COMP 480/580 — Probabilistic Algorithms and Data Structure Feb 21, 2019

Lecture 12

Lecturer: Anshumali Shrivastava *Scribe By: Yuqiu Wang & Xiaoyong Zhang*

In the previous lecture, we introduced the near-neighbor search problem and locality sensitive hashing. We also defined MinHash, a locality sensitive hash function with a collision probability equal to $J(A, B)$. In this lecture, we will discuss an additional property of MinHash and we will explain how to use locality sensitive hashing for search problems.

1 Review: Jaccard Similarity

Suppose we have two sets A and B and we want to quickly estimate the similarity between A and B. In the previous set of notes, we introduced the Jaccard similarity index:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Where $|\cdot|$ is the cardinality (or number of elements) operation. The intuition behind the Jaccard index is that when A and B are the same, the intersection $A \cap B$ and the union $A \cup B$ are both the same size, meaning the similarity is 1. Now suppose that A contains all the elements of B, *plus some additional elements that aren't in* B. Now, $|A \cap B|$ is smaller than $|A \cup B|$, so the similarity is less than 1.

It is possible to directly compute the Jaccard index for two sets by finding the intersection and union. However, it is computationally budensome to calculate the similarity for very large sets.

2 Revew: MinHash

MinHash is a hash function $h(\cdot)$ defined on sets A and B such that the collision probability of A and B is equal to $J(A, B)$. By finding many such MinHash values and counting the number of collisions, we can efficiently estimate $J(A, B)$ without explicitly computing the similarities.

To compute a MinHash signature of a set $A = \{a_1, a_2, ...\}$, generate a universal hash function U and compute the set of signatures $U(A) = \{U(a_1), U(a_2), ...\}$. The MinHash signature (or value) is $\min U(A)$.

3 Parity of MinHash

MinHash outputs an integer value, but it turns out that the (1-bit) parity of a MinHash signature is also locality-sensitive. This has applications where we want to minimize the space needed to store a set of MinHash values and speed up computation.

We define a 1-bit *parity MinHash* L as 1 if MinHash(A) is odd and 0 if MinHash(A) is even.

$$L(A) = \begin{cases} 1 & \text{if MinHash}(A) \text{ is odd} \\ 0 & \text{else} \end{cases}$$

Without proof, we state the following fact about the collision probability of $L(\cdot)$.

$$\Pr(L(A) = L(B)) = J(A,B) + \frac{1 - J(A,B)}{2} = \frac{1}{2}(1 + J(A,B))$$

This is sufficient to estimate the similarity.

3.1 Example

Suppose we have 50 MinHash parity values (i.e. $L(\cdot)$ values) and we want to estimate $J(A, B)$. How can we do this?

We can store 50 $L(\cdot)$ values using < 7 bytes. To store $L(A)$, we store the 50 binary hash values in a 50-bit signature. We will refer to this signature as $S(A)$. To estimate $\frac{1}{2}(1+J(A,B))$, we compute $S(B)$ and count the number of 1s in $\text{XNOR}(S(A), S(B))$).

If $J = 0.8$, then with 50 parity values the error is slightly greater than 0.05. (i.e. our estimate $\widehat{J(A,B)} - J(A,B) < 0.05$ with high probability). To compute the error, note that our estimate of $\frac{1}{2}(1 + J(A,B))$ has a Binomial distribution with $N = 50$ and $p = \frac{1}{2}(1 + J(A,B))$.

4 LSH for Similarity Search

We can use LSH functions to make similarity search faster for certain inputs. To do this, we create a hash table using an LSH function $h(\cdot)$. We insert each element into the corresponding bucket. To perform a similarity search for query q, we compute $h(q)$ and search for near neighbors only in bucket $h(q)$.

By constructing K independent LSH functions $h_1(\cdot), ...h_K(\cdot)$, we can reduce the quantity and improve the similarity of items that hash to the same bucket as $h(q)$. In Figure 1, we show this process for $K = 2$. We compute $h_1(q) = 01$ and $h_2(q) = 11$ and consider only the near-neighbor candidates in bucket $01, 11$. The intuition is that searching in this bucket is better than randomly searching the dataset. It is possible to prove that with many hash functions (large K) and - as we will see in a minute - many hash tables, the results are much better than random.

