COMP 480/580 — Probabilistic Algorithms and Data Structure Oct 10, 2024

Lecture 13

Lecturer: Anshumali S. Scribe By: Dongwei L.(dl82), Lingxi Z.(lz88), Rex L.(rl105), Thanasis H.(th81)

1 Motivation

1.1 Example Problem: Large-scale Search

What's Large-scale search problem?
1. First, we have a query image.
2. Second, we want to search a giant database(internet) to find similar images.
3. Third, we want the search is fast and (mostly) accurate.

What makes Large-scale search challenging?
1. **Memory constraints:** Large datasets, such as billions of images, cannot fit into the main memory of a single machine. Storing data on disk is too slow for real-time search operations, and memory limitations make it difficult to load all data at once.
2. **Huge database size:** Platforms like Facebook handle massive datasets. Managing and searching through such a vast number of data points is challenging.
3. **High-dimensional data:** Images and other multimedia content are often represented as high-dimensional vectors. High-dimensional data is difficult to search efficiently, both due to its dimensionality (which causes distances between points to become less meaningful) and the computational cost of performing nearest-neighbor search in high-dimensional spaces.

1.2 Solution: Hash Algorithm

Exact searches can theoretically be executed in constant time $O(1)$ using an ideal hash function. However, in reality, when searching for images across the internet, the metadata of an image can vary from site to site, even if the image itself remains identical. This poses a challenge for hash functions, which are highly sensitive to small changes—any slight modification in the input can produce a completely different hash value.
Moreover, it's impossible to design a deterministic hash function that generates the same hash values for near-duplicates while ensuring distinct values for non-duplicates.
Instead, we can consider probabilistic hash functions, which are more forgiving when dealing with near-duplicates. These functions generate hash values that are highly likely, though not guaranteed, to be the same for near-duplicate inputs. For instance, universal hash functions can be designed to behave probabilistically, increasing the likelihood of similar hash values for similar inputs.

1.2.1 Near Neighbour Search

Given a (relatively fixed) collection C and a similarity (or distance) metric *sim*. For any query q, compute:

$$x^* = \arg\min_{\{x \in C\}} \text{sim}(q, x)$$

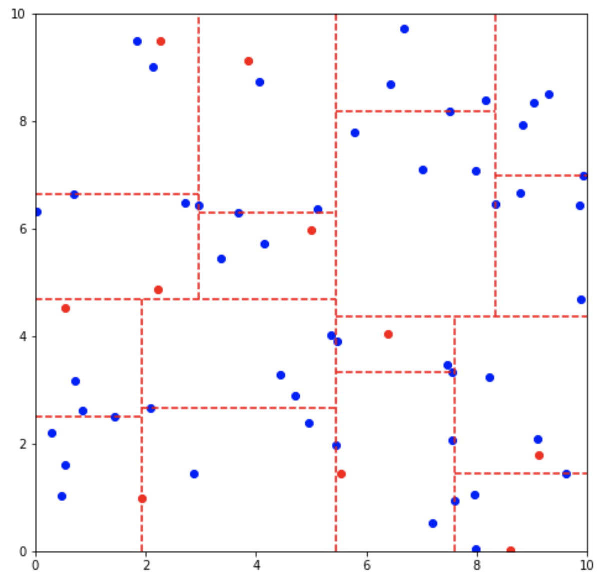

Figure 1: Example of space partition method

Settings:
1. $O(nD)$ per query, where n is size of C and D is dimensions.
2. Querying is a very frequent operations.
3. n and D are large.

1.2.2 Space Partitioning Method

1. **What is space partitioning?**
Divide a space into non-overlapping regions to manage and query spatial data. Partition the space and organize database into trees (Figure 1).
2. **However,** in high dimensions, space partitioning is not at all efficient. Even $D > 10$, leads to near exhaustive.

