S \cap T|}{|S \cup T|}$$

This does not take multiplicities of items into account. If sets are represented as non-negative vectors $\vec{x}, \vec{y} \in R^d_+$, then

$$J(\vec{x},\vec{y}) = \frac{\sum_{k=1}^{d} \min(x_k, y_k)}{\sum_{k=1}^{d} \max(x_k, y_k)}$$

Angular Distance. This is the angle between $\vec{x}$ and $\vec{y}$ defined as:

$$D(\vec{x},\vec{y}) = \cos^{-1}\left(\frac{\vec{x}\cdot\vec{y}}{\|\vec{x}\|\|\vec{y}\|}\right)$$

A very related measure is *cosine similarity*, which is the cosine of the angle between vectors.

Euclidean Distance. This is a measure of dissimilarity between items represented as vectors in d-dimensional space.

$$d(\vec{x}, \vec{y}) = \|\vec{x} - \vec{y}\|_2 = \sqrt{\sum_{k=1}^{d} (x_k - y_k)^2}$$

General Norms. Instead of Euclidean, which is a squared measure, we can use any p-norm, for $p > 0$.

$$d_p(\vec{x}, \vec{y}) = \|\vec{x} - \vec{y}\|_p = \left(\sum_{k=1}^{d} (x_k - y_k)^p \right)^{1/p}$$

For $p \neq 2$, this norm is not rotationally invariant, meaning that if the vectors are rotated by the same amount, then the distance between them changes. The case of $p = 1$ is termed the *Manhattan distance*, and is commonly used as a distance measure for binary vectors, where it is also called the *Hamming distance*.

Low-dimensional Search and k-d Trees

Suppose the items are points on a line. Given a query q, the goal is to output all points in D within distance R from q. This can be achieved by storing D in a binary search tree. The range of radius R around q defines an interval; we search for its left and right end-points in the binary search tree, and output the collection of subtrees found. The details are an easy exercise, and the running time is $O(\log n + s)$, where $n = |D|$, and s is the number of items that are output. Note that this beats brute force search that takes $O(n)$ time.

We next consider the same problem when items are points in 2 dimensions. We assume a query is a rectangle, and the goal is to output all items in D that fall within this rectangle. Can we maintain D in a data structure so that the running time of a query beats brute force evaluation of each point. It is *a priori* not obvious that this can be done in the worst case. We will present the simplest such data structure, termed a *k-D tree*.

The k-D tree partitions the points by alternate horizontal and vertical lines as follows: At a recursive step, suppose S is the set of points, and the goal is to find a horizontal split. Find the median of the y-coordinates of the points in S, and consider the horizontal line defined by this median. Generate two sets S_1 and S_2, which are the points above and below this line. Recurse on S_1 and S_2 with the split lines being vertical. If S is a single point, then stop.

The above procedure defines a binary tree. Each level of the tree is either "horizontal" or "vertical". Each node in the tree corresponds to the set of points in its subtree, and is a region defined by the intersection of the half-planes defined by the lines at this node and its ancestors in the tree. The points corresponding to this node lie in this region. As we go deeper into the tree, the region defined by the nodes become smaller and smaller, till we reach a leaf which has only one point in its region.

Since the line at each node splits the points in two, the depth of the tree is at most $\log_2 n$. It is an easy exercise to show that such a tree can be constructed in $O(n \log n)$ time.

Query Procedure and Analysis

We will show that if there are s points in the query rectangle, and $n = |D|$, then the query time is $O(\sqrt{n} + s)$. The analysis also shows a query time of $O(dn^{1-1/d} + s)$ if the data structure and query are generalized to d dimensional points.

The query algorithm is simple: Suppose we reach a node v in the tree. If the region defined by v does not intersect q, then we return. Else, if q contains the region defined by v, we return all the points in the subtree rooted at v. Else we recurse on the left and right subtrees of v.

The analysis proceeds by showing that the number of recursive calls made is $O(\sqrt{n})$. If at any step, the procedure did not recurse, then it output all the points in the subtree, which can be done in time linear in the number of points. (Why?) The fact that the number of recursive calls is much less than n is somewhat surprising, and needs a careful argument.

Let's ask: Suppose the query algorithm reaches node v in the tree. Then why did it recurse? It recursed because the query region intersected the region defined by v, but did not contain this region. This means some edge of q intersected this region. So far so good. So let's take an edge, and ask: How many regions of the k-d tree (defined by vertices of the tree) can this edge intersect? Suppose this is $O(\sqrt{n})$. Then since there are only 4 edges in the query, the number of recursive calls will also be $O(\sqrt{n})$ and we are done.

