COMP 480/580 — Probabilistic Algorithms and Data Structure Oct 5, 2023

Lecture 14

Lecturer: Anshumali Shrivastava *Scribe By: {pc45, eh58, gw23}@rice.edu*

1 Overview

Search engines with users input queries remains challenges for deployments. One significant bottleneck is the real-time correction of user-typed queries. Imagine a scenario where a database, denoted as D, contains approximately 50 million statistically significant query strings that have been observed in the past. The goal is that when a user types a new query q, the system must sift through the database to identify the closest string, based on a certain distance metric, to q and promptly return the pertinent results. However, the sheer volume of the database means that approximately 50 million distance computations are required for every single user query. Even with a cheap distance function, processing such a volume would require about 400 seconds—equivalent to 7 minutes on a standard CPU. If we were to use more complex algorithms like the edit distance, this duration could extend to hours, rendering the process impractical for real-time applications. Not mention that the acceptable latency limit is a mere 20 milliseconds.

Can we do better? Through advanced approximation methods, it is not only feasible but entirely possible to correct user-typed queries in as little as 2 milliseconds—a staggering 210,000 times faster than conventional methods!

1.1 Jaccard Similarity

The Jaccard Similarity is a measure used to quantify the similarity between two sets, which is particularly useful for comparing sets of items. The Jaccard Similarity between sets A and B is calculated as [2]:

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

Where:

- A and B are two sets.
- $|A \cap B|$ represents the size of the intersection of sets A and B (i.e., the number of elements common to both sets).
- $|A \cup B|$ represents the size of the union of sets A and B (i.e., the number of elements without duplicates from either set).

The value of the Jaccard Similarity ranges from 0 to 1:

- A value of 1 indicates the two sets are identical.
- A value of 0 indicates the two sets have no elements in common.

As an example, consider the sets:

$$A = \{1, 2, 3, 4\}$$

$$B = \{3, 4, 5, 6\}$$

The intersection of A and B is:

$$A \cap B = \{3, 4\}$$

The union of A and B is:

$$A \cup B = \{1, 2, 3, 4, 5, 6\}$$

Thus, the Jaccard Similarity between A and B is:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{2}{6} = \frac{1}{3}$$

1.2 Locality Sensitive Hashing

LSH is a technique designed to identify items that are close (or similar) to one another in a high-dimensional space[3]. Unlike classical hashing, where the aim is to avoid collisions, LSH aims for controlled collisions to capture similarity.

Classical Hashing

In the realm of classical hashing:

- For items x and y, if $x = y$ then it is always true that $h(x) = h(y)$.
- Conversely, if $x \neq y$ then, ideally, $h(x) \neq h(y)$.

Locality Sensitive Hashing (Randomized Relaxation)

LSH introduces a nuanced, probabilistic approach to hashing:

- If the similarity between items x and y (denoted as $\text{sim}(x, y)$) is high, then the probability that $h(x) = h(y)$ is also high.
- On the flip side, if $\text{sim}(x, y)$ is low, then the probability that $h(x) = h(y)$ is correspondingly low.
- Notably, if $h(x) = h(y)$, this implies that $\text{sim}(x, y)$ is high, but only in a probabilistic sense.

1.3 Minwise Hashing

How is similarity search connected to Minwise Hashing and Jaccard Similarity?

Statement:

$$\Pr\left(\text{Minhash}\left(S_1\right) = \text{Minhash}\left(S_2\right)\right) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = J$$

Justification: Consider a hash function, U, used to determine the minhash of sets S_1 and S_2. Let's denote e as the element possessing the minimum hash value when considering the union $S_1 \cup S_2$. We can observe that if e belongs to S_1, then its minhash value corresponds to $U(e)$. Similarly, if e is in S_2, its minhash value will be $U(e)$. There's also a scenario where e is a member of the intersection $S_1 \cap S_2$, making the minhash values of both sets equal to $U(e)$. If the size of the intersection is N and the union is M, among the M potential choices for e, N can lead to a match. Thus, the collision likelihood is $\frac{N}{M} = J$.

2 Minwise Hashing

Minwise hashing, commonly referred to as Minhash, is a technique primarily designed for efficiently estimating the similarity between sets, such as documents. This method has been particularly popular for estimating the Jaccard similarity between large sets[1].

