Big Data Analytics Programming
Assignment 1: *Locality sensitive hashing*

Milos Dragojevic
r0780142
milos.dragojevic@student.kuleuven.be

15 December 2020

1 Introduction

Brute force is a method which calculates the similarity of an input by comparing it with all other inputs. This requires high memory usage and computation time. Locality-sensitive hashing (LSH) is an alternative method which hashes similar inputs into the same buckets. Only inputs ending in the same buckets will be compared to calculate their similarity. This method could lead to a smaller memory use and a reduction of computation time.

2 Implementation Description

The main implementation decision was to only keep in memory the necessary objects to be able to run on the full question set, as to remain within the memory bounds of 2GB maximum required to run on department's machines. This results in the algorithm computing similarities on signatures and not on shingles as these require too much memory.

The algorithm firstly reads a single question from which it gets its shingles, then computes from the document shingles the document signature using the hash functions. The question's signature is then added to the signature matrix and the algorithm then reuses the shingle variable for the shingles of the next question. In this way, the number of objects in memory was is minimised by only keeping for the entire runtime duration the signature matrix and the hash function parameters in memory, all the other variables are reused.

Another memory decision is that the implementation defines the question matrix as a 2-D int array. This is because the int type is only using 4 bytes of memory while the Java Integer type uses 16 bytes which would mean for signature matrix of 1250000 questions with 100 rows that it would take 500MB of memory with the int type and 2GB with the Integer type which would exceed 2GB memory limit.

Another implementation decision was to store into memory only the candidate buckets of one band at a time as the type of the object (Map<Integer, List<Integer>>) used is very memory consuming.

To further minimise memory usage, the garbage collector was regularly called after having analysed all similar pairs in a band. Thereby the garbage collector has a limited impact on the runtime and is useful to keep memory usage and fragmentation low.

The implementation also defines a CheckMem class to measure at regular intervals the consumed memory and report this usage to tune the code.

As to avoid potential user error, a control was set up in Runner to ensure that numHashes is a multiple of numBands. If it is not a multiple of numBands, the number is changed to the next multiple.

Concerning execution time, appending strings using the standard Java notation is tremendously slow. Therefore the implementation switched to using the StringBuilder class to build the strings to save the signatures in the CSV file at a higher speed.

During the test of the implementation, checking the false negatives was done using a trimmed down Duplicates.xml file which only contained the 12449 pairs existing in the Questions.xml file as to decrease drastically total runtime. For the full dataset this allowed reducing the time necessary for the testing of the duplicates pairs from 45 minutes to 45 seconds. As a result, many more runs were performed then otherwise possible.

To limit disk accesses, the program only reads the Questions.xml file once, performs all calculations in memory and then saves at the end the result file and if requested the CSV signature file.

It is worth noting that there are four pairs identified as duplicates in the duplicates XML file which are pairs with themselves as they have the same questions IDs. The program would report them as false negatives when they are in fact errors.

3 Experiments

The experiments firstly focused on finding a combination of parameters which would find the similar pairs in a set of 20000 questions. These parameters were then applied to LSH with the expectation that the results are equal or better compared to the performance of brute force.

3.1 Brute Force

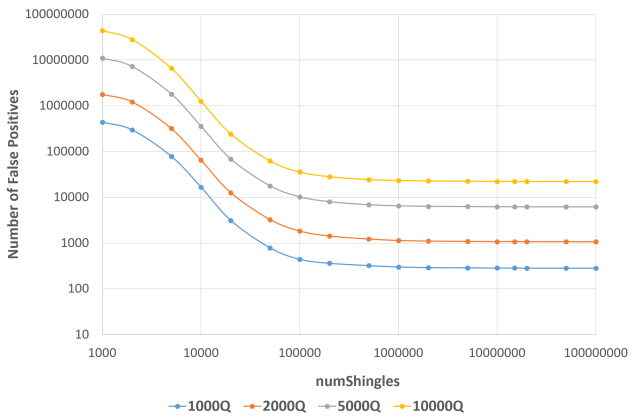
3.1.1 Shingle length

Suppose that only the 26 letters of the alphabet and blanks spaces appear in the questions. For k equal to 5 it means that the number of different shingles is 27^5, so 14.348.907 possible shingles. However, some characters predominate the questions (such as blank spaces and the letter a), and are therefore more likely to appear than other characters (such as the letter z). This would mean the number of possible shingles would be lower, closer to 20^5. All the questions in the Questions.xml file were analysed with the minimum length of characters being 65, the maximum length 36953 and the average length 1040. Since all the questions are shorter than 14.5 million characters, **$k = 5$ is the right choice for the implementation**.