Suppose edge ℓ intersects the region defined by v. Suppose ℓ is a horizontal line, and suppose the next split (to get the children of v) is a vertical split. Then it could happen that ℓ lies on *both* sides of this split. But if we go down one more level, to the grandchildren of v, the splits that take the children of v to their grandchildren are horizontal splits. A horizontal line ℓ cannot lie on both sides of a horizontal split. This means ℓ can only intersect two the four regions formed when we split v in two levels.

Let $Q(n)$ denote the number of k-d tree nodes that intersect with a horizontal line ℓ, when the root has n points in its tree, and the first split is vertical. The above argument shows:

$$Q(n) = 3 + 2Q(n/4)$$

where the 3 is because ℓ intersects the root and both its children. Solving this recurrence, we have $Q(n) = O(\sqrt{n})$, and this completes the argument!

The generalization to high dimensions is relatively straightforward. In practice, the query time tends to be a very pessimistic estimate. There are several tricks to speed up the queries. First, we can choose to split at a random point instead of at the median. Secondly, we can split along the dimension whose extant is largest, instead of going in round-robin order. Finally, it is not even clear why the splits have to be axis-aligned – choosing random directions is one alternative. Such tricks make $k - d$ trees practical for moderate dimensions, like $d = 50$ or so.

Curse of Dimensionality

In high dimensions, k-D trees become closer and closer to brute force search. This is not a flaw in our data structure, but rather a fundamental property of high dimensional spaces. In particular, high dimensional spaces are "lonely" – given any set D of n points, and a random query point q, with high probability, all the n points in D will appear to be roughly as far away from q as the extant of the space. We will formalize this below.

Without loss of generality assume the points in D and the point q lie in $[0, 1]^d$, and assume each coordinate of $x \in D$ and q is chosen uniformly at random from $[0, 1]$. Suppose we consider a d-dimensional square of width $1/2$ centered at q. Then, the probability that $x_i \in D$ lies in q is $(1/2)^d$. This is because each coordinate is drawn uniformly at random in $[0, 1]$ so that the probability that it lies in the corresponding side of the square is $1/2$. The different dimensions are independent, so the overall probability is $(1/2)^d$.

Now, the probability that no point from D lies in the square is $(1-(1/2)^d)^n \approx 1 - \frac{n}{2^d}$ assuming $n \ll 2^d$, or $d = \omega(\log n)$. This means that with high probability, *all* points in D are at least distance $1/2$ from q in every dimension! Since the largest possible distance in any dimension is 1, this means the closest and furthest points from q are both roughly the same distance away. This complicates designing a data structure for finding the nearest neighbor – we have to search among points which are all comparable in distance. For instance, with a k-D tree, we need to search the tree with a large region around q. But such a large region will intersect exponentially many cells of the k-D tree, essentially the entire tree.

Where do we go from here?

High dimensional spaces behave in a very counterintuitive fashion. For instance, how many points can be placed in d dimensions so that they are all equidistant from each other? In $d = 1$ dimension, the answer is two points. On a plane, the answer is 3 points. How does this scale as a function of d? The answer is that for large d, there can be roughly $O(e^{\sqrt{d}})$ such points, so the scaling is exponential. This is the fundamental reason why nearest neighbor search is complicated.

Is there anything that can save us from brute force search? There are three possible answers that we will explore in detail.

Dimensionality Reduction. The random query point instance only works if $d \gg \log n$. Maybe we can reduce the dimension to $\log n$ while approximately preserving pairwise distances between n points. This is the idea of *embedding* a Euclidean space into a smaller dimensional space while preserving distances approximately. The dimension of $\log n$, though large, may be sufficiently small for k-D trees to beat brute force search. This is the idea of the *Johnson-Lindenstrauss (J-L) transform.*

Approximation and Hashing. The random query example also shows something more subtle – if all squared distances are at least $d/36$, we can output any point $x \in D$ as the nearest neighbor, and it will be correct to a factor of $\sqrt{36} = 6$. Suppose we are happy with approximation, then maybe the problem admits to fast algorithms by hashing. This is a non-trivial, yet very practical, idea that we will explore. In fact, such an approach generalizes to finding approximate nearest neighbors in non-Euclidean similarity measures. This is the idea of *Locality Sensitive Hashing (LSH).*

Data-dependent Schemes. Often, though data is high-dimensional, it is usually not a random collection of points. It is possible the data lies in some low-dimensional space, and so does the query point. We will spend some time exploring techniques to unearth low-dimensional structure in data. This is the idea of *Principal Component Analysis (PCA).*