1. **Representation of Documents as Shingles**:

   Documents can be represented as a set of shingles (or substrings). For instance, the document "Amazon" might be shingled into overlapping substrings of length three: $S = \{\text{ama, maz, azo, zon, on.}\}$.

2. **Hashing with Random Seeds**:

   To apply minwise hashing, one starts by using a hash function that can accept a random seed to generate different hash values for the same input. An example of such a hash function is `Murmurhash3`.

   When you use a new random seed i with a hashing algorithm, say `Murmurhash3`, it will produce a distinct hash value for the same input. For our shingles in S, it might produce values such as $\{153, 283, 505, 128, 292\}$.

3. **Finding the Minhash**:

   The minhash value for a particular seed is simply the smallest hash value generated for the shingles in the document. Using our previous example, the minhash value would be 128, as it's the smallest hash value in the set $\{153, 283, 505, 128, 292\}$.

4. **Multiple Minhash Values**:

   By changing the seed for the hash function, one can obtain a new set of hash values for the shingles, and consequently, a new minhash value. Multiple minhash values can be generated by repeating this process with various seeds, which allows for the creation of a minhash signature representing the document. The similarity between two documents can then be estimated by comparing the fraction of minhash values that match between their signatures.

In summary, Minhash provides a mechanism for capturing the essence of documents in a compact form, making it feasible to compare large numbers of documents efficiently. The approach leverages the randomness of hash functions and the idea of using the minimum hash value as a representative signature.

2.1 Property of Minwise Hashing

1. **General Applicability**:

   Minwise hashing can be applied to any set.

2. **Range of Minhash**:

   Minhash generates values in the range [0-R], where R is sufficiently large to accommodate the expected variations.

3. **Randomness and Hash Functions**:

   For any set, the properties of Minwise Hashing are provable under the assumption that the hash function, denoted as U, is random.

4. **Random Sample Property**:

   Fact1: Given the randomness of the hash function U, for any set, the element with the minimum hash value can be treated as a random sample from the set.

5. **Random Element and Hash Function**:

   Consider a set S, and let's sample a random element e using the hash function U.

   Claim1: There seems to be a condition or property regarding e and the set S that is implied by its Minhash value, but the statement is incomplete in the provided content (denoted as "if and only if").

The presented properties showcase the versatility and robustness of Minwise Hashing. Particularly, the reliance on the randomness of the hash function U is instrumental in several of its key features.

2.2 Parity of MinHash

MinHash's properties extend beyond its ability to estimate set similarities. One such property is its parity. The parity of a number refers to whether it's even (parity = 0) or odd (parity = 1). When applied to MinHash, this concept provides another layer of information compression.

2.2.1 Parity in Action

1. **Document Representation**:

   A document can be represented as a set of shingles. For instance: $S = \{\text{ama, maz, azo, zon, on.}\}$.

2. **Hashing with Random Seeds**:

   Using a hash function, such as `Murmurhash3`, with a random seed i, we can produce a set of hash values for the shingles in S. For example: $\{153, 283, 505, 128, 292\}$.

3. **Computing MinHash**:

   The MinHash value is the smallest hash value from the set. From the previous example, Minhash $= \min\{153, 283, 505, 128, 292\} = 128$.

4. **Parity of MinHash**:

   The parity of 128 is 0 (since it's even), which provides us with 1-bit of information about the document.

2.2.2 Parity of MinHash for Compression

The parity of MinHash serves as an effective compression mechanism, especially for similarity estimation.

- Given 50 parities of MinHash values, one might ask: how can we estimate the Jaccard similarity J?
- The memory requirement for storing these parities is minimal, only requiring 50 bits or less than 7 bytes (equivalent to 2 integers).
- For $J = 0.8$, the error is slightly worse than 0.05 (though the computation method wasn't provided).

- Interestingly, the storage requirement only depends on similarity and not the actual size of the set. Whether a set has 100, 1,000, or 10,000 elements, the storage cost for similarity estimation remains constant.

This highlights a unique tradeoff presented by the parity of MinHash: despite the varying size of the datasets, the storage requirement remains fixed for similarity computations.

3 Locality Sensitive Hashing

Minwise Hashing is locality sensitive as we dicussed earlier.

3.1 Classical Hashing

In traditional hashing:

- If $x = y$ then $h(x) = h(y)$.
- If $x \neq y$ then, with high probability, $h(x) \neq h(y)$.

