Big Data Analytics Programming: Assignment 3 Report

Fotios Kalioras (r1017513)

1 Important development choices

1.1 Hash Values and Signature Matrix construction and speedup

Per assignment requirements, a hash value matrix and a signature matrix of shapes $numValues * numHashes$ and $numHashes * numObjects$ respectively must be created. This approach has many unavoidable inefficiencies such as the fact that we need to compare the value of each hash function for every shingle. Due to the structure of the matrices, regardless of how we order the loops, one of the matrices will always be accessed column-wise, which is cache-unfriendly and leads to performance penalties.
To address this, after completing the initial implementation as required, a 1D approach was developed; For the 1D hash values matrix the different hash values for each unique value was stored sequentially. That means that the i_{th} value has it's j_{th} hash value on position $i * value + j$. This access pattern is optimised and cache friendly. Regarding the 1D signature matrix creation, access is again optimized by saving for every object o each hash value h on position $o * numHashes + h$. Same principle holds true for the LSH1D implementation, where the signature matrix is getting accessed with the pattern $doc * numHashes + row$, which ensures good spatiality. This access pattern would be inefficient if the signatures were large but in our case due to the memory and timing constraints, 2GB and 5 Minutes respectively, this approach seems the best.
Performance gains were tested on Alken departmental computer on a night with no traffic by running LSH on the full dataset. For 50 hash functions, 10 bands with 2234 buckets each the 2D version required 578 seconds for the signature matrix creation and 1750 seconds for the rest algorithm. Totaling ≈2200 seconds, 36 minutes, this performance is unacceptable. On the contrary, the 1D implementation needed 226 seconds and 510 seconds for the matrix creation and the rest of the LSH algorithm respectively. While 740 seconds, 12.5 minutes, still exceeds the assignments requirements, emphasize the superiority of the 1D implementation's performance, which is more than 3 times faster. It also means that since signature matrix cretion takes ≈200 seconds, there are ≈100 seconds left for the second part.

1.2 Hash buckets structure

Since each hash bucket can grow arbitrarily in size we are faced with the task of building a 2D array which is fixed on the first dimension but grows on the second, aka an Array of ArrayLists, a huge no-no in the Java community. This approach would suffer from the autoboxing overhead. We could alleviate this issue by allocating a static 2D array that is of size $numBuckets * numObjects$ but this approach was deemed to be inelegant. Instead a new "IntBucket" homemade Array structure was created. It is essentialy an int array that can grow in size if needed. It is initialized with small size (100) and every time it is exhausted it gets doubled. By creating an array of these IntBuckets the advantages of the Arrays and the flexibility of Lists are combined.

1.3 Early exit strategy

In the real life when you see Max Verstappen take the lead on the first turn of a GP you can close the livestream since you know who is going to win. Similarly this LSH implementation implements early stopping both when the number of matched signature hashes exceed the threshold and when the remaining hashes couldn't reach the threshold even if they all matched. With these checks, we exit the loop faster than Ferrari's pit crew can say 'We are checking' over team radio [1].

2 Experiment Setup

2.1 Verification of LSH implementation correctness

The correctness of all the LSH sub-components was tested by comparing the results of the brute force algorithm (bf) against the ones of the LSH algorithm (lsh) on the small subset of 5000 tweets. By using an arbitrarily large amount of hash functions (e.g. 420) we can approximate the bf results almost perfectly, although it should be noted that we still have some False Positives and False Negatives.

2.2 Selection of starting number of Hash functions

It goes without saying that when deployed on the full 5000000 tweet database and given the memory constraints (2GB), the hash functions must be reduced. It's been found empirically that using 50 hash functions, along with the necessary auxiliary arrays such as hash_values and signature_matrix almost top up the heap. Since we want to make the similarity approximation as good as possible, this value of hash functions is selected as a starting point, balancing both accuracy and runtime. However, as we have already established, this configuration leads to more than 10 minutes of runtime. For these reasons 20 hash functions were chosen as the maximum

2.3 Selection of starting number of buckets

Now that we have a number of hash functions the number of buckets must be determined. Fewer buckets contribute to an increase of recall, since the true-neighbor collision probability increases with bucket width w. However, the tradeoff is a decrease in precision; Excessive collisions increase the number of false positives [2]. Additionaly, a decrease in the number of buckets leads to more documents getting hashed to the same bucket and consequentially the brute-force signature check becomes more time consuming. Stack Overflow users suggest beginning with $\sqrt{numObjects}$ buckets, then adjusting this number and compare

the retrieved document distributions [3]. Therefore, the chosen number of hash buckets was determined to be $\sqrt{5000000} \approx 2236$, however this number proved to be quite low and lead to many unnecessary collisions. Consequentially the starting point was readjusted to 10000 hash buckets.

