

# CS168: The Modern Algorithmic Toolbox

## Lecture #3: Similarity Metrics and $kd$ -Trees

Tim Roughgarden & Gregory Valiant\*

April 9, 2024

### 1 Similarity Search

We begin with the basic problem of how to organize/represent a dataset so that similar items can be found quickly. There are two slightly different settings in which one might want to consider this question:

1. All the data is present, and one wants to find pairs or sets of similar items from within the dataset.
2. There is a reference dataset that one has plenty of time to process cleverly, but when we are given a *new* datapoint, we want to very quickly return a similar datapoint from the reference dataset. This setting is sometimes referred to as the “nearest neighbor search” setting.

In general, similar techniques/approaches are used for the above two settings, though it is worth being aware of the the different objectives.

There are many real-world applications of similarity search:

- Similar documents, web pages, genomes, etc. (de-duplication of datasets, plagiarism detection in code/essays, detecting mirror web sites, finding similar genes or sequencing populations of organisms such as in the human gut “microbiome”).
- Collaborative filtering: find similar products (based on whether the same set of people purchased them) or individuals (based on purchase history, demographics, web browsing behavior, etc.).
- Machine learning/classification: If we find two similar datapoints, they might have the same label....

---

\*©2015–2024, Tim Roughgarden and Gregory Valiant. Not to be sold, published, or distributed without the authors’ consent.

- Combining datasets (e.g. in astronomy—different telescopes take pictures of the same portions of the sky, maybe in different wavelengths, etc.—very useful to automatically aggregate these datasets).
- Super fast labeling: e.g. in CERN, which bashes particles together and needs to very quickly figure out which new particle traces/trajectories are “interesting” and worth saving, and which trajectories are just boring/common particles.

## 2 Measures of Similarity

Before talking about algorithms for finding similar objects, we should begin by considering several quantifications of what “similarity” means.

### 2.1 Jaccard Similarity

Jaccard similarity, which we denote by  $J(\cdot, \cdot)$ , is a distance metric between two sets,  $S, T$  (or two multisets—sets where elements are allowed to appear more than once):

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

Equivalently, if we represent our sets (or multisets)  $S, T$  as vectors  $v_S, v_T$ , with the  $i$ th index of the vector  $v_S(i)$  equalling the number of times that the  $i$ th element is represented in  $S$ , the above definition becomes:

$$J(S, T) = J(v_S, v_T) = \frac{\sum_i \min(v_S(i), v_T(i))}{\sum_i \max(v_S(i), v_T(i))}.$$

This expression is undefined if  $S, T$  are both the empty set, in which case we can define the distance to be 0.

Jaccard similarity works quite well in practice, especially for sparse data. For example, if we represent documents in terms of the multiset of words they contain, then the Jaccard similarity between two documents is often a reasonable measure of their similarity. Similarly, to estimate similarity between individuals, an online marketplace like Amazon, might represent people as multisets of items purchased, movies reviewed, etc., and use Jaccard similarity.

### 2.2 Euclidean Distance/ $\ell_2$ distance, and $\ell_p$ distance

Given datapoints in  $\mathbb{R}^d$ , the Euclidean distance metric, which we are all familiar with, is simply

$$D_{euclidean}(x, y) = \|x - y\|_2 = \sqrt{\sum_{i=1}^d (x(i) - y(i))^2}.$$

More generally, we can define other measures of similarity for points in  $\mathbb{R}^d$  that generalize the above form; the  $\ell_p$  distance is defined as

$$\|x - y\|_p = \left( \sum_{i=1}^d |x(i) - y(i)|^p \right)^{1/p}.$$

If  $p = 1$ , we get “manhattan” distance, and for large  $p$ ,  $\|x - y\|_p$  is more and more dependent on the coordinate with maximal difference, with the  $\ell_\infty$  distance simply being defined as  $\max_i |x(i) - y(i)|$ .

