COMP 480/580 — Probabilistic Algorithms and Data Structure Oct 4, 2022

Lecture 13

Lecturer: Anshumali Shrivastava *Scribe By: Ben Harris, Cheng Peng*

1 Review

In the previous lecture, we introduced Near-Neighbor Search and the motivating problem of Efficient Search Engines. The brute force method of calculating similarity between the input and all items is far too slow to be used in practice, so we explored alternatives using hashing.

In classical hashing, we define a hash function such that if $x = y \rightarrow h(x) = h(y)$ and conversely if $x \neq y \rightarrow h(x) \neq h(y)$. However, this technique provides no assurance that similar outputs will be mapped to similar values in the output space. In fact, modern day hash functions are designed with the opposite effect (minor changes to inputs lead to major variations of outputs) as a way to protect against sophisticated attacks.

In their seminal paper "Similarity Search in High Dimensions via Hashing", Gionis, Indyk, and Motwani introduced Locality Sensitivity Hashing (LSH). They defined a whole new class of hash functions (denoted sim) with the property that the probability of a collision between two values directly correlate to the similarity of the respective items. In other words, $sim(x, y)$ is high $\rightarrow Pr(h(x) = h(y))$ is high and $sim(x, y)$ is low $\rightarrow Pr(h(x) = h(y))$ is low. Conversely, $h(x) = h(y) \rightarrow sim(x, y)$ is high and $h(x) \neq h(y) \rightarrow sim(x, y)$ is low.

1.1 Jaccard Similarity

Before we go any further, we need to define the notion of similarity.

For demonstration purposes, we will restrict the values to strings and convert them into a set of 3-grams (all contiguous 3-characters tokens). For example, $amazon \rightarrow \{ama, maz, azo, zon\}$ and $amazing \rightarrow \{ama, maz, azi, zin, ing\}$.

For comparing two strings, we will calculate the Jaccard Similarity (cardinality of the intersection divided by the cardinality of the union) over the 3-gram sets. For example:

$$J(amazon, amazing) = \frac{|\{ama, maz, azo, zon\} \cap \{ama, maz, azi, zin, ing\}|}{|\{ama, maz, azo, zon\} \cup \{ama, maz, azi, zin, ing\}|}$$
$$= \frac{|\{ama, maz\}|}{|\{ama, maz, azo, zon, azi, zin, ing\}|} = \frac{2}{7}$$

1.2 Minwise Hashing

How does this relate to similarity search? Let's examine the relationship between Minwise Hashing and Jaccard Similarity.

Claim: $Pr(Minhash(S_1) = Minhash(S_2)) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = J$

Proof: Let us consider a hash function U which is used to calculate the minwise hash of the sets S_1 and S_2. Let e be the element that has the smallest hash value in $S_1 \cup S_2$. It's easy to see that $Minwise(S_1) = U(e)$ iff $e \in S_1$ and $Minwise(S_1) = U(e)$ iff $e \in S_1$. However, it is also possible for $Minwise(S_1) = Minwise(S_2) = U(e)$ iff $e \in S_1 \cap S_2$. If we say $|S_1 \cap S_2| = N$

and $|S_1 \cup S_2| = M$, then there are M possible choices for e with N of which resulting in a collision, so the probability of a collision is $\frac{N}{M} = J$.

2 Extensions of Minwise Hashing

2.1 Parity of Minwise Hashing

As demonstrated earlier, Minwise Hashing typically returns an integer. However, if we reduce that integer to just a parity bit, we can still obtain information about strings S_1 and S_2's similarity. Let us define a parity function L such that $L(A) = Minhash(A)\ \%\ 2$. Note that if $Minhash(S_1) = Minhash(S_2)$, the parity of minhash $L(S_1)$ and $L(S_2)$ is guaranteed to be equal. If $Minhash(S_1) \neq Minhash(S_2)$, there is a 50% chance that $L(S_1) = L(S_2)$. So, we have:

$$Pr(L(S_1)) = L(S_2))) = J(S_1, S_2) + 0.5 * (1 - J(S_1, S_2)) = 0.5 * (1 + J(S_1, S_2))$$

Using multiple independent hash functions, we can compute, say, 50 $L(S_1)$ and $L(S_2)$ values. Based on how many $L(S_1) = L(S_2)$, we can estimate the similarity between S_1 and S_2 (i.e. $J(S_1, S_2)$) with a much lower storage cost compared to storing 50 integer values in Minwise Hashing. 50 parity bits, in total, takes up $50/8 \approx 7$ bytes, which is less than the cost of storing 2 integers. Additionally, the storage cost does not depend on how heavy the n-gram set or the length of the string is. The string can have 100, 1000, 10000 characters, but the storage cost is the same for similarity estimation.

2.2 Maxwise Hashing

Other than taking the min, some implementation may choose to take the max. Since min and max have the same probability distribution, taking the max should have identical sampling effects.

2.3 One Permutation Hashing

Despite its many benefits, in industry practice, *minwise hashing* is not very desirable. Li et al. (2012) states that pre-processing for a dataset containing 350,000 samples and 16 million features can take up to 6000 seconds when 500 hash functions are used. In industrial applications, the datasets are much bigger. Hence, the pre-processing can be much more expensive.