2.4 Selection of starting number of bands and rows

After fixing the number of buckets our attention shifts to the number of bands b and the number of rows within a band r, which have a inversely proportional relationship; For a given number of hash functions, if the number of bands increases the number of rows decreases, whereas a smaller b denotes a bigger r. To calculate the optimal starting point of these values we can utilize the LSH S-Curve, which has the formula $P(s) = 1 - (1 - s^r)^b$, where s is the similarity. Since the assignment's goal is to find all the documents with similarity above 0.9 we can fix this parameter and compute the False Positives and the False Negatives for the other two. For this we have to calculate the area under the FP and FN curves and normalize by the threshold width to get the average rates.
Figure 1 depicts FP (dashed lines) and FN (dotted lines) for different configurations of b and r as well as different amount of hash values (depicted with different colours). Since the goal is to retrieve all truly similar pairs (i.e., minimize false negatives), configurations with the lowest FN rates are preferred, even though they yield higher FP rates.

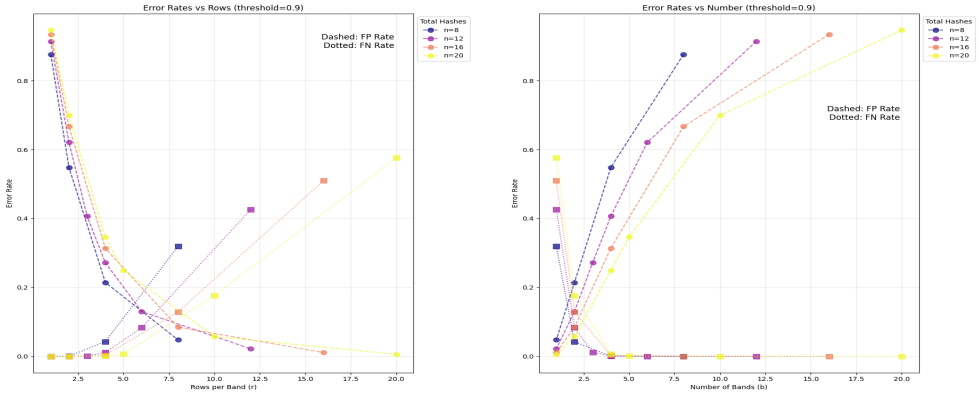

Figure 1: FP and FN rates for different b and r values for different amount of hash functions

2.5 Selection of number of unique shingles

The selection of unique signature count is again a task of balancing between precision, recall, and computational efficiency. A more verbose vocabulary increases the number of elements in each input set, leading to longer MinHash signatures and therefore higher computational and memory costs. However since we have more unique shingles per document this tends to reduce random collisions in LSH buckets and hence have fewer candidates to check and fewer false positives at the expense of memory footprint.

2.6 Python driver script

In order to optimize hyperparameter tuning, a python script that performs a hyperparameter sweep was created. The script check all available key LSH parameters, including the number of hash functions, bands, and buckets, while ensuring valid combinations (e.g., the number of bands always divides the number of hashes evenly). During execution, the script captures and parses runtime output to extract metrics, such as: Time taken to compute the signature matrix, time taken for the LSH banding and candidate pair generation, total execution time and average bucket size. In order to showcase sub-optimal LSH parameters, poor configurations did not get pruned but a 15 minute runtime limit was enforced for all runs. The results are presented in the following section.

2.7 Experiment results and discussion

The following table depicts the average execution time as well as the average bucket size and number of found pairs for a list of possible configurations. Time is also divided in two categories; t_{sig}, which is the time for signature matrix creation and t_{rest}, which is the time needed for the rest of the algorithm. The last 3 entries are configurations that led to more than 15 minutes of runtime and therefore timed out.

The results seem to evaluate the theory; First, the time to build the MinHash signature matrix grows roughly linearly with the number of hash functions and accounts for roughly 50 - 60% of the total allowed runtime.

Regarding the number of unique shingles, their increase to 2000 contribute to a slight rise in the amount of pairs found but further increasing to 3000 causes a slight decline, likely because documents become "too unique" reducing collision probability.