Note that  $\ell_2$  distance is rotationally invariant, whereas  $\ell_p$  for  $p \neq 2$  is not invariant to rotations of the space. One consequence of this fact is that if you are using  $\ell_p$  distance with  $p \neq 2$ , you should make sure that the coordinates of your space actually mean something—it does not make too much sense to use a distance metric that depends on your choice of coordinate axes, if your choice of coordinate axes are arbitrary.

### 2.3 Other similarity metrics

There are many other similarity metrics, including “cosine similarity” which you will see on the homework, “edit distance” that measures the similarity between strings (documents, genetic sequences, etc.) by asking “how many edits—i.e. insertions/deletions” does it take to get from one string to the other, and “earthmover distance” which measures how much effort it takes to turn one datapoint into another (this is often used to measure distance between probability distributions, or distance between images). Given all these similarity metrics, it is always worth spending some time thinking about which metric is appropriate for a given problem.

### 2.4 The Relationships between Metrics, and Metric Embeddings

A very natural mathematical question is “how different are the different metrics”? A specific practical motivation is that many geometric algorithms are designed for Euclidean distance (in part because of its rotational invariance). Given a set of points, and a distance metric  $D$ , is it possible to map the points into a set of point in  $\mathbb{R}^d$ , such that the original distance between points is equal to (or closely approximated by) the Euclidean distance between the images of those points? Formally, given a distance metric  $D$ , and a set of points  $X = x_1, \dots, x_n$ , is it possible to map the points to a set  $Y = y_1, \dots, y_n$ , where  $y_i \in \mathbb{R}^d$ , ideally for a lowish dimension  $d$ , s.t. for all  $i, j$ ,

$$D(x_i, x_j) \approx \|y_i - y_j\|_2.$$

This is known as a “metric embedding”—in this case, an embedding into  $\mathbb{R}^d$  under Euclidean distance—and there is a whole area of math/geometry/computer science devoted to studying when such embeddings exist.

### 3 A Datastructure for Similarity Search: kd-trees

For the remainder of the lecture, we will focus on Euclidean distance, though it is worth thinking about how one would apply these ideas to other similarity measures.

A *kd-tree*, originally proposed by Bentley (a Stanford undergrad) in 1975, is a space partitioning datastructure that allows one to very quickly find the closest point in a dataset to a given “query” point. *kd-trees* perform extremely well when the dimensionality (or effective dimensionality) of the data is not too large—usually people say that it works well if the dimensionality of the space is less than 20, or if the number of points is at least  $2^d$ , where  $d$  is the dimensionality of the points. There are many variants of *kd-trees*, and you should think of this as a general framework for designing such a datastructure, though the specifics can be fruitfully tailored to individual datasets and applications.

The high-level idea is to build a binary search tree that partitions space. Edges of the tree will correspond to subsets of space, and each node,  $v$ , in the tree will have two data-fields: the index of some dimension  $i_v$ , and a value  $m_v$ . Let  $S_v$  denote the subset of space corresponding to the edge going into a node  $v$ , and let  $S_<, S_>$  denote the subsets of space corresponding to the two outgoing edges of  $v$ . These subsets will be defined as  $S_< = \{x : x \in S_v, \text{ s.t. } x(i_v) < m_v\}$  and  $S_> = \{x : x \in S_v, \text{ s.t. } x(i_v) \geq m_v\}$ .

We build this tree as follows:

#### Algorithm 1

##### KD-TREE

Given a pair  $[S, v]$ , where  $S = x_1, \dots, x_n$  is a set of points with  $x_i \in \mathbb{R}^d$ , corresponding to node  $v$  in a partially built *kd-tree*:

- if  $n = 1$ , then store that point in the node  $v$ .  $v$  will now be a leaf of the tree.
- Otherwise, pick a dimension  $i \in \{1, \dots, d\}$  [there are many suggested heuristics: picking uniformly at random, choosing in round-robin order, choosing the dimension with largest variance, etc.]
- Let  $m$  be the median of the  $i$ th dimension of the points:  $m = \text{median}[x_1(i), \dots, x_n(i)]$ . Store dimension  $i$  and median  $m$  at node  $v$ . Partition the set  $S$  into  $S_<$  and  $S_>$  according to whether the  $i$ th coordinate of each point exceeds  $m$ . [Note that in some implementations, “median” might be replaced by “mean” or some other value.]
- Make two children of  $v_<$  and  $v_>$ , and recurse on  $[v_<, S_<]$  and  $[v_>, S_>]$ .

