**COMP 480/580 — Probabilistic Algorithms and Data Structures** Sep 29, 2022

# Lecture 12: Minwise Hashing, Near-Duplicate Deletion, and LSH 1

*Lecturer: Anshumali Shrivastava* *Scribe By: A. Cantu, A. Walia, B. Thomas*

## 1 Motivating Problem: Large Scale Search

Let's imagine a scenario where we have to search the internet for similar images given a query image. When the background details (parameters) of the image aren't mentioned and the search has to be fast and (mostly) accurate, this problem become complex.

As we know the images are heavy objects: A 24 megapixel image consumes 72,000,000 bytes which is 68.7MB. Traversing through the entire database looking for similar images would be severely time consuming.

**Solution: Hashing Algorithm** (Notion for search problem is different - Similarity)

In general when we talk about search, we look for exact matches. Exact matches are easy i.e, they can be performed in O(1) time complexity with a perfect hash function. However, the internet is not so straightforward. Any data - such as an image - would have different metadata across different web pages even though the content may be same. To a hash function, there is no difference between between slightly different and significantly different keys. We can define this problem more formally as follows.

Given a query point q, the goal is to find the point in the set P that is closest to this query point. Naively, this requires O(n) comparisons, since we have to compare q to each point in P. In the setting where the number of points is really large, it is not computationally feasible to use the brute-force algorithm.

## 2 Similarity or Near-Neighbor Search

Given a fixed collection $c$ and a similarity (or distance) metric $sim$, for any query $q$, compute

$$x^* = argminsim(q, x), \{x \in c\}$$

This is a linear problem of the order $O(nD)$, where $n$ is the *size* and $D$ is the *dimension.*

**Setting:** Query is a very frequent operation (in millions/sec) i.e, $n$ & $D$ are very large
**Hope:** Already processed collections
**Approximation:** It is fine as long as it gives reasonable result

### 2.1 Motivating Problem: Search Engines

Lets look at the problem of quick auto-correction in search engines. Lets say we have a query that is mistakenly typed as "redshoes" instead of "red shoes". Mistakes like these are quite common and we expect a robust search engine to handle such queries and return relevant results

in a short time frame. Assume we are given a database of 50 million statistically significant queries. Our problem now transforms to mapping our query to a key in the database. How do we map a wrongly typed query "redshoes" to "red shoes"?

One approach to the solution would be to attempt to find the most similar key to the query using a cheap distance function. The time complexity for the cheapest distance calculating algorithm would be at least 400s excluding the time taken for sorting. For edit distance, it would be hours. According to studies, for every 100ms lag in latency, one can expect to lose 1% of their customers. Any realistic latency requirement for such an algorithm would be within $20ms$.

**Can we do better i.e, in 2ms or 21000x faster?**
**Magic of Hash Function**: Could we construct a hash function that helps us in quickly finding a similar object? We would need to construct a hash function such that its probability of collision increases as the similarity between two object increases.

We formulate the idea of such a hash function in the upcoming section.

## 2.2 Locality Sensitive Hashing (LSH)

- Classical Hashing
  - if $x = y$ then $h(x) = h(y)$
  - if $x \neq y$ then $h(x) \neq h(y)$
- LSH (randomized relaxation)
  - if $sim(x, y)$ is high then $P(h(x) = h(y))$ is high
  - if $sim(x, y)$ is low then $P(h(x) = h(y))$ is low

We can't work with exact hash function, so we need some evidence (collision) using probability. From the above classical hashing and LSH we can conversely conclude that:

$$h(x) = h(y) \text{ implies } sim(x, y) \text{ is high}$$

## 2.3 Notion of Similarity: Jaccard

Let's take two sets $S_1$ and $S_2$

$$J(S_1, S_2) = \frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|} = \frac{\text{cardinality of intersection}}{\text{cardinality of union}}$$

**Simple Example:**

$$S_1 = \{3, 10, 15, 19\},\ S_2 = \{4, 10, 15\}$$
$$J(S_1, S_2) = \frac{\{10,15\}}{\{3,4,10,15,19\}}$$
$$\text{Thus, } J = \frac{2}{5}$$

**Extending Jaccard to strings using N-grams**
N-grams represents set of each string in a contiguous manner. Universe is all possible set elements. 3-gram for "iPhone 6" gives us {"iPh", "Pho", "hon", "one", "ne ", "e 6"}

Similarity between two strings can be found using Jaccard Distance between the sets of the two strings.

**Example:** "amazon vs anazon"
"amazon" $\longrightarrow \{"ama","maz","azo","zon"\}$
"anazon" $\longrightarrow \{"ana","naz","azo","zon"\}$

$J("amazon","anazon") = \frac{\{"azo","zon"\}}{\{"ama","ana","maz","naz","azo","zon"\}} = \frac{1}{3}$

### 2.4 Random Sampling

**Problem:** Given a random universal hash function, $U$ and string, $s$,

$$U : s \longrightarrow [0 - R]$$

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

Using the hash function U, how to randomly sample element from the set?
This can be done by hashing every element of the set and then taking the smallest or biggest hash value. To generate a random number continuously we need access to a random hash function with a random, different seed.

