Top-k Set Search using MinHash Locality Sensitive Hashing:

Given a set Q (the query) and a collection of L={C_1, …, C_n}, find a sub-collection w of size k distinct sets such that:

Jaccard(Q,C) > 0, C ∈ L and

Jaccard(Q,C), C ∈ w Jaccard(Q,C)>= Jaccard(Q,X), X ∉ w, X ∈ L

Note that ties are broken arbitrarily so that only k sets exist in the result.

The brute-force algorithm computes the Jaccard similarity of Q with every set in L, sorts the sets based on similarity, and returns k sets with highest similarity values. This algorithm returns the exact results. As we saw in the lecture, MinHash LSH provides a space- and time-efficient technique for threshold-based set similarity search based on Jaccard. In this assignment you are to implement an algorithm (in your programming language of choice) that adopts MinHash LSH for top-k search. Recall MinHash LSH for threshold-based search returns approximate results. That is, there will be False Positives (FPs) and False Negatives (FNs). The FNs impact the precision of the algorithm, since some correct results are not returned by LSH as candidates and will not be in the final result set. The FPs impact the efficiency of the algorithm, since the algorithm needs to verify the exact Jaccard similarity of candidate sets which counts as the post-processing cost. Your algorithm for top-k search will likely have such considerations. In this assignment, we explore the trade-off between precision and efficiency for top-k search using MinHash LSH.

Data Set:

You may find the curated set collection here: TBA

set in this collection corresponds to a column in the crawled data, from web and open data, and is created by extracting unique values from the column.

We also provide you with a set of queries to experiment with. The query set is divided into five groups. Sets in each group have cardinalities within a pre-determined interval. For example, all sets in the first group have cardinalities in [10,250) and the sets in the second group have cardinalities in [250,500), and so on. The query set can be found here: TBA

Comparison to Baseline:

Compare your algorithm with the brute-force algorithm in terms of precision and efficiency.

Precision evaluation: Plot the average precision of your algorithm for each query interval. We use the standard definition of precision based on TPs (True Positives), TNs (True Negatives), FPs (False Positives), FNs (False Negatives).

$$Precision = \frac{TP+TN}{TP+TN+FP+FN}$$

Report your observations.

Efficiency evaluation: Plot the average query time of your algorithm as well as the baseline for each query interval.

We suggest you follow these steps

Sketching

- Create n=128 hash functions that will be used for generating minhash values
- Implement a function that takes as input a set and returns the minhash signature of the set using the pre-defined hash function. Note you should use the same set of hash functions for all sets.
    - Save an array of the minhash signature of all sets on disk. Alternatively, you could use in-memory data structures, however, storing on disk saves you some time every time you need to test your code.

Indexing

- Choose reasonable values for parameters b and r. For example, b=8 and r=16.
- Create another set of hash functions for building an LSH index. You can implement an LSH index as r hash maps.
- Hash bands of each signature into buckets. Each signature is inserted into a bucket with its ID.

Querying

- Create the signature of a query set using the signature creation function implemented above.
- Hash bands of the query signature into LSH buckets and retrieve the sets that are in the hit buckets. These are candidates and include false positives.
- Evaluate the Jaccard of candidate sets and the query using their signatures.
- This information can be used for evaluating the precision of LSH on this specific data set with the chosen parameters.