Assignment 2
Big Data Analytics Programming

Evangelos Ntavelis
r0692337
evangelos.ntavelis@student.kuleuven.be

February 16, 2018

Contents

1 Introduction

Firstly, we are are going to discuss the choices that were made when implementing the LSH algorithm. Afterwards we are discuss the effect of the parameters' values on the task performance and execution time and lastly we are going to motive our choice for parameter for running on the whole dataset.

2 Implementation Choices

There are two major keywords describing our approach to developing the solution for this assignment: Objects and Hash-Maps/Sets.

In order to be able to write code in a sustainable and manageable manner we followed the objected oriented principle. We had both LSH and BruteForce classes to extend a basic Searcher Class. We also created a class for the Universal Hash to help us structure better the creation of the Signature Matrix. This later proved useful as we also Implemented a second LSH class for short Signatures.

A very important choice during the implementations was the data structures we used. For the most of the cases a HashSet provided the most performant solution. Yet, for the

case of the Signature Matrix a two dimensional array provided a better solution due to the dense nature of the Data.

There were three ways to implement the **Universal Hash**: store the hash values on an array or a HashSet or call the function each time. By testing all approaches we found out that even for large shingles, the array solution is the best (3 quicker than HashSet and better than function calling). However, when dealing with the whole dataset we found out that for small shingle numbers it was better to store an array but for bigger to just call the function. Having two implementations for short and integers permitted us to implement this both ways according to the shingle size.

The design and creation of the **Signature Matrix** was the most important aspect of our program both time-wise and space-wise. The first important choice was in cases we are using a number of shingles that can fit into *Short Type* and by doing so we could half the space needed and improve the overall time. We created two version of LSH, one with short signatures and one for Integers.

Given also the way java saves multi-dimensional arrays in memory it made sense to create $\#MinHashes \times Arrays - of - \#Tweets$ both because the number of tweets is a lot bigger than the MinHashes as well ass the way we traverse the Matrix at its creation and the buckets assignment. This way we have less Cache Misses.

Another significant choice was how to handle the **Similarity Pairs**. We wanted to make sure that there are no duplicates in our Similar Pairs HashSet but also we didn't want to check the similarity of pairs we have already decided they are similar. In order to do, so we added the functions *equals()* and *hashCode()* to the Similarity Class. We followed the language guidelines to make sure that the two functions are designed as intended [1]. If two Pairs end up being in the same bucket we first check if we have already assigned as similar and we don't redo the calculations. However, if the two pairs have already been checked for similarity and got rejected, if they hash in to the same bucket again for another band, we are going to redo the calculations. The reason is that the probability of finding again a rejected pair is quite low if we take into account that a subset of the signatures which are similar was not enough to deem them similar, and thus we didn't want to introduce an overhead to check this out. The effect is larger when the threshold is smaller. Also, we made sure that the Pairs are examined by having the smaller id always first.

In order to measure and evaluate the solutions we used Python. We run the experiments locally and we used Python to calculate the TN, FP, FN as well as the derived performance metrics: Precision, Recall and F1-score and execution time.

A last remark: we could also try to write parallel code both for the transformation to the Signature Matrix, document-wise as well to the bucket assignment procedure, band-wise.

3 Parameter Analysis

Number of Shingles: As the size of the vocabulary grows larger we expect to have more collisions in the Shingle Hashing process if we don't accordingly increase the number of Shingles.

[1] http://programmergate.com/working-hashcode-equals-java/

More collisions mean that that our algorithms, both BruteForce and LSH, we'll predict more False Positives as it will regard different substrings as same. On the other hand there is a point where increasing the number of shingles won't alter the result. There if we don't have a significant difference in performance due to the HashSet implementation, by using a Number of Shingles that can be represented by a *Short Type* we can save half the space we are using. This in fact permits us to double the size our *Signature Matrix* before reaching our limits when dealing with *Big Data*.

Moreover, we found out that for the 100.000 sample the while we have a significant improvement in precision from 1000 to 10.000 shingles ($0.8 \rightarrow 0.875$ for 8 bands and 10 rows), it flats out when going to 100.000 and 1.000.000 shingles.

Our insight agrees to the empirical results for 100.000 tweets on the following table.