## 3 Minwise Hashing

The MinHash method was invented by Andrei Broder, when he was working on Altavista search engine. The method works as follows. Let us take a random universal hash function ( $U_i : strings \longrightarrow N$) which maps strings 0 - N and hash all elements in the set.

$U_i(s) = \{U_i(ama), U_i(maz), U_i(azo), U_i(zon)\}$

After processing the hash function, let's assume the hash values to be:

$U_i(s) = \{ 154 , 223 , 256 , 505\}$

Minhash for $U_i$ = min $h_i(s)$ = 154

Every time we want to generate a new minhash value, we'll generate a new universal hash function and compute the minimum.

### 3.1 Properties

- For any set of objects $S_1$ & $S_2$. Probability of hash collision is exactly equal to Jaccard index

P( Minhash($S_1$) = Minhash($S_2$) ) = $\frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|}$

If similarity between the two sets increases then the probability of collision also increases.

- Take two sets $S_1$ & $S_2$. Hash every element of $S_1 \bigcup S_2$ and find the minimum hash value. The chance of having the same minimum value after this permutation is equal to the number of common elements proportional to the union. If the minimum belongs to $S_1 \bigcap S_2$, then MinHash of both sets will be the same i.e., if c $\in |S_1 \bigcap S_2|$ then MinHash($S_1$) = MinHash($S_2$). If it did not belong to $S_1 \bigcap S_2$, the MinHash of the two sets will be different.

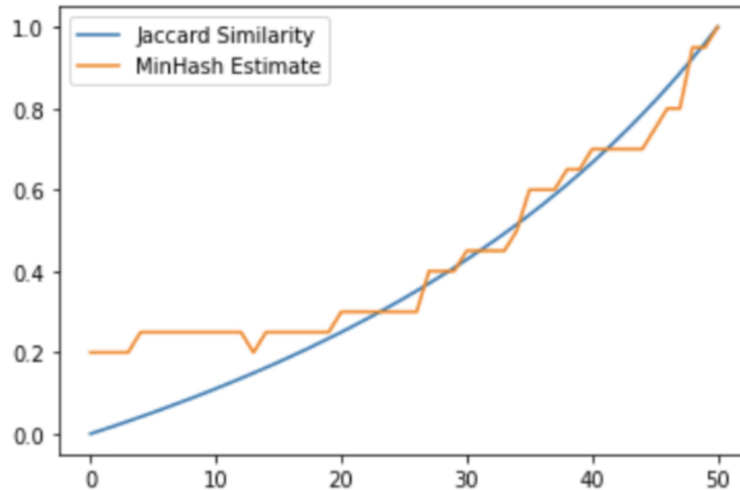


***Figure 1:*** *Simulated result of Jaccard Index and MinHash estimate*

### 3.2 Estimate Similarity Efficiently

How can we estimate $J$ if we have 50 minhashes of $S_1$ & $S_2$ ?

Instead of dealing with large sets, which requires a lot of computing time and memory, MinHash can provide a sketch to approximate this measure in a scalable way by computing a small fixed sized sketch which represents each large set.

**Example:** Using this technique, to estimate how many collisions out of 50 there are:

$$Variance = \frac{J(1-J)}{\text{number of hashes}}$$

from Beronoulli distribution. If $J = 0.8$, then

$$Error = \sqrt{\frac{0.16}{50}} \approx 0.05$$

Thus, if $J = 0.8$, then the similarity is estimated within 0.05 error.

### 3.3 Parity of Minhash

Recall Minhash for $U_i = 154$. Instead of storing 154 (32 bits), we can just store whether the Minhash was odd or even. This property is known as parity. This results in only 1 bit of information.

$$P(Parity(Minhash(S_1)) = Parity(Minhash(S_2)))$$
$$= \frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|} + (1 - \frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|}) * 0.5$$
$$= 0.5 * (1 + \frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|})$$

With $P = J$, minhashes are equal and hence parities must be equal. With $P = 1 - J$, the two minhashes are not equal, but the parity can still be equal with probability 0.5. You can get a good estimate of Jaccard similarity from the parity of minhash.

Using the new probability $P' = 0.5 * (1 + J)$ we can calculate the new $Variance'$ for parity MinHash:

$$Variance' = \frac{P'(1 - P')}{\text{number of hashes}}$$

Replacing the value of $P'$ calculated above we get:

$$Variance' = \frac{0.5 * (1 + J)(1 - (0.5 * (1 + J))}{\text{number of hashes}}$$

$$= \frac{0.5 * (1 + J)(1 - 0.5 - 0.5 * J)}{\text{number of hashes}}$$

$$= \frac{0.5 * (1 + J)(1 - 0.5 * J)}{\text{number of hashes}}$$

Using the same case as before of J = 0.8 and number of hashes = 50:

$$Variance' = \frac{0.5 * (1.8)(1 - 0.5 * 0.8)}{50}$$
$$= \frac{(0.9)(0.6)}{50}$$
$$= \frac{.54}{50} = 0.0108$$
$$Error = \sqrt{0.0108} \approx 0.104$$

As we can see the error is about twice as much as in the case of comparing Min-Hashes, however, we save significantly on memory. Increasing the number of hashes will can make these error rates comparable by only occupying slightly more memory.