Coding Project 1 Explanation

27 February 2025

1 Methodology

To find the q most similar pairs of numbers, I use a hybrid approach where I first use locality sensitive hashing to shrink the set of potential document pairs by eliminating most document pairs, and then I compute the true Jaccard similarity of the remaining pairs and return the q pairs with highest similarity.

In more detail, we start with $n = 28592$ documents corresponding to

$$\frac{n(n-1)}{2} \approx 400{,}000{,}000$$

pairs. Then, I repeatedly apply a locality sensitive hash and remove the pairs that do not hash to the same value. After 3 iterations, I am left with a set of 165,500 pairs, with 99.96% of pairs being eliminated. This step takes around a minute.

Then I manually compute the Jaccard similarity of the remaining 165,500 pairs. This involves computing the intersection of the set of k-shingles with approximately linear time complexity in the number of shingles, which is on average about 10000. This step takes about 100s.

Sorting the 165,500 pairs by descending similarity, I return the first $q = 1000$ pairs which have the highest similarity. This step takes about 0.1s.

2 Code

Functions:

1. get_shingles: String, Int $\to$ List[Int].

   Given a line of text and an int k, returns a list of the k-shingles hashed to ints. Compared to using a dictionary from shingles to ints, this allows the function to be run concurrently on multiple threads without shared objects. Note that there are on the order of 10^7 unique shingles. Since I am on 64-bit architecture the expected number of collisions is around $\frac{(10^7)^2/2}{2^{64}} \ll 1$. On 32-bit architecture the expected number of collisions is on the order of 10,000.

2. read_documents: None → Int, List[List[Int]].

   Uses python's multiprocessing library to read in and compute the k-shingles of the data in the input file. Switching out the dictionary for a hash and using multiple threads lowered the runtime of this function from 300s to 20s.

3. hash_mask: Int, Int → Int.

   Hashes the first parameter and xors the result with the mask passed as the second parameter.

4. hash_n: Int → Int → Int.

   Returns a random hash function dependent on n based on pythons builtin hash function. I tried other fast hash functions such as the xxhash library and a universal hashing algorithm by Dietzfelbinger et al. as described by Wikipedia. They were all slower than applying a random mask to python's builtin hash function.

5. compute_lsh: List[Int], (Int → Int) → Int.

   Given a set of shingles and the hash function given by the second parameter, computes the maximum value among the hashes of the shingles. I tried to convert the lists to numpy vectors to speed up this computation but it is actually slightly slower.

6. next_sim_matrix: List[List[Int]] → Iterator[Array[Bool]]

   Given the list of documents, returns an iterator that gives random similarity matrices of the document, where $m[i][j]$ is true if document i and j hash to the same value.

7. analyze_true_similarity : List[List[Int]], Int, Int → Number

   Computes the exact Jaccard similarity of two documents by comparing their k-shingles.

8. find_most_similar : List[List[Int]], List[Int], List[Int] → List[Number]

   Finds q pairs of similar documents using the steps described in the methodology.

9. main

   Reads data from documents file and writes similar pairs to output file.

3 Additional Considerations

The program does not take much time to run. To increase the accuracy I could add additional or statements to catch more similar documents. Even just doing this a few times should greatly increase accuracy while also allowing the program to still run in < 15 minutes.