The inefficiency of minwise hashing comes from the fact that only the minimum element is used after the algorithm scans all nonzero elements in a set. In response to the inefficiency, Li et al. (2012) proposed *one permutation hashing*. The idea is to hash the strings using some random $U_i : Strings \to N$, divide the space $[0, N]$ into k bins and take the minimum of each. This methodology allows sampling with $\frac{1}{k}$ as much pre-processing costs as the original min-wise hashing.

To give an example. Consider $S = \{ama, maz, azo, zon, on.\}$ and $U_i : Strings \to [0, 600)$. Suppose $U_i(S) = \{153, 283, 505, 128, 292\}$ and $k = 4$. We have four bins with identical size [0,150), [150, 300), [300, 450), [450, 600). If we take the minimum of each, then the selected elements are [128, 153, $\star$, 505]. Note that we have an empty bin denoted by '$\star$'. How to handle the empty bins is the key of *one permutation hashing*.

Shrivastava & Li (2014) introduced a "rotation" scheme that assigns new values to all the empty bins. The idea is that for every empty bin, the scheme borrows the value of the closest

non-empty bin in the clockwise direction (circular right hand side) added with offset C. In the previous example, the "rotation" scheme would take $505 + C$ to fill the empty $\star$.

3 Minwise Hashing vs Random Sampling

Technique: Pick some random words from two documents and compare them.

Claim: Minwise Hashing is significantly more accurate than Random Sampling.

Argument: We can view Random Sampling as asking a series of "yes" or "no" questions on a document. Say you have a document on Machine Learning, and you want to know how similar your document is to the other person's document. You probably want to start by asking questions like "Does your document contain the word 'computer?"" If the answer is "yes," you get some information, but if the answer is "no," you know close to nothing about the document. Random Sampling gives you keywords from two documents, but if the keywords don't match, you get close to no information.

For Minwise Hashing, suppose the word "computer" is hashed to 128 and you get 128 when you ask for min-hash from the document. You not only know the word "computer" is in the document, you also know that document doesn't contain any word that hashes to below 128. This gives you much more information than Random Sampling. If the two documents have the same min-hash, you know they are similar and contains no word that would hash to a lower value. Even if the two documents' min-hash values don't match, the values themselves allow for a drastic reduction in the search space.

4 LSH via Binning

In LSH, we want to create hash functions such that if $|x-y| < \delta$ for some small δ, $h(x) = h(y)$ with high probability. One way to achieve this is random binning.

Say we want a locality sensitive hash function $h : [0, 100) \to [0, 10)$. We could randomly generate offsets $o_1, o_2, ..., o_9$ s.t. $\mathbb{E}[o] = 10$. The hash function h maps every number $n \in [\sum_{k=1}^{i} o_k, \sum_{k=1}^{i+1} o_k]$ to i. For example, suppose $o_1 = 9$ and $o_2 = 22$. Then, every number $n \in [0, 9)$ is mapped to 0. Every number $n \in [9, 22)$ is mapped to 1.

Notice that we use random binning instead of deterministic binning. This is because we want both $x + \delta$ and $x - \delta$ hashed to the same value as x with high probability regardless of x. If we use deterministic binning in the previous example and partition [0,100] into intervals of identical length, we could get a hash function $h(x) = x/10$. Notice that if we take $x = 10$, $x' = 9.99$ is hashed to a different bin as x whereas $x'' = 15$ is hashed to the same bin as x although $|x' - x| < |x'' - x|$. We want to avoid this scenario where some x has deterministicly bad performance by introducing randomness via random binning.

5 Search Engine using LSH

5.1 Overview

How do we build a search engine using *LSH*? Modern search engines perform the task of *near-neighbor search*. That is, identifying a set of data points that are "most similar" to the query data point. Using *LSH*, we can easily query the "most similar" data points in sublinear time (i.e., no need to scan all data points).

Suppose we have a data set R^D. We can hash the data set using *LSH* and use the hash values to index the data. Say we have two hash functions $h_1, h_2 : R^D \to \{0, 1, 2, 3\}$. We can create a hash table with $4^2 = 16$ entries. For each $x \in R^D$, we compute $h_1(x)$ and $h_2(x)$, store the pointer to x (x's memory location) in bucket $(h_1(x), h_2(x))$. In the example below, the orange item x has hash values $h_1(x) = 0$, $h_2(x) = 1$. Since 0 and 1 can be represented by 00 and 01 using bits, the pointer to x is put into bucket 0001.

When we want to query the items similar to q, we can directly go to the bucket indexed by $(h_1(q), h_2(q))$ and return all the elements in the bucket. In the example below, if we want to find elements similar to the orange item, we directly go to bucket 0001 and return all the elements there. Since collision implies high similarity in *LSH*, we are guaranteed to return elements most similar to q.