Nb of Shingles	BruteForce Pairs
1000	7232
10000	6955
100000	6955

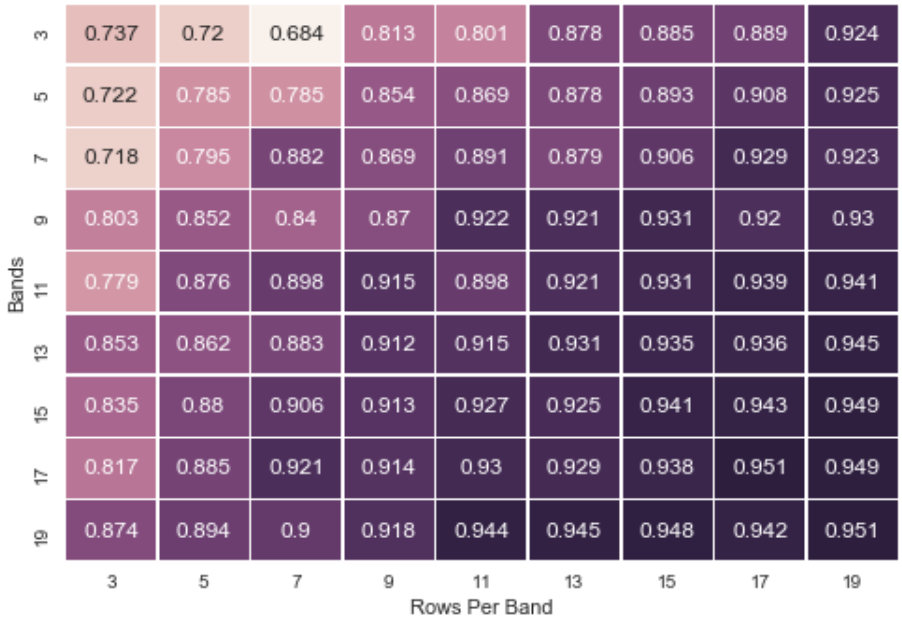
Shingle Length: By increasing the shingle length we observed the execution time going up, while recall value was going down and precision going up. The increase in time is expected as the reading takes k*all characters steps. Increasing the K will result to more combinations and to reduce the collisions we should also increase the number of shingles.

Threshold: By using the same combination of bands and rows as for the 0.9 threshold, we can see the effect described in the pdf and shown in Figure 5. The high S-curve threshold makes our model to produce nearly no False Positives but it Produces a lot of FN as well (2 times the TN). As expected, we are dealing with a lot more predictions both on the lsh and the brute force algorithm. Yet we can manipulate the performance in a similar way with higher thresholds using the S-curve theory.

Threshold	#Pairs
0.5	35162
0.7	18383
0.9	6955

Number of Buckets: Having a small number of buckets compared to the number of tweets we examine has a bad effect on both regards. Pigeonhole principle states that having less "holes/buckets" than "pigeons/tweets" will result to collisions. The more collisions we have the more dissimilar pairs we are going to check. The number of similarity checks grows exponentially as the number of the tweets per document increase. Here we don't have a problem to increase the number of buckets as it doesn't affect the execution time as we see in the following table which describes the running time for 10 runs and 100.000 tweets locally:

Figure 1: Precision



Nb of Buckets	10*Execution Time(sec)
1000	786
10000	184
100000	122
1000000	113
10000000	109

We can observe that the time goes down exponentially. At the same time we don't have any differences performance-wise.

Signatures, Bands and Rows per Band. These three parameters are the most important and can be regarded as two as:

$$Bands * RowsPerBand = \#Signatures$$

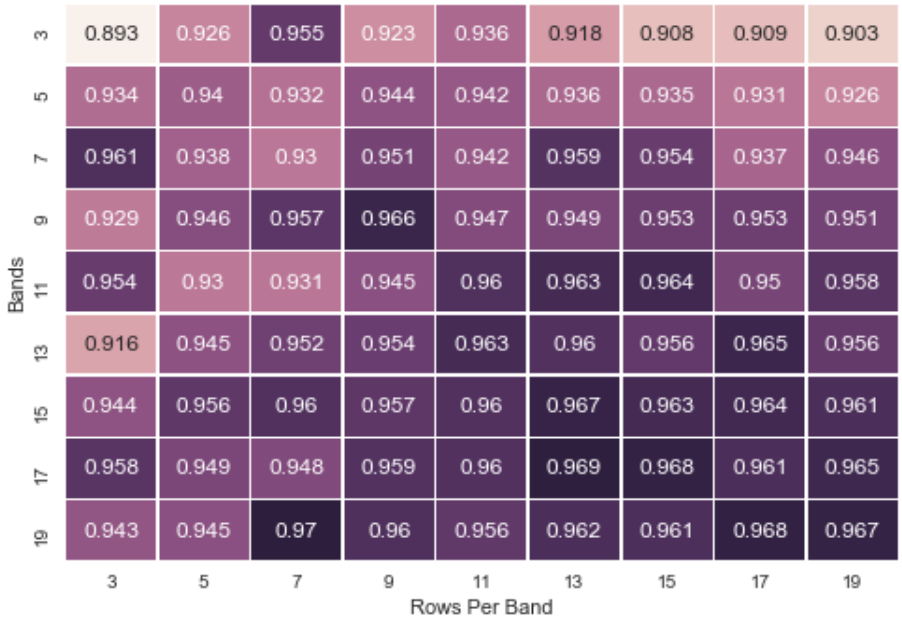
The experiments of the results presented here were run with the following settings, and the results is the average of 5 running times:

