Algorithms for Data Science: Lecture on Finding Similar Items

Barna Saha

1 Finding Similar Items

Finding similar items is a fundamental data mining task. We may want to find whether two documents are similar to detect plagiarism, mirror websites, multiple versions of the same article etc. Finding similar items is useful for building recommender systems as well where we want to find users with similar buying patterns. In Netflix two movies can be deemed similar if they are rated highly by the same customers.

While, there are many measures of similarity, in this lecture, we will concentrate one such popular measure known as *Jaccard Similarity.*

Definition (Jaccard Similairty)**.** Given two sets S_1 and S_2, Jaccard similarity of S_1 and S_2 is defined as $\frac{|S_1 \cap S_2}{S_1 \cup S_2}$

Example 1. *Let* $S_1 = \{1, 2, 3, 4, 7\}$ *and* $S_2 = \{1, 4, 9, 7, 5\}$ *then* $|S_1 \cup S_2| = 7$ *and* $|S_1 \cap S_2| = 3$. *Thus the Jaccard similarity of* S_1 *and* S_2 *is* $\frac{3}{7}$.

1.1 Document Similarity

To compare two documents to know how similar they are, here is a simple approach:

- Compute k shingles for a suitable value of k k shingles are all substrings of length k that appear in the document. For example, if a document is *abcdabd* and $k = 2$, then the 2-shingles are ab, bc, cd, da, ab, bd.

- Compare the set of k shingled based on Jaccard similarity. One will often map the set of shingled to a set of integers via hashing. *What should be the right size of the hash table?* Select a hash table size to avoid collision. For example, if $k = 9$ then considering the 26 letters and white space the possible number of k-shingles is 27^9. Consider a hash function that maps shingles to a range of $1, 2, ..., 27^9$.

1.2 Min-wise Hashing

There are multiple challenges when dealing with large documents, especially when the number of such documents is itself quite large. In that case, the number of shingles from a document could be really large, and storing them for all the documents in the main memory for similarity comparison is infeasible.

We rather want to compute a small fingerprint and store that instead for every document, such that if we compare the fingerprints, then with high probability we will be able to compute the Jaccard similarity of the original set of shingles.

We now describe such an approach, popularly known as *min-hash* computation.

We compute a single min-hash of a set of shingles S as follows. Generate a random permutation σ_1 of all possible shingles (e.g. for $k = 9$, it is a permutation of 1 to 27^9) and report the element in S that appears first in σ_1. Do you need to generate the entire permutation before computing the min-hash?

The fingerprint of S consists of t minhashes computed from t randomly generated permutations $\sigma_1, \sigma_2, ..., \sigma_t$. To compute an estimate of the Jaccard similarity of S_1 and S_2, we simply compute the number of min-hashes that match–if that number is r, the estimated Jaccard similarity is $\frac{r}{t}$.

The question is then how large t should be? [Will be posted after the homework]

1.3 Applications of Min-hash

Source: Wikipedia A large scale evaluation has been conducted by Google in 2006 to compare the performance of Minhash and Simhash algorithms. In 2007 Google reported using Simhash for duplicate detection for web crawling and using Minhash and LSH for Google News personalization. Check out the Description from blogs:

http://matthewcasperson.blogspot.com/2013/11/ minhash-for-dummies.html

http://robertheaton.com/2014/05/02/ jaccard-similarity-and-minhash-for-winners/: matching twitter users

http://blog.jakemdrew.com/2014/05/08/ practical-applications-of-locality-sensitive-hashing-for-unstructured-data/

There are implementations available in github. *https://github.com/rahularora/MinHash* which may have bugs.

2 Locality Sensitive Hashing

Having fingerprints allow us to compute similarity between any pair of documents fast. However, computing pair-wise similarity for all document pairs to find all documents above say $\frac{1}{10}$ Jaccard similarity will be highly time consuming.

A data structure that helps us here to reduce the running time significantly is *locality sensitive hashing*. At a high-level locality sensitive hashing is a hashing mechanism such that items with higher similarity have higher probability of colliding into the same bucket than others. We will use multiple such hash functions and only compare the documents that are hashed to the same bucket. We would need to worry about (i) false positive: when two "non-similar" items hash to the same bucket–this increases search time and (ii) false negative: when two similar items do not hash to the same bucket under any of the chosen hash functions from the family. We will now see details about locality sensitive hashing (LSH) in this section.