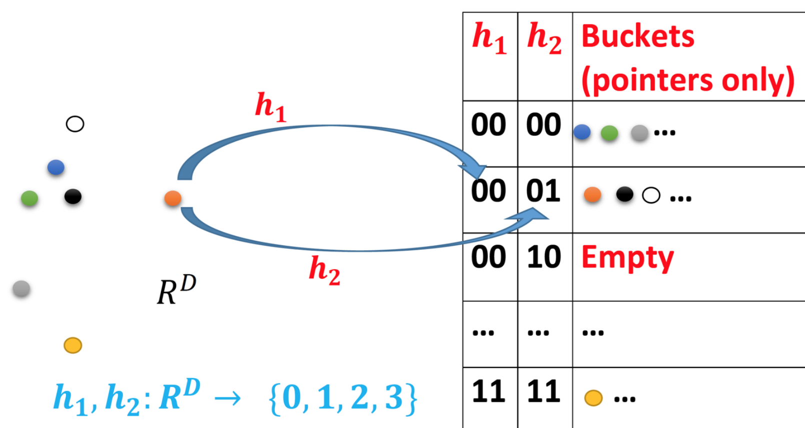

Figure 1: Similarity search for query q with $K = 2$ LSH functions. To search for near-neighbors, we only need to examine elements inside the second bucket. In practice, we do not store the elements in the bucket - just a pointer to the data

The problem with the approach in Figure 1 is that the candidate bucket may be empty. Furthermore, suppose that q has a very good near neighbor x with high $\text{sim}(x, q)$. It is likely that x is in the candidate bucket, but it is possible that we were very unlucky and had $h(q) \neq h(x)$ for one of the K hash functions.

To fix this, we can repeat the hash table L times, then search through the union of candidate sets (Figure 2).

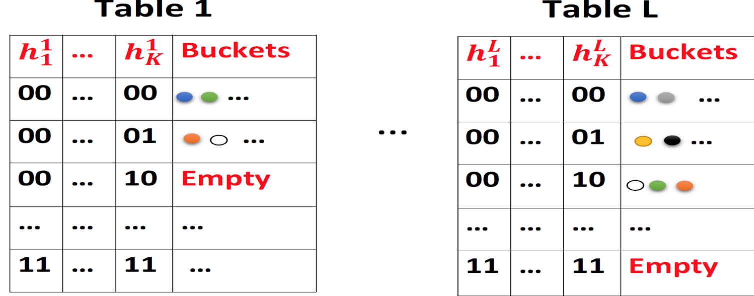

Figure 2: Near neigbor search with L independent hash tables

During the preprocessing stage, we insert all elements of $\mathcal{D}$ into L different hash tables. One practical issue is that we cannot store an infinite number of buckets. Therefore, we use a universal hash function to take the K hash signatures $v_1, v_2, ...v_k$ (where $v_i = h_i(x)$ and turn them into a single bucket index $b \in [1, R]$. This may be done, for example, by using the following hash function:

$$b = u(v_1, v_2, v_3, ...) = a_1 v_1 + a_2 v_2 + a_3 v_3 + ... + a_k v_k + c$$

where the set of $\{a_i\}$s and c are chosen randomly. In practice, this results in very few undesirable random collisions and is not a problem.

Algorithm 1: LSH Similarity Search Algorithm: Preprocess

Input: Dataset $\mathcal{D}$ of size n

Output: A set $\mathcal{S}$ of L hash tables

1 Initialize: $K \times L$ LSH functions
2 **for** $x_i \in \mathcal{D}$ **do**
3   Compute $K \times L$ hash values $h_{k,l}(x_i)$
4   **for** *Hash table* $T_j \in \mathcal{S}$, *where* $j = 0$ *to* $j = L$ **do**
5   Map L LSH values $h_{1,j}, h_{2,j}, ...h_{K,j} \rightarrow$ a bucket index b using a universal hashing
6   Insert x_i into bucket $T_j(b)$
7 **return** $\mathcal{S}$

The preprocessing complexity is $O(LK)$ for each element, or $O(nLK)$ for the whole dataset. K controls the quality of each bucket and L controls the failure probability of the near-neighbor search.

