COMP 480/580 — Probabilistic Algorithms and Data Structure Feb 19, 2019

Lecture 11

Lecturer: Anshumali Shrivastava *Scribe By: Zichuan Gong*

1 Similarity or Near-Neighbor Search

The near-neighbor search problem is motivated by applications where we want to find a set of similar elements in a dataset. There are many formulations of this problem - we will consider a characteristic example. Near-neighbor algorithms are overwhelmingly common in practice. They form the first stage of many machine learning pipelines and are a useful heuristic for difficult problems (i.e. traveling salesman).

1.1 Similarity and Distance

We suppose that we can compute the distance between the elements of our dataset using some distance function. As an example, consider the Euclidean distance function (or the standard Euclidean metric) for 2D vectors $\mathbf{x} = [x_1, x_2]$ and $\mathbf{y} = [y_1, y_2]$:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

When $\mathbf{x}$ and $\mathbf{y}$ are nearby, the value of x_1 is close to y_1 and x_2 is close to y_2. As a result, the distance $d(\mathbf{x}, \mathbf{y})$ is small. When $\mathbf{x}$ and $\mathbf{y}$ are not close, then the distance is arbitrarily large.

A similarity measure ($\text{sim}(\mathbf{x}, \mathbf{y})$) has the opposite behavior. When two elements $\mathbf{x}$ and $\mathbf{y}$ are nearby, the similarity should be large. When the elements are far from each other, the similarity should be small. There are many ways to get a similarity from a distance function. For reasons we will see later, we want $0 \leq \text{sim}(\mathbf{x}, \mathbf{y}) \leq 1$. One way to do this is to use:

$$\text{sim}(\mathbf{x}, \mathbf{y}) = \frac{1}{1 + d(\mathbf{x}, \mathbf{y})}$$

Later, when we introduce locality sensitive hashing, we will exclusively consider similarity measures without worrying about the corresponding distance function. The main takeaway of this section is that there is an inverse relationship between similarity and distance. *For near-neighbor / similarity search, finding points with high similarity is equivalent to finding points that are close in distance.*

1.2 Example

Problem: Large scale image search.
We have a query image q and a large database (say, the entire internet). We would like to search the database to find similar images. This problem can be interpreted as a similarity search problem.
Goal: We want to find the nearest-neighbor, or the image x^* in the dataset that is most similar to q. We want a fast and accurate solution.
Difficulties:
This is difficult beacuse images are high-dimensional objects. Standard near-neighbor methods do not handle high-dimensional data well.

Solution: Exhaustive search
Given a (relatively fixed) collection of images C, a similarity measure $\text{sim}(x, y)$, and a query, compute

$$x^* = \arg\min_{x \in C} \text{sim}(x, q)$$

The query time is $O(nD)$ per query, where n is the size of C and D is the dimensionality of an image. Querying is a very common operation, and n and D are very large. Therefore, the exact, exhaustive search solution will not work. To obtain better performance over the brute force method, we will allow algorithms that return an approximate solution $\hat{x}$.

2 Space Partitioning Methods

Solution: Trees
Many near-neighbor methods store the database as a tree structure, where the branches of the tree partition the data space. To search for a query, we can perform an efficient search through the space partitions until we find the partition that contains similar points. This solution is exact and works well in low dimensions.
Problem: Efficiency
In high dimensions, space partitioning is not efficient. Even with $D > 10$, the query time is close to exhaustive search.

3 Locality Sensitive Hashing

Locality sensitive hashing provides an approximate solution that is much more efficient in high dimensions.

3.1 Motivating Problem: Autocorrect in Search Engines

Task: We want to correct/suggest a user-typed query in real time. Suppose a user wants to search for "amazon" but actually types "amaozn". We observe that many other users have searched for "amazon" and that sim("amaozn", "amazon") is high, so we suggest this query instead.

More technically, if we store statistically significant (common) query strings in a database D and we are given a new user typed query q, we need to find the closest string $s \in D$ to q. We want to do this in real time, which means we must limit the latency to 20ms. For an exact solution, a cheap distance function takes 400ms to compute, so we cannot afford even one exact similarity computation! In this case, we can use locality sensitive hashing to get an approximate solution and obtain performance 210000 times better than the exact solution.

3.2 Locality Sensitive Hash Functions

A locality sensitive hash function has a collision probability that is related to the distance between elements.
Classical Hashing:

1. If x = y $\rightarrow h(x) = h(y)$
2. If x $\neq$ y $\rightarrow h(x) \neq h(y)$

Locality Sensitive Hashing (LSH):

1. If sim(x,y) is high $\rightarrow \Pr(h(x) = h(y))$ is high.
2. If sim(x,y) is low $\rightarrow \Pr(h(x) = h(y))$ is low.