2.1 Applications

LSH has found wide-spread applications in

- Near-duplicate detection
- Hierarchical clustering
- Genome-wide association study
- Image similarity identification
- VisualRank
- Gene expression similarity identification
- Audio similarity identification
- Nearest neighbor search
- Audio fingerprint

- Digital video fingerprinting
- Anti-spam detection
- Security and digital forensic applications

Check out: *http://www.mit.edu/ andoni/LSH/*
and
https://github.com/triplecheck/TLSH

2.2 Approximate Near Neighbor Search

The problem that we will tackle with LSH is approximate near neighbor search. We start with by defining *near neighbor problem.*

Definition (Near Neighbor Problem)**.** Given a set of points V, a distance metric d and a query point q, is there any point x close to query point q such that $d(x, q) \leq R$

The problem is easy to solve efficiently in low dimension, e.g. via voronoi diagram construction without requiring to go over the entire data set. However, the complexity increases exponentially in dimension.
We therefore, relax the problem and ask for *approximate near neighbor problem.*

Definition (Approximate Near Neighbor Problem)**.** Given a set of points V, a distance metric d and a query point q, the (c, R)-approximate near neighbor problemm requires if there exists a point x such that $d(x, q) \leq R$, then one must find a point x' such that $d(x', q) \leq cR$ with probability $> (1 - \delta)$ for a given $\delta > 0$.

We will use LSH to solve this problem.

Definition (Locality Sensitive Hashing)**.** A family of hash functions $\mathcal{H}$ is said to be (c, R, p_1, p_2)-sensitive LSH for a distance metric d if it satisfies the following conditions.

1. $\Pr_{h \sim \mathcal{H}}[h(x) = h(y)] \geq p_1$ for all x and y such that $d(x, y) \leq R$.
2. $\Pr_{h \sim \mathcal{H}}[h(x) = h(y)] \leq p_2$ for all x and y such that $d(x, y) > cR$.
3. $p_1 > p_2$

Example 2 (Hamming Distance)**.** *Let* $V \subseteq [0, 1]^n$ *and* $d(x, y) =$ *Hamming distance between* x *and* y*. Let* R *and* cR *both be much less than* n*. Define* $\mathcal{H} = \{h_1, h_2, ..., h_n\}$ *such that* $h_i(x) = x_i$

For the above $p_1 \geq 1 - \frac{R}{n}$ since when the distance is at most R, the number of bits where the two vectors differ is at most $n - R$. On the other hand, $p_2 \leq 1 - \frac{cR}{n}$ since when the distance is $> cR$, the number of bits where the two vectors differ is at least $n - cR$.

Example 3 (Jaccard Distance)**.** *Define Jaccard Distance between two sets* x *and* $y \subseteq [1, n]$ *as* $1 - Jaccard(x, y)$*. Define* $\mathcal{H} = \{h_1, h_2, ..., h_{n!}\}$ *such that* $h_i(x)$ *corresponds to min-hash of* x *for the* i*th permutation of* $[1, n]$ *in the lexicographic order.*

We have $p_1 \geq 1 - R$ and $p_2 \leq 1 - cR$. Why?

Example 4 (Cosine Distance)**.** *The cosine of two non zero vectors* $\mathbf{a}$ *and* $\mathbf{b}$ *can be derived by noting* $\langle \mathbf{a}.\mathbf{b} \rangle = ||\mathbf{a}||||\mathbf{b}||cos\theta$ *where* $\mathbf{a}.\mathbf{b}$ *denote the inner dot product of the two vectors* $\mathbf{a}$ *and* $\mathbf{b}$.

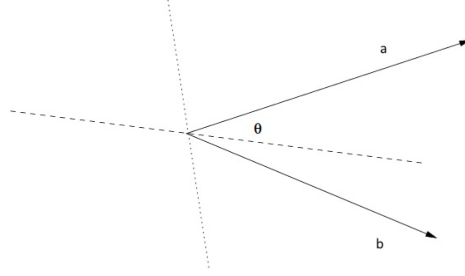

Figure 1: Pictorial representation of LSH for Cosine distance

Cosine similarity of **a** and **b** is defined as $\frac{\langle \mathbf{a}.\mathbf{b} \rangle}{||\mathbf{a}||||\mathbf{b}||}$.

Think of a point as a vector from the origin $(0, 0, ..., 0)$ to its location. Two points' vectors make an angle, whose cosine is the normalized dot product of the vectors: $\frac{\langle \mathbf{a}.\mathbf{b} \rangle}{||\mathbf{a}||||\mathbf{b}||}$. The cosine distance is simply $\theta = arccos(\frac{\langle \mathbf{a}.\mathbf{b} \rangle}{||\mathbf{a}||||\mathbf{b}||})$. θ can range from 0 to 180 degrees.

To construct LSH for the cosine distance simply do the following: pick a random hyperplane v, which determines a hash function h_v with two buckets. Define $h_v(x) = +1$ if $\langle v.x \rangle \geq 0$ and $h_v(x) = -1$ if $\langle v.x \rangle < 0$.

Suppose $\theta = arccos(\frac{\langle \mathbf{a}.\mathbf{b} \rangle}{||\mathbf{a}||||\mathbf{b}||})$. Look into the plane defined by a and b and let u denote the normal vector perpendicular to the random hyperplane v. The hashed value of a and b will be different if a and b lie on two sides of u–this happens with probability $\frac{\theta}{180}$. Hence, we have $p_1 = 1 - \frac{R}{180}$ and $p_2 = 1 - \frac{cR}{180}$ where $0 \leq cR \leq 180$. A pictorial representation is given below.