Note that the size of the data-structure is linear in the size of the initial pointset. To add a point,  $v$ , to the structure, one simply goes down the tree, querying the indices of  $V$ , and comparing them to the various medians until one reaches a leaf, at which point one will then

split the leaf into two children. The tree will initially be balanced (because we are using the medians of the coordinate values), and hence will have depth  $O(\log n)$ .

Given a node  $v$ , if we want to find the closest point in our  $kd$ -tree structure to  $v$ , we will first go down the tree and find the leaf in which  $v$  would end up. We then go back up the tree, at each juncture, asking “is it possible that the closest point to  $v$  would have ended up down the other path?”. In low dimensions, the answer to this question will often be “no”, and the search will be efficient. For example, in 1 dimension, the leaf node corresponding to  $v$  will *always* contain the closest point to  $v$ . [Think about why this is the case!] In higher dimensions, we might end up needing to explore more leaves. In general, the number of leaves explored before finding the closest point will typically scale *exponentially* with the dimension, ie  $\max(2^d, n)$ , which is why kd-trees are often ill-suited to very high dimensional data.<sup>1</sup>

## 4 The Curse of Dimensionality [might cover at beginning of Lecture 4 depending on time]

Why are high-dimensional spaces often hard to deal with? Why do the running times of many algorithms scale exponentially with the dimension? One answer is that high-dimensional spaces, in some sense, can lack geometry. For example, they can have lots and lots of points with the property that all pairs of points have roughly the same distance.

**Example 4.1** What is the largest number of points that fit in  $d$ -dimensional space, with the property that all pairwise distances are in the interval  $[0.75, 1]$ ?

- $d = 1$ : At most 2 points have this property...if you try to fit a third point, at least one of the 3 pairwise distances will be off.
- $d = 2$ : At most 3 points have this property...if you try to fit a fourth point, at least one of the 6 pairwise distances will be off.
- $d = 100$ : You will be able to fit several thousand points!
- In general, you will be able to fit an exponential number of points (a quick calculation shows that a random set of  $\exp(\sqrt{d})$  will satisfy this property with high probability).

## 5 Take-Home Messages

Below are the highest-level take-home points from this lecture.

---

<sup>1</sup>Of course, if your data is high dimensional, but is mostly contained in some low-dimensional subspace, then kd-trees might perform well provided that the heuristic for choosing the dimension to split on—in the second step of Algorithm 1—is chosen in that subspace.

- Similarity search is a fundamental problem with two common variants: 1) given a set of point, and distance metric, how can we quickly find pairs or sets of similar datapoints? 2) Given a set of point,  $S$ , and distance metric, how do we store the points such that if we are given a new “query” point,  $q$ , we can efficiently find the closest point (or a close point) to  $q$  in set  $S$ .
- Many different useful distance metrics: Jaccard similarity, Euclidean distance, L1/Manhattan distance, cosine similarity, edit distance, etc. Different distance metrics have very different properties. In any application, it is worth thinking about which distance metric is most appropriate.
- One datastructure that enables an efficient similarity search is a kd-tree: a binary tree that partitions space.
- kd-trees work well for low dimensional settings. The usual rule of thumb is that if the dimension,  $d$  is less than 20, or if the number of points is larger than  $2^d$ , then kd-trees will be a reasonable datastructure.
- (Might cover during lecture 4) The curse of dimensionality: high dimensional spaces often have strange properties (e.g. kissing number exponential in dimension), and many geometric algorithms have a runtime that scales exponentially with the dimension.

**Next Time: Dimension Reduction (MinHash, Johnson-Lindenstrauss)!**