Big Data Analytics Programming Assignment 2

Grigoris Konstantinis

February 16, 2018

1 Important decision made when developing my algorithm

For this assignment my goal was to make the algorithm as fast as possible. To that extend I decided I will read the tweets and compute the MinHash signatures from the file only once. Furthermore, I tried to use primitive data types as much as possible, as Java Objects have significant overhead.

The most important and memory consuming part of my program is the 1-dimensional array `int[] signatures` that keeps in memory the signatures of all Tweets. The signatures of Tweets are saved in the following fashion. Tweet with ID `0` has its signature saved in the cells `signatures[0]`...`signatures[(b*r)-1]`, tweet with ID `i` has its' signature saved in the cells `signatures[i*b*r]`...`signatures[((i+1)*b*r)-1]`. Note that we have no need to store a "mapping" from a Tweet's ID to its' signature. IDs in the array are implicitly "stored", as long as the parameters `b` and `r` are known.

A 1-dimensional array has advantages compared to other data structures such as a `HashMap<Integer, int[]>`. Since Java Collections HashMap works only with Objects and since creating Objects is a slow process, HashMap is much slower and much more memory hungry. The the HashMap memory overhead for 7.500.000 tweets (0.75 loading factor) is going to be on average 400 megabyte in a 32-bit machine, and 750Megabytes of overhead in a 64-bit machine. Compared to the 1-D array which has an overhead of just a few bytes the memory overhead here is huge.

As I show later the FPs reduce as the size of signature increases, but due to the memory constraints I cannot use infinitively long signatures, following I propose 2 solutions to this problem.

The first (not implemented), is to read the tweets file multiple times doing part of the work each time. For example the following architecture. Compute the signature of each tweet one at a time, then use Murmurhash and hash it to a bucket, do not keep in the memory the signatures, just keep in the memory all tweets hashing to the same bucket. Then re-read the tweets that are relevant to the first X candidate pairs, compute their signatures again and confirm whether they are similar, do that for all candidate pairs. In this approach I am able to use larger signatures for the cost of speed, as re-reading tweets and re-computing signatures is very slow, it also leads to disk thrashing.

The second solution that I propose (implemented) is a 2-pass LSH - brute force algorithm. The main problem of LSH when I run it for a small signature size is that it produces many False Positives. In the first pass I use LSH with as large signature size as I can for the full dataset, and save the results of the LSH in the memory. Then in the second pass for each proposed pair from LSH I check if it really is similar using the Shingled representation of the Tweet, the same way the brute force approach does it. That way I ensure after the second pass all TP remain TP, but all FP are eliminated, Precision is guaranteed to be 100% which may be very important for some applications. The downside is that it requires extra computing time. If the number of similar pairs identified by LSH are 50.000.000 then the 2nd pass will require as much time as a brute force for 10.000 tweets (since $\frac{10000*9999}{2} \approx 50.000.000$), making it a very good trade off. Furthermore this approach is better than using LSH with a very large signature size even when the memory is not a problem. As I show in the 2nd chapter computing large signatures requires a lot of computing time.

2 Parameter analysis

The most important parameters are the shingleLength, nShingles, b, r and the size of buckets.

2.1 Shingle length, nShingles

Shingle length are very important parameters that change the results of both LSH and Bruteforce algorithm substantially depending on their number. The parameters nShingles is related with shingle length and the number of tweet. For a fixed number of tweets I expect at first as the shingle length increases the nShingles to also increase, but after some point as shingle length continues to increase the nShingles will start to decrease. As the number of tweets increase, the number for shingle length that result in the peak nShingles will get larger, therefore for larger number of tweets I will have a larger number of nShingles.

For a small shingle length and for large number of tweets I can expect that all combinations of characters appear. The total number of nShingles can be estimated by the following equations $nShingles = 80^{shingleLength}$, the reasoning for 80 is that there are 52 lowercase & uppercase English letters, 10 numbers and 18 special character that are likely to appear in a tweet (i.e. #). For shingleLength = 3, 