For all cases we can see that

Parameter	Value
nTweets	100.000
nShingles	10.000
ShingleLenght	3
nBuckets	100.000
Threshold	0,9

We can observe the following:

Figure 2: Recall



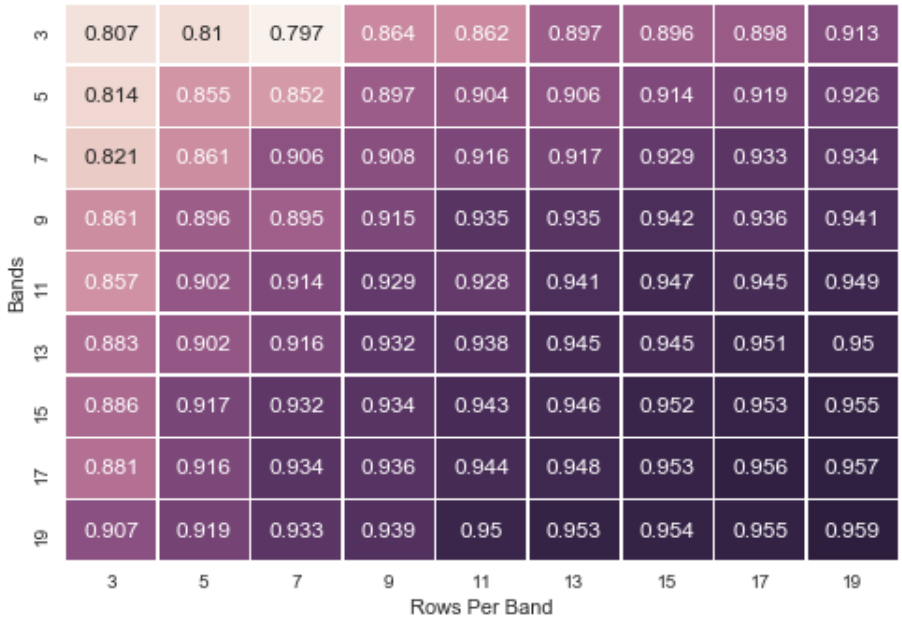
- As we increase the signature size (moving down- and left-wards) we are increasing the overall performance but the execution time also increases. The size of the signature matrix was the biggest bottleneck on the execution time.
- By keeping the number of signatures(minHashes) constant we find out that as we increase the number of rows and decrease the number of bands we increase the precision but decrease the recall value. The f1-score remains approximately the same. Yet as precision goes up, both FPs and TPs go down.
- Especially when running on the whole dataset we can observe the differences in execution time:

Bands	Rows	Running Time
20	4	1:03:47
4	20	0:07:36

Short VS Integer Signatures There were many experiments that took place in order to evaluate which way is the better. We found that by setting the number of Shingles to 1 million, we produce 152643 distinct shingles for the our 100.000 sample and 415500 for the whole dataset. Following the chart of characters 3-grams for *Zipf's law*[2] we expect the additional shingles to be less frequent that those observed firstly in our sample. We also found that maximumly at 10.000 shingles the performance evens out for our sample set. Based on the above we can argue that 32.000 Shingles will be enough, given the execution time and storage boost.

[2]I. Sicilia-Garcia, E & Ming, Ji & J. Smith, F. (2009). Extension of Zipfs Law to Word and Character N-Grams for English and Chinese. 8. .