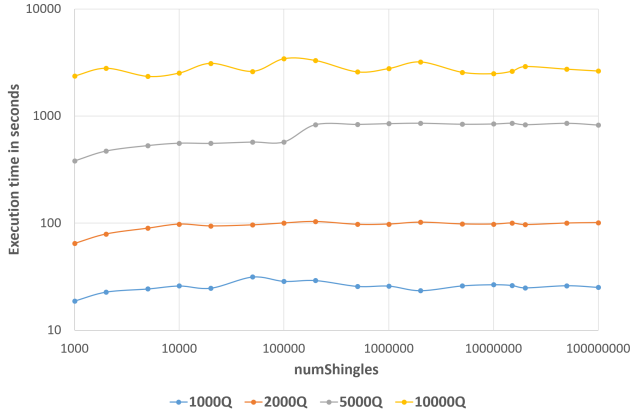
3.1.2 numShingles

Following from section 3.1.1, taking a small number of shingles will lead to many shingles collisions thereby falsely identifying document as duplicates which increases the number of false positives as can be seen in figure 1a. Thus, a high number of shingles needs to be taken as it will not impact negatively computation time as seen in figure 1b.

Further, by increasing the number of shingles while retaining all other parameters equal in brute force, the number of false positives will decrease massively until reaching a plateau of false positives at around 15000000 numShingles as seen in figure 1a. **Thus numShingles is chosen to be equal to 15 million in the implementation**.



(a) Brute force false positives vs -numShingles parameter

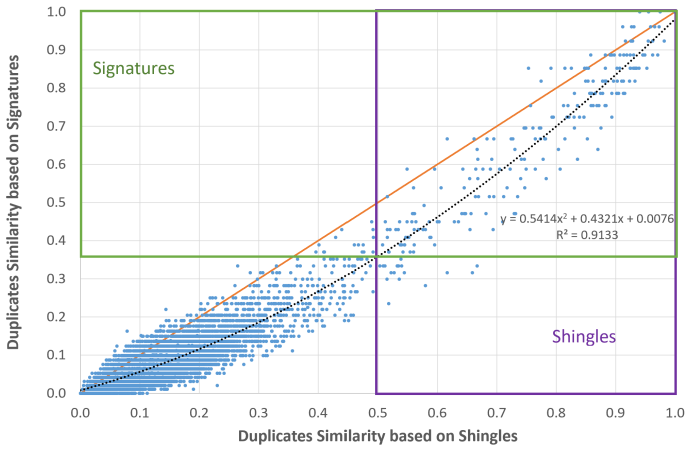


(b) Brute force execution time vs -numShingles parameter

Figure 1: Impact of numShingles on false positives and execution time in brute force

3.2 Similarity

Brute force is incapable of running on more than 20000 questions as the time it takes to run increases quadratically with the number of questions. As the performance of brute force is poor, an analysis of the Duplicates.xml file was performed to examine the similarity of the identified duplicates and to know the number of duplicate pairs above a certain threshold for a defined number of questions.



Similarity Range	Shingles		Signatures	
	Count	*Percentage*	*Count*	*Percentage*
[0.0-0.1]	5373	43.86%	9833	80.28%
]0.1-0.2]	5367	43.82%	1872	15.28%
]0.2-0.3]	982	8.02%	266	2.17%
]0.3-0.4]	215	1.76%	70	0.57%
]0.4-0.5]	79	0.64%	43	0.35%
]0.5-0.6]	56	0.46%	27	0.22%
]0.6-0.7]	32	0.26%	30	0.24%
]0.7-0.8]	35	0.29%	32	0.26%
]0.8-0.9]	53	0.43%	50	0.41%
]0.9-1.0]	57	0.47%	26	0.21%
Total	12249	100.00%	12249	100.00%

Figure 2: Comparison of duplicate pairs similarities for 1.250.000 questions

To this end, the similarities of the Duplicates.xml pairs were calculated in two distinct ways. Firstly, similarities were computed based on shingles, secondly similarities were computed based on signatures.

The analysis of the duplicates file revealed that the false negatives check does not account for the fact that some pairs from the duplicate XML file have a similarity lower than the threshold and should therefore not be counted as false negatives. The F1 score was used in the analysis, but it should be noted that its calculation is not perfect regarding the number of false negatives.

3.2.1 Similarity based on shingles

Using k-shingle of 5 and a numShingles of 15 million, the similarity between the pairs of the Duplicates.xml file based on shingles was calculated for different numbers of questions ranging from 1000 to the full dataset (1.25 million) to see the distribution of this similarity amongst the questions.

Further, it is seen from figure 2 that most of the duplicate pairs have a similarity based on shingles in the range of threshold 0.0 and 0.4 when looking at the full dataset. Similarity based on the shingles indicates that 233 pairs would have a similarity above a threshold of 0.5.