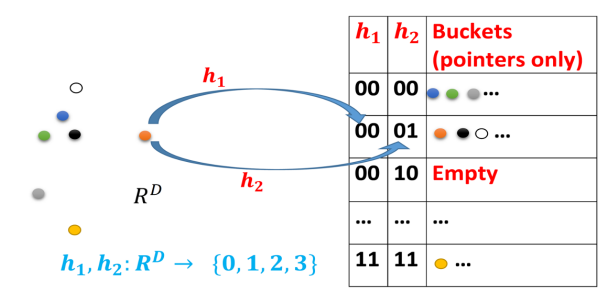

Figure 1: Near neighbour search illustration

5.2 Using Multiple Independent Hash Tables

5.2.1 Idea

In the previous example, we use two hash functions h_1 and h_2. In reality, we can use any number of hash functions and any number of hash tables. In the example below, we use k hash functions and L hash tables. For every query, we return the union of the L buckets.

Why do we use L buckets? Notice that if we have $k = 10$ hash functions for each table, the probability that two items x and y land in the same bucket is $sim(x, y)^{10}$. When $sim(x, y) = 0.9$, $Pr(Buckets(x) = Buckets(y)) = 0.35$. There's a good chance we might miss some candidates. Hence, we use L hash tables to include more candidates. When we use $L = 10$ hash tables, $Pr(y \text{ is retrieved}) = 1 - (1 - sim(x, y)^{10})^{10} = 1 - (1 - 0.9^{10})^{10} = 0.986$.

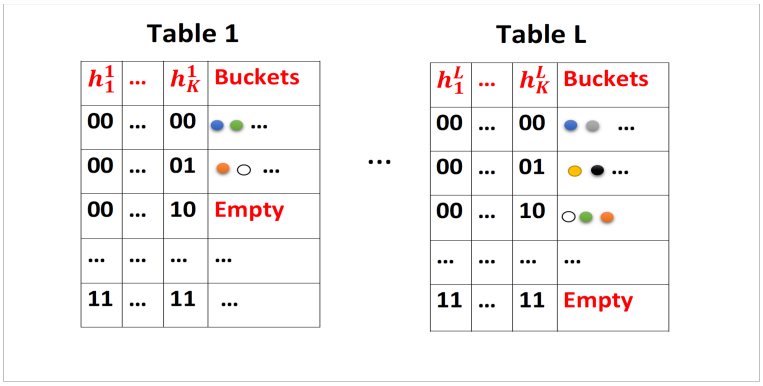

Figure 2: Near neighbour search with L independent hash tables

5.2.2 The LSH Algorithm

Algorithm 1: *Preprocess_Database*(D, *hash_functions*, *hash_tables*)

Input : D, *a data set.*
hash_functions, $K * L$ *independent hash functions* h_1^1, h_2^1 ... h_k^1, h_1^2 ... h_k^L.
hash_tables, L *independent hash tables* T_1, T_2 ... T_L

Output: *None.*

```
for x ∈ D do
    for i from 1 to L do
        Put the pointer of x into T_i[h_1^i(x), h_2^i(x), ..., h_k^i(x)];
```

Algorithm 2: *Query*(q, *hash_functions*, *hash_tables*)

Input : q, *a query.*
hash_functions, $K * L$ *independent hash functions* h_1^1, h_2^1 ... h_k^1, h_1^2 ... h_k^L.
hash_tables, L *independent hash tables* T_1, T_2 ... T_L

Output: *None.*

```
S ← empty set;
for i from 1 to L do
    S = S ∪ T_i[h_1^i(q), h_2^i(q), ..., h_k^i(q)];
Get the best elements from S based on similarity with q;
```

5.2.3 Analysis of K and L

As previously noted, K controls the quality of the bucket and L controls the failure probability (i.e. the probability that a similar item is not retrieved). The key is to find a balance between K and L.

Let $p_x = Pr(minhash(x) = minhash(q)) = Jaccard$.

- p_x^K is the probability that x is in a bucket mapped by q in hash table 1.
- $1 - p_x^K$ is the probability that x is not in a bucket mapped by q in hash table 1.
- $(1 - p_x^K)^L$ is the probability that x is not in any of the L buckets. Or x is not retrieved.
- $1 - (1 - p_x^K)^L$ is the probability that x is retrieved.

$1 - (1 - p_x^K)^L$ is a monotonic increasing function of p_x. Thus, the higher the p_x, the more likely it is that x is retrieved. With $K = 5$ and $L = 10$, if $p_x > 0.8$, probability of retrieval > 0.98; if $p_x < 0.5$, probability of retrieval < 0.2.

The general rule of thumb is to choose $K \approx log(n)$ and $L \approx \sqrt{n}$. Increases in K decreases the candidates retrieved exponentially whereas increases in L increases the candidates linearly.

References

[1] Gionis, A., Indyk, P., & Motwani, R. (1999). Similarity search in high dimensions via hashing.

[2] Li, P., Owen, A., & Zhang, C. H. (2012). One permutation hashing for efficient search and learning.

[3] Shrivastava, A., & Li, P. (2014). Densifying one permutation hashing via rotation for fast near neighbor search.

[4] Shrivastava, A., & Li, P. (2014). Improved densification of one permutation hashing.