Note that $\Pr(h(x) = h(y))$ is between 0 and 1. This is why we assumed $0 \leq \text{sim}(x, y) \leq 1$ in Section 1.1. There are many families of similarities that we can use for $\text{sim}(x, y)$ - if a similarity measure has a corresponding LSH function $h(\cdot)$ (and most do), then we can use this method. For our example application, we will introduce the Jaccard similarity measure and the MinHash LSH function.

3.3 Jaccard Similarity Measure

Given two sets: S_1 and S_2, the Jaccard similarity is

$$\text{sim}(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

Example:

$$S_1 = \{3, 10, 15, 19\} \;\; S_2 = \{4, 10, 15\}$$

$$S_1 \cap S_2 = \{10, 15\} \;\; S_1 \cup S_2 = \{3, 4, 10, 15, 19\}$$

Therefore, $\text{sim}(S_1, S_2) = \frac{2}{5}$.

For our application:
Suppose we have sets of characters or strings rather than numbers. In this case, we can use N-grams. A set of N-grams is the set of all contiguous n-character tokens in a string. We will use 3-grams for our application.
Example: "iphone 6"
We take the string "iphone 6" and convert it into set of all contiguous 3-character tokens:

"iphone 6" $\rightarrow$ {"iph", "pho", "hon", "one", "ne ", "e 6"}

4 Minwise Hashing

To obtain the minwise hash (MinHash) value of a set, we hash each item in the set and return the minimum hash value. It is not immediately obvious why we would want to do this, so we first take a step back.

4.1 Random Sampling with Universal Hashing

Question: Given the string "Amazon", the set of 3-grams "Ama", "maz", "azo", "zon", and a random hash function $U : string \rightarrow [0 - R]$, can we get a random element of the set?
Observe that

$$\Pr(h(s) = c) = \frac{1}{R}$$

To get a random element of the set, we can perform random sampling using universal hashing. To do this, we hash every token using U and pick the token that has minimum or maximum hash value.

Example:

$$\{U(\text{Ama}), U(\text{maz}), U(\text{azo}), U(\text{zon})\} = \{10, 2005, 199, 2\}$$

Our choice in this case is "zon".

4.2 Minwise Hashing (MinHash)

Suppose we have a set S with L elements: $S = \{s_1, s_2, ...s_L\}$. To compute the minwise hash value of S, we compute the set of random hash values $U(\cdot)$ of each element and we take the minimum:

$$\text{MinHash}(S) = \min\{U(s_1), U(s_2), ...U(s_L)\}$$

Example:

Suppose we have a document S = {ama, maz, azo, zon, on.}
We generate a random hash function $U_i : String \rightarrow N$
For this example, suppose

$$U_i(S) = \{U_i(\text{ama}), U_i(\text{maz}), U_i(\text{azo}), U_i(\text{zon}), U_i(\text{on.})\} = \{153, 283, 505, 128, 292\}$$

In this case, MinHash = 128.

Properties:

1. Minwise Hashing can be applied to any set.

2. $\Pr(\text{MinHash}(S_1) = \text{MinHash}(S_2)) = \frac{|S_1 \cap S_2|}{|S_1 \cup s_2|}$. That is, the collision property is equal to the Jaccard similarity.

Proof that MinHash is locality sensitive (Property 2):

Consider the set $S_1 \cup S_2$. This set is the union, so it contains all elements of S_1 and all elements of S_2. Let e be the element that has the smallest hash value in $S_1 \cup S_2$ (i.e. if e is in both S_1 and S_2).

Suppose we find $\text{MinHash}(S_1)$ and $\text{MinHash}(S_2)$ using $U(\cdot)$ as described earlier. At least one of these MinHash values has to be $U(e)$. If $e \in S_1$, $\text{MinHash}(S_1) = U(e)$ and likewise if $e \in S_2$, $\text{MinHash}(S_2) = U(e)$.

It is easy to see that $\text{MinHash}(S_1) = \text{MinHash}(S_2) = U(e)$ if and only if $e \in S_2 \cap S_1$.

Suppose that $S_1 \cup S_2$ contains N elements and $S_1 \cap S_2$ contains M elements ($M \leq N$). From our previous discussion on random sampling with universal hashing, it is easy to show that e is randomly selected element of $S_1 \cup S_2$. *To see this, observe that $U(e)$ is the minimum $U(S_1 \cup S_2)$, which from our previous discussion means that e is a randomly selected element of $S_1 \cup S_2$.*

Since $|S_1 \cup S_2| = N$, there are N different possible values for e. Since $|S_1 \cap S_2| = M$, exactly M of these N different possible choices of e will have a collision. That is, there are M

choices for which $\text{MinHash}(S_1) = \text{MinHash}(S_2) = U(e)$. We randomly chose e, so the collision probability is $\frac{M}{N}$ or

$$\Pr(\text{MinHash}(S_1) = \text{MinHash}(S_2)) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$