3.2.2 Similarity based on signatures

Using a numHashes parameter equal to 50, to obtain approximately 233 pairs based on the signatures the threshold needs to be lowered to 0.36 as seen in the figure 2. However, some of these pairs will not be the same as indicated in the figure 2 by the two rectangles.

Threshold of 0.5 is chosen for the implementation as it is high enough to identify most of the pairs having a signature similarity with a value representative of the shingle similarity.

3.3 LSH parameters

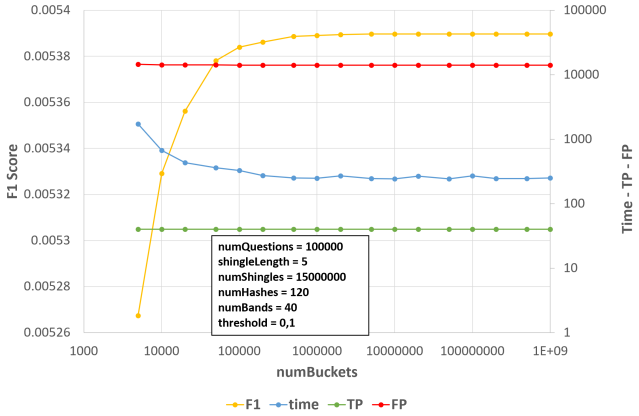
The analysis further explains for LSH the influence of parameters such as numHashes, numBands and numBuckets on execution time and amount of true positives and false positives. This was done using 100000 questions as this would be a good representation of the results when looking at the full dataset and would not take much execution time.

3.3.1 numBuckets

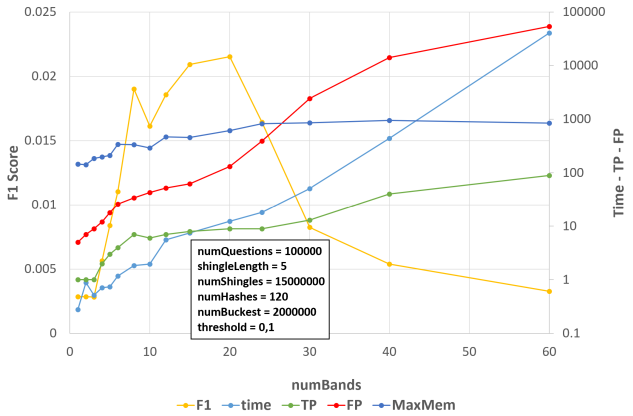
It is seen in figure 3a that the number of buckets influences time when its value is lower than 100000, and it does not affect the number of true positives and false negatives. **The numBuckets further affects the F1 score when its value is under 2000000 which is therefore the chosen numBuckets value of the implementation.**

3.3.2 numHashes vs numBands

With a constant number of hashes, execution time and number of false positives increase substantially with an increase in the number of bands as seen in figure 3b.



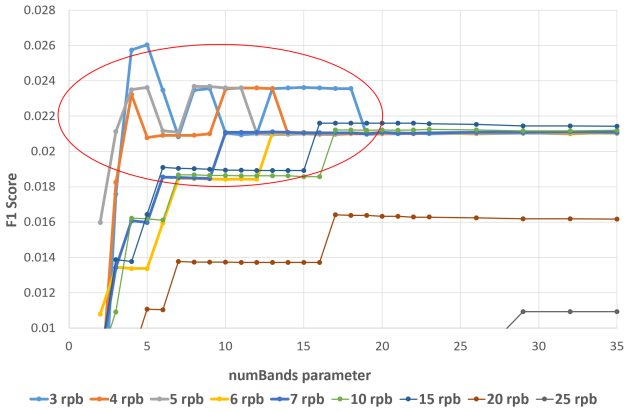
(a) Influence of numBuckets on performance measures



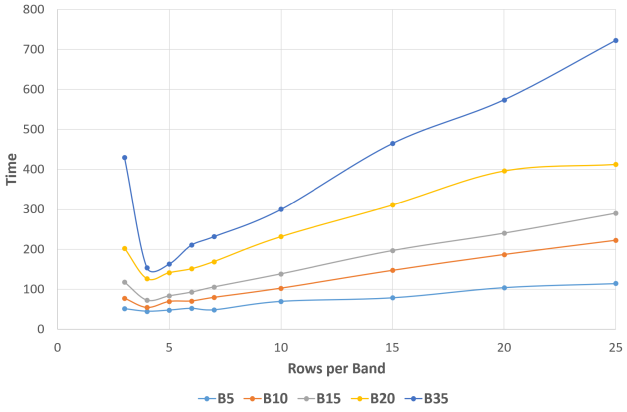
(b) Influence of numBands on different performance measures

Figure 3: Impact of numBuckets and numBands on performance measured with 100000 questions

In figure 4a, all the different rows per band lines join the same F1 Score plateau as the numBands increases. The ideal numBands would be located between 3 and 20 (circled in red) as this is where the peak F1 occurs. However putting a very high number of rows per band is non-optimal as time increases linearly with the number of rows per bands as seen in figure 4b. Thus number of rows per band between 3 and 7 are the levels that reach the ideal F1 Score the fastest and who take little execution time.