During querying, we find the buckets q would have hashed to, and we search for near neighbors in those buckets.

Algorithm 2: LSH Similarity Search Algorithm: Query

Input: A query q, a set $\mathcal{S}$ of L hash tables, $K \times L$ LSH functions
Output: An approximate near neighbor x
1 Compute $K \times L$ query hash values $h_{k,l}(q)$
2 Initialize candidate set $\mathcal{C} = \emptyset$
3 **for** *Hash table* $T_j \in \mathcal{S}$*, where* $j = 0$ *to* $j = L$ **do**
4   Map L LSH values $h_{1,j}, h_{2,j}, ...h_{K,j} \rightarrow$ a bucket index b using a universal hashing
5     $\mathcal{C} = \mathcal{C} \cup T_j(b)$
6 $x = \arg\max_{x_i \in \mathcal{C}} \text{sim}(q, x_i)$
7 **return** x

4.1 Analysis

Here, we provide a partial analysis of the algorithm presented in the previous section. This analysis is sufficient for practical insight on how to use the algorithm.

Suppose that we use MinHash as our LSH function $h(\cdot)$. Then the collision probability of x and q is

$$p(x, q) = \Pr(\text{MinHash}(x) = \text{MinHash}(q)) = J(x, q)$$

We want to find the probability that an element is considered as a near-neighbor candidate. The probability that x is in the same bucket as q in a hash table is $p(x, q)^K = J(x, q)^K$, since we require that all K MinHash signatures be the same.

The probability that x is not in the same bucket as q is $1 - p(x, q)^K$. We use L independent hash table repetitions, so the probability that x is not in the same bucket as q for *any* of the tables is $(1 - p(x, q)^K)^L$. Therefore, for our example

$$\Pr(x \text{ not retrieved}) = (1 - J(x, q)^K)^L$$

Likewise, the probability that x is considered as a near-neighbor candidate is

$$\Pr(x \text{ is retrieved}) = 1 - (1 - J(x, q)^K)^L$$

Note that the probability that x is retrieved is a monotonic function of $p(x, q)$. This is desirable, since it means the algorithm is better than random. We are more likely to search through similar data than through dissimilar data.

4.1.1 Query Complexity

In high dimensions, the query time complexity of non-LSH methods quickly degrades to be no better than linear search. Linear search through the dataset is $O(n)$. We have just seen that our guarantees are likely to be in terms of $p(x, q)$ - that is, the similarity between x and q. So the question remains: What is the query time complexity?

LSH-based search is $O(n^\rho)$, where ρ depends on the similarity threshold and gap in the near-neighbor search problem. If we have a good query - one for which there are high similarity elements in the dataset, then $\rho << 1$. If we have bad queries that do not satisfy the threshold and gap conditions, we can reject the query or ignore the match.

If we want very high similarity neighbors ($J(x, q) > 0.8$ or so), this scheme is very time-efficient. As $J(x, q) \rightarrow 1$, the runtime complexity approaches $O(1)$.

4.2 Practical Notes

In the previous section, we found that the probability of retrieval for an element in the dataset is dependent on its similarity with the query. For example, suppose $K = 5, L = 10$. Then if

$$p(x, q) > 0.8 \rightarrow \Pr(x \text{ is retrieved}) > 0.98$$

$$p(x, q) < 0.5 \rightarrow \Pr(x \text{ is retrieved}) < 0.2$$

For applications we need to know whether the most common value of $p(x, q)$ is large or not. If $p(x, q)$ is a large string and we compute the 3-grams as described previously, the average $p(x, q)$ is very close to 0. Therefore, this near neighbor search method will work well for our application.

We also saw that the runtime, space complexity, and performance are dependent on L and K. In practice, the best way to determine good values of L and K are through practice and experience. The following rule of thumb provides a good starting point:

$$K \approx \log n \quad L \approx n$$

where n is the size of the dataset. For good intuition when turning parameters, remember that an increase in K exponentially decreases the number of candidates retrieved and that an increase in L linearly increases the number of candidates retrieved.