Motivating Problem: Search Engines
Task: Correcting a user typed query in real-time, for example, we mistakenly typed "eraser-pen" instead of "eraser pen".
One solution: take a database D of statistically significant query strings observed in the past. (around 50 million). Given a new user typed query q, find the closest string $s \in D$ (in some distance) to q and return associated results.
Latency: 50 million distance computation per query. A cheap distance function takes $400s$ or $7min$ on a reasonable CPU. If you used edit distance, it will be hours.
Latency limit is roughly $< 20ms$.

Can we do better?
1. Exact solution: No.
2. Approximation: Yes, we can do it much faster!

2 Locality Sensitive Hashing

2.1 Definition

Locality Sensitive Hashing (LSH) is a technique used to efficiently find similar items in large datasets. The key idea is to hash items in such a way that similar items are more likely to end up in the same bucket. This allows us to quickly identify similar items without comparing every pair directly.

Formally, a hash function h is considered **locality sensitive** if it satisfies the following property for a given similarity measure:

- For two items x and y: - If x and y are similar, then $h(x) = h(y)$ with high probability. - If x and y are not similar, then $h(x) = h(y)$ with low probability.

By using such hash functions, we can group similar items together and perform approximate nearest neighbor searches efficiently.

2.2 Notion of Similarity: Jaccard

Before we can apply LSH, we need a way to measure the similarity between items. One common measure for sets is the **Jaccard Similarity**.

The **Jaccard Similarity** between two sets A and B is defined as:

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

This measures how similar two sets are by comparing the size of their intersection to the size of their union.

Example: Let's consider two strings and convert them into sets of 3-grams (all contiguous sequences of 3 characters). For instance:

- "amazon" $\rightarrow \{\text{ama}, \text{maz}, \text{azo}, \text{zon}\}$ - "amazing" $\rightarrow \{\text{ama}, \text{maz}, \text{azi}, \text{zin}, \text{ing}\}$

Now, we compute the Jaccard Similarity between these two sets:

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

This calculation shows that "amazon" and "amazing" share two 3-grams out of a total of seven unique 3-grams between them, resulting in a Jaccard Similarity of $\frac{2}{7}$.

2.3 Random Sampling Using Universal Hashing

Universal hashing involves selecting a hash function at random from a family of hash functions with certain mathematical properties. This randomness helps ensure that the hash function behaves well for any input data.

In the context of LSH:

- We use universal hashing to randomly sample elements from sets. - By applying the same hash function to different sets, we ensure consistent sampling. - This random sampling reduces the amount of data we need to process while preserving the ability to estimate similarity.

Procedure: 1. Choose a universal hash function h. 2. Define a threshold t. 3. For each set S, select elements where $h(e) \leq t$. 4. Compare the sampled subsets to estimate similarity.

Benefits: - Reduces computational overhead by considering only a subset of elements. - Maintains a probabilistic guarantee on the estimation accuracy. - Scales well to large datasets.

2.3.1 Example of Random Sampling

Suppose we have two sets:

- $S_1 = \{\text{ama}, \text{maz}, \text{azo}, \text{zon}\}$ - $S_2 = \{\text{ama}, \text{maz}, \text{azi}, \text{zin}, \text{ing}\}$

We apply a universal hash function h to each element and select those with hash values below a certain threshold.

- For S_1, sampled elements might be $\{\text{ama}\}$. - For S_2, sampled elements might be $\{\text{ama}\}$.

By comparing the sampled subsets, we can estimate the Jaccard Similarity between S_1 and S_2 without processing all elements.

2.3.2 Advantages over Full Comparison

- **Efficiency**: Reduces the amount of data to process. - **Scalability**: Suitable for large datasets where full comparisons are impractical. - **Simplicity**: Easier to implement and parallelize.

2.3.3 Application in LSH

Using random sampling with universal hashing in LSH allows us to:

- Hash similar items into the same buckets with high probability. - Quickly retrieve candidate items for similarity comparison. - Perform approximate nearest neighbor searches efficiently.