(a) Influence of numBands on F1 Score with 100000 questions (rpb refers to rows per band)



(b) Influence of rows per band on time

Figure 4: Impact of numBands and rows per band on F1 Score and time

3.4 Choosing the right command

For 100000 questions, different combinations were tested using number of bands ranging from 3 to 20 and number of rows per band ranging from 3 to 7. These runs were then ranked using the F1 descending and the top 20 most performant combinations of numBands and numHashes whose memory usage was below 2GB were selected to be able to run on the departmental machines.

The top results occurred when the numbers of rows per band was equal to 3, 4 or 5 as these give the best trade-off between runtime, F1 score and memory usage. The parameters of these best performing runs were then applied to the full dataset, with each each parameter combination being ran 11 times using each time a different seed. The performances were then averaged for each combination to select the final command.

Nine out of the twenty top combination were removed as their run time were excessive for certain seeds. Most of these were combinations for 3 rows per band, generating lots of collisions and thereby needing lots of time.

For the 11 remaining combinations, figure 5 shows that the combination which offers the best performance regarding F1 score is with 10 bands and 40 hashes which is in line with the analysis performed in section 3.3.2.

The average processing time is 772 seconds and the average memory use is 849MB while the false positives and F1 score stay very stable.

numHashes	numBands	Average of time	Average of TP	Average of FP	Average of FN	Average of F1	Average of MaxMem	Count of seed
40	10	772	160	1409	12089	0.02309	849	11
52	13	1025	158	1322	12091	0.02308	907	11
55	11	869	156	1280	12093	0.02276	968	11
44	11	835	156	1331	12093	0.02275	868	11
48	12	895	155	1271	12094	0.02272	888	11
50	10	763	154	1256	12095	0.02260	940	11
40	8	655	154	1318	12095	0.02247	891	11
45	9	721	151	1234	12098	0.02214	920	11
25	5	476	149	1375	12100	0.02158	799	11
20	4	438	140	1394	12109	0.02026	701	11
12	4	777	153	3812	12096	0.01965	584	11
Grand Total		933.9056906	153.2230216	1533.985612	12095.77698	0.022091364	826.705036	139

Figure 5: Performance of LSH ranked by F1 Score with top performing numBands and numHashes combinations

Thereby, the implementation chooses numBands equal to 10, the numHashes parameter is set to 40 and the seed equals 7825942 as it gives the best results.

4 Parameters Evaluation

To first confirm that the implementation of LSH and the parameters selected were optimal, these parameters were applied on brute force using 20000 questions. LSH was consistently better when compared to brute force as seen in figure 6. Using the threshold of 0.5, LSH was on average 1041 times faster, used 45 times less memory, resulted in the same number of true positives and false negatives and a lower number of false positives and a better F1 Score.

method	maxQuestions	shingleLength	numShingles	numHashes	numBands	numBuckets	threshold	time	TP	FP	FN	F1	precision	recall	MaxMem
lsh	20000	5	15000000	40	10	2000000	0.5	9.24	1	11	52	0.030769231	0.083333333	0.018867925	36
bf	20000	5	15000000				0.5	9618.04	1	21	52	0.026666667	0.045454545	0.018867925	1623

Figure 6: Performance of brute force vs LSH with the chosen parameters for 20000 questions

The final command with the optimal parameters and the best seed takes **742.137 seconds to run**, it saves the signature matrix in 2.48 seconds. The test results are true positives equal 172, false positives equal to 1379, and false negatives equal 12077 with the F1 score being equal to 0.0.024927536. The maximum memory used is 847 MB.

5 Conclusion

The implementation of LSH led to a performant balance between execution time, memory usage and F1 score. Using the similarities based on the signatures provided the advantage of reduced memory usage which was necessary to result in a viable and efficient solution.

Further research could focus itself on the fact that the duplicates XML file contains a majority of pairs which have a shingle similarity inferior to a specified threshold. Therefore these should not be counted as false negatives as it is impossible to find them with a threshold higher than the similarity.

Analysing the pairs of the Duplicates.xml file using the implementation parameters with 15000000 shingles (parameter that influences similarity) and a threshold of 0.5, only 233 pairs have a similarity superior to 0.5. Therefore does it make sense to report the other 12216 pairs as false negatives?

Calculation were redone with 233 pairs and using the now corrected F1 score, it was seen that the chosen combination of parameters yielded results with 160 true positives and 73 false negatives and thereby a corrected F1 Score was recalculated which results in a value of 0.18 which is a more realistic representation of the implementation's performance.