Figure 3: F1-score



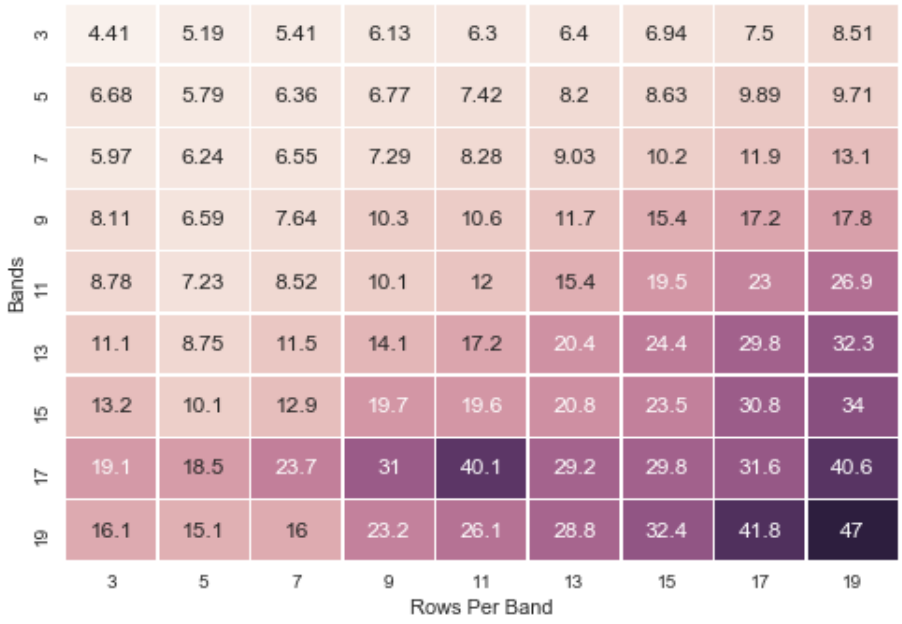
4 Going Big (Data)

There are two things we want to optimize: execution time and TP/FP performance. The first is pretty straight forward, for the latter we will try to maximize the value of precision as:

$$precision = \frac{TP}{TP + FP}$$

- We picked the Shingles Number to be equal to the MaxValue a Short Number can take. This permitted us to increase the Signature Size
- We picked the buckets to be 10.000.000 following the experiments we made on the sample dataset: Less collision without overhead.
- In order to maximize the precision and also minimize the running time we will pick $Rows >> Bands$.
- We also choose the configuration based on the chapter provided on the assignment and following the lovely table on Figure 5 where the values of $(1/b)^{1/r}$ are computed. We pick a threshold bigger than 0.9 as we wanted to produce less FP and have less execution time.
- One thing that we encountered during experimenting on the departmental machines, sometimes some configurations work and others don't. We attribute this to the different load that certain machines are facing at certain times. It was made sure that the solution can be run in different machines and at different times with different workloads.

Figure 4: Execution time



- Finally, we picked 4*20 instead of 4*25, because of the execution time gain and the less memory needed.

Parameter	Value
Bands	4
RowsPerBands	20
nShingles	32.000
nBuckets	10.000.000
Running Time	real 06:14m (@Namen)

Figure 5: $(1/b)^{1/r}$

		2	4	6	8	10	12	14	16	18	20	22	24	26
	2	0,707	0,841	0,891	0,917	0,933	0,944	0,952	0,958	0,962	0,966	0,969	0,972	0,974
	4	0,500	0,707	0,794	0,841	0,871	0,891	0,906	0,917	0,926	0,933	0,939	0,944	0,948
	6	0,408	0,639	0,742	0,799	0,836	0,861	0,880	0,894	0,905	0,914	0,922	0,928	0,933
BANDS	8	0,354	0,595	0,707	0,771	0,812	0,841	0,862	0,878	0,891	0,901	0,910	0,917	0,923
	10	0,316	0,562	0,681	0,750	0,794	0,825	0,848	0,866	0,880	0,891	0,901	0,909	0,915
	12	0,289	0,537	0,661	0,733	0,780	0,813	0,837	0,856	0,871	0,883	0,893	0,902	0,909
	14	0,267	0,517	0,644	0,719	0,768	0,803	0,828	0,848	0,864	0,876	0,887	0,896	0,903
	16	0,250	0,500	0,630	0,707	0,758	0,794	0,820	0,841	0,857	0,871	0,882	0,891	0,899
	18	0,236	0,485	0,618	0,697	0,749	0,786	0,813	0,835	0,852	0,865	0,877	0,887	0,895
	20	0,224	0,473	0,607	0,688	0,741	0,779	0,807	0,829	0,847	0,861	0,873	0,883	0,891
	22	0,213	0,462	0,597	0,680	0,734	0,773	0,802	0,824	0,842	0,857	0,869	0,879	0,888
	24	0,204	0,452	0,589	0,672	0,728	0,767	0,797	0,820	0,838	0,853	0,865	0,876	0,885
							ROWS							