By combining these techniques, we can build systems that handle large-scale similarity search tasks effectively.

3 Minwise Hashing

3.1 Definition

The Minhash method, invented by Andrei Broder, is commonly applied to large-scale clustering problems, such as grouping documents based on the similarity of their word sets. For instance, given a document with a string "amazon." we can split it into a set of substrings,

$$S = \{\text{“}ama\text{”}, \text{“}maz\text{”}, \text{“}azo\text{”}, \text{“}zon\text{”}, \text{“}on.\text{”}\}$$

Using a random hash function, introduced in the last section, $U_i : Strings \to N$, which maps a string to a random number. To be noted, we can generate this function easily with MurmurHash3 by setting a unique random seed i. By hashing each substring in the set S, we obtain a set of numbers

$$U_i(S) = \{U_i(\text{“}ama\text{”}), U_i(\text{“}maz\text{”}), U_i(\text{“}azo\text{”}), U_i(\text{“}zon\text{”}), U_i(\text{“}on.\text{”})\}$$

and assume that the value of the hash function be

$$U_i(S) = \{153, 283, 505, 128, 292\}$$

The Minhash value for document S is the minimum value from the set $U_i(S)$, which, in this example, would be 128. Importantly, we can generate a new Minhash function by simply choosing a different seed for the random hash function.

3.2 Properties

Minhash can be applied to any set, as it maps a set to a value within the range [0, R], where R can be large enough. For any two sets S_1 and S_2, we have

$$Pr(Minhash(S_1) = Minhash(S_2)) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|},$$

which indicates that the probability of a hash collision between two sets is exactly equal to their Jaccard similarity. In other words, if two sets are very similar and have a high collision rate, corresponding to a high Jaccard similarity, the probability of their Minhash values being the same is also high.
Proof. Under randomness of hash function U, for any set, the min() process acts as a random permutation, meaning each substring has an equal probability of having the minimum hash value. Now, consider that e has the minimum hash value in the set $\{S_1 \cup S_2\}$. We assume that $e \in S_1$ (the proof will be similar if $e \in S_2$), then the Minhash values for S1 and S2 will be the same if and only if $e \in S_2$, that is $e \in \{S_1 \cap S_2\}$. Therefore, we have

$$Pr(Minhash(S_1) = Minhash(S_2)) = Pr(e \in \{S_1 \cap S_2\})$$

Since the permutation is random, we have

$$Pr(e \in \{S_1 \cap S_2\}) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

The property is proven,

$$Pr(Minhash\,(S_1) = Minhash\,(S_2)) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

3.3 Estimate Similarity Efficiently

3.3.1 Estimating Jaccard Similarity with Minhash

Given 50 minhash functions of sets S_1 and S_2, we can estimate the Jaccard similarity J as follows:

$$\hat{J} = \frac{\#\{i : h_i(S_1) = h_i(S_2)\}}{50}$$

where h_i represents the i-th minhash function. The fraction of minhash functions for which the hashes are equal gives an estimate of the Jaccard similarity between the two sets.

3.3.2 Variance of the Estimate

The variance of the estimate is given by:

$$\text{Var}(\hat{J}) = \frac{J(1-J)}{50}$$

For example, when $J = 0.8$, the variance is approximately:

$$\text{Var} = \frac{0.8 \times (1 - 0.8)}{50} = 0.05$$

Thus, the variance is quite small, indicating a reliable estimate.

3.4 Parity of MinHash

3.4.1 Using Parity of Minhash for Estimation

We can also use the parity of minhash values to estimate the similarity between two sets. Specifically, the probability that the parities of the minhashes are equal is:

$$P(\text{parity}(h(S_1)) = \text{parity}(h(S_2))) = J + (1 - J) \times 0.5$$

This method gives an alternative way to estimate the similarity, though with different probabilistic behavior compared to direct minhash comparison.