#	Hashes	Bands	Buckets	Shingles	t_{sig} (s)	t_{rest} (s)	t_{tot} (s)	AvgBucket size	Pairs
1	8	1	300000	2000	140.26	5.31	145.57	16.67	2918728
2	8	1	500000	3000	147.26	4.85	152.11	10.00	2837893
3	8	1	100000	2000	150.43	6.56	156.99	50.00	2918728
4	12	1	300000	2000	154.00	5.73	159.72	16.67	2559372
5	12	1	300000	1000	155.68	5.84	161.51	16.67	2519239
6	8	2	500000	3000	146.32	18.21	164.53	10.00	2837893
7	12	2	500000	3000	156.87	10.58	167.46	10.00	3464715
8	8	2	100000	3000	148.26	20.31	168.57	50.00	2837893
9	12	2	100000	2000	154.27	16.33	170.59	50.00	3494542
10	16	1	500000	3000	165.28	5.71	170.99	10.00	2436582
11	8	2	300000	1000	147.95	24.42	172.36	16.67	2900101
12	16	2	500000	2000	165.02	9.97	174.99	10.00	2909917
13	20	1	300000	1000	177.59	6.34	183.93	16.67	2327880
14	20	1	500000	3000	178.15	6.17	184.32	10.00	2342210
15	20	2	100000	1000	176.48	15.61	192.08	50.00	2832343
16	20	2	300000	3000	177.81	11.02	188.83	16.67	2827430
17	20	4	100000	3000	179.36	30.90	210.26	50.00	3044182
18	20	4	300000	1000	176.08	28.36	204.44	16.67	3047325
19	16	4	100000	1000	164.20	70.71	234.90	50.00	2856288
20	8	2	300000	2000	145.82	92.44	238.27	16.67	2918728
21	8	2	100000	2000	147.41	103.25	250.66	50.00	2918728
22	20	5	300000	3000	178.62	104.68	283.30	16.67	3044182
23	12	3	300000	2000	156.67	142.68	299.35	16.67	3494542
24	8	4	100000	1000	—	—	—	—	—(T)
25	8	4	100000	2000	—	—	—	—	—(T)
26	8	4	100000	3000	—	—	—	—	—(T)

Table 1: Selected 30 LSH configurations: trade-off between run time and retrieved pairs. The last three rows are timeouts (T).

It seems that 2000 unique shingles are the sweet spot for our problem. Additionally, when a configuration with 500 shingles was tested internally, an explosion in runtime was observed, most likely due to the lack of shingle expressiveness and a resulting rapid rise in candidate pairs.

Number of buckets is inversely proportional to the size of them; On average this also leads to slightly faster candidate filtering. Beyond 300000 bucket diminishing returns are observed; Further increase leads to almost the same runtime, but memory overhead grows needlessly for mostly empty buckets

The number of bands is the dominant performance factor. With 1 band, the filtering phase t_{rest} averaged just 6 seconds, since each document is compared only once. As we grow the number of bands, and essentially reducing the rows per band, the candidate set grows rapidly and therefore smaller bands make it more likely that dissimilar signatures collide and therefore T_{rest} grows sharply.

To optimize for high-similarity detection (threshold = 0.9), the best configuration is:

- Hash functions: 12–16 (balances signature cost vs. discrimination)
- Bands: 1–3 (limits candidate explosion)
- Buckets: ∼300000 (yields average bucket size ≈20)

This combination locates all near-duplicates well under our 5-minute limit.

References

[1] Scuderia Ferrari Strategy Team. *Ferrari We Are Checking Compilation*. `https://www.youtube.com/watch?v=g-DnxNc3GpQ`. [Classic F1 radio exchange; Required viewing for all students of over-engineering. Transcript features 15 variations of "Standby..." and 7 existential crises.] 2022.

[2] Web3 Waffle. "Locality-Sensitive Hashing: An Engineer's Guide to Scaling Similarity Search". In: *Medium* (2023). Accessed: 2025-05-02. URL: `https://medium.com/%40web3waffle/locality-sensitive-hashing-an-engineers-guide-to-scaling-similarity-search-81d196d1591b`.

[3] Stack Overflow Community. *Number of Buckets in LSH*. Stack Overflow. Accessed: 2025-05-02. 2016. URL: `https://stackoverflow.com/questions/37171834/number-of-buckets-in-lsh`.