For the above, it is sufficient to construct v such that each component is ± 1 with equal probability.

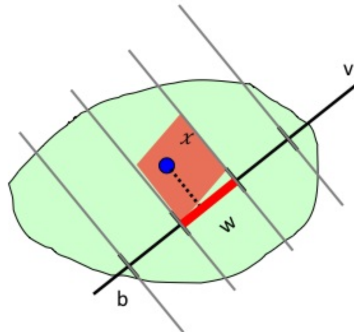

Figure 2: Pictorial representation of LSH for Euclidean distance

Example 5 (Euclidean Distance)**.** *Here hash functions correspond to random lines. Partition the line into buckets of width w starting from a random shift b. Hash each point to the bucket containing its projection onto the line. Nearby points are always close; distant points are rarely in same bucket.*

Specifically, we do not need to consider all random lines–we rather select a vector v whose each component is a gaussian random variable with 0 *mean and standard deviation* 1*. $h_v(x) = \lfloor \frac{v.x+b}{w} \rfloor$ where b is chosen uniformly at random in $[0, w]$.*

3 General Scheme of using LSH to solve (c, R)-Near Neighbor Problem

Suppose we are given a LSH family of hash functions $\mathcal{H}$ which is (c, R, p_1, p_2)-sensitive. We will construct L composite hash functions from it as follows. Select $h_{i,j}$ uniformly and randomly

from $\mathcal{H}$ for $i = 1, 2, .., K$ and $j = 1, 2, .., L$. Define the composite hash functions $g_1, g_2, .., g_L$ as follows:

$$g_j = \langle h_{1,j}, h_{2,j}, ..., h_{K,j} \rangle$$

3.1 Preprocessing.

1. Create a hash table $bucket_j$ for g_j for $j = 1, 2, .., K$.
2. For all $x \in V$ and for all $j \in [L]$, add x to $bucket_j(g_j(x))$

Time for preprocessing is $O(NKL)$ where N denotes the number of data points.

3.2 Query(q)

1. for $j = 1, 2, .., L$
   - for all $x \in bucket_j(g_j(q))$ do
     - if $d(x, q) \leq cR$ then return x
2. Return none

Time for querying is $O(KL + NLF)$ where F denotes the probability that for any j $g_j(x) = g_j(q)$ when $d(x, q) > cR$. Then NF denotes the expected number of entries that are more than cR distance away from q but are in the same bucket as q under g_j. Since there are L composite hash functions, the total number of such bad items is on expectation $O(NLF)$. We would like to choose the parameters K and L in a way such that $NLF \approx KL$. In fact, we will set K as $O(\log n)$ and $NF = 1$.

There are two parts to the analysis (i) computing success probability and (ii) analyzing the time complexity.

Computing success probability Suppose there is a point x such that $d(x, q) \leq R$. Then the algorithm will be successful if it finds points y such that $d(y, q) \leq cR$. Clearly, this success probability is at least as high as the probability that the algorithm finds the actual point x.

Hence, we have

$$\begin{aligned} \Pr[Success] &\geq \Pr[\exists j \text{ such that } g_j(x) = g_j(q) \mid d(x, q) \leq R] \\ &\geq 1 - (1 - p_1^K)^L \end{aligned}$$

Set $L = \frac{1}{p_1^K}$, then

$$\Pr[Success] \geq 1 - (1 - \frac{1}{L})^L \approx 1 - \frac{1}{e}$$

Question. Suppose you want your success probability to be $1 - \frac{1}{n}$, what should be the value of L as a function of p_1 and K?

Computing time complexity To compute the querying time, we need an upper bound on F.

$$F = \Pr(g_j(y) = g_j(q) \mid d(y, q) > cR) = p_2^k$$

Hence the time requirement is $O(KL + NLp_2^K)$. To simplify calculation, we will select $Np_2^K = 1$ or

$$N = \frac{1}{p_2^K} = \left(\frac{1}{p_1}\right)^{\log_{1/p_1}(1/p_2^K)} = \left(\frac{1}{p_1^K}\right)^{\log_{1/p_1}(1/p_2)} = \left(\frac{1}{p_1^K}\right)^{\frac{\log 1/p_2}{\log 1/p_1}} = L^{\frac{\log 1/p_2}{\log 1/p_1}}$$

Or, in other words,

$$L = N^{\frac{\log 1/p_1}{\log 1/p_2}}$$

Let us use the notation $\rho = \frac{\log 1/p_1}{\log 1/p_2}$. We have $L = N^\rho$.

Now, $Np_2^K = 1$ gives $\frac{1}{p_2^K} = N$ or $K = \frac{\log N}{\log \frac{1}{p_2}}$.

Example 6. *If one has* $p_1 = 0.1$ *and* $p_2 = 0.01$*, then* $\rho = \frac{1}{2}$

3.3 Practical Consideration

The range of values of the composite hash functions could be R^K if R is the range of the LSH hash family. Therefore, it may not be practical to maintain a hash table where there is an index for every possible value that the composite hash functions can take. Suggest a way to overcome this difficulty.