3.2 Locality Sensitive Hashing (Randomized Relaxation)

In contrast, LSH focuses on similarity:

- If the similarity $\text{sim}(x, y)$ is high, the probability that $h(x) = h(y)$ is also high.
- If the similarity $\text{sim}(x, y)$ is low, the probability that $h(x) = h(y)$ is correspondingly low.

This type of hashing is especially beneficial when trying to identify approximate matches or when working with large datasets where exact matches are rare or less meaningful.

3.3 Jaccard Similarity and LSH

We'll explore how to design hash functions, h, that are tailored for the Jaccard distance. The fundamental intuition is:

- If $h(x) = h(y)$, then the Jaccard similarity between x and y is likely to be high, though this is probabilistic.

The Jaccard similarity is a measure of the similarity between two sets and is a popular metric used in conjunction with LSH.

3.4 Search Engine using LSH

Search engines must handle vast amounts of data and provide results quickly. Locality Sensitive Hashing (LSH) offers an efficient way to perform approximate similarity search, making it valuable for search engines. Instead of comparing a query to every single document in the database, LSH helps in narrowing down the search to a smaller subset that's likely to contain the relevant results.

3.4.1 Multiple Independent Hash Tables

- The LSH approach creates multiple hash tables, each with its own set of hash functions.
- Each table returns a bucket of candidate documents that are potentially similar to the query.
- With multiple tables, there is an increased chance of finding truly similar items, even if they are missed in one table due to the probabilistic nature of LSH.

3.4.2 LSH Algorithm for Search Engines

Algorithm 1 LSH Preprocessing

```
Input: Dataset $D$, Number of hash tables $L$, Number of hash functions $K$
Output: Hash tables $T_1, T_2, \ldots, T_L$
1: Initialize $L$ hash tables, $T_1, T_2, \ldots, T_L$.
2: for each data point $P$ in $D$ do
3:     for each hash table $T_i$ from 1 to $L$ do
4:         Compute compound hash key, $k$, for $P$ using $K$ hash functions: $h_1, h_2, \ldots, h_K$
5:         Insert $P$ into $T_i$ using key $k$.
6:     end for
7: end for
8: return $T_1, T_2, \ldots, T_L$
```

Algorithm 2 LSH Query

```
Input: Query point $q$, Hash tables $T_1, T_2, \ldots, T_L$, Number of hash functions $K$
Output: Set of candidate points $C$ that are likely to be similar to $q$
1: Initialize candidate set $C$ to empty.
2: for each hash table $T_i$ from 1 to $L$ do
3:     Compute compound hash key, $k$, for $q$ using $K$ hash functions: $h_1, h_2, \ldots, h_K$
4:     Retrieve all points in $T_i$ with key $k$ and add to candidate set $C$.
5: end for
6: return candidate set $C$.
```

- **Preprocess Database D:**
  This algorithm is designed to preprocess a given dataset D. D using the principles of Locality Sensitive Hashing (LSH). For each data point in the dataset, the algorithm computes a key using a set of hash functions and then inserts this data point into multiple hash tables. This preprocessing is essential for efficient similarity searches, as items that are similar (under some similarity metric) are likely to be hashed into the same bucket in one or more of these tables.

- **Query with q:**
  After preprocessing the dataset, this algorithm performs a query to retrieve items from the dataset that are similar to a given query point q. The algorithm computes the hash key for the query point and searches for this key in each of the hash tables. All items with a matching key are considered as potential candidates for being similar to the query.

3.4.3 Advantages in Search Engines

- **Speed**: LSH reduces the number of comparisons needed, resulting in faster search results.
- **Scalability**: LSH is designed to handle vast datasets, making it suitable for large-scale search engines.
- **Flexibility**: Different LSH functions can be tailored for various similarity measures, catering to diverse data types and requirements.

Locality Sensitive Hashing provides a robust framework for efficient similarity searches in vast datasets, making it invaluable for search engines that must provide rapid and relevant results to user queries.

References

[1] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th Very Large Database (VLDB) Conference*, 1999.

[2] Jure Leskovec, Anand Rajaraman, and Jeffrey Ullman. *Mining of Massive Datasets*. Cambridge, 2020. Earlier version available at http://infolab.stanford.edu/ ullman/mmds/ch3.pdf, p. 76-77.

[3] T. T. Tanimoto. An elementary mathematical theory of classification and prediction. Internal technical report, IBM, 1958. Report from 1957, Published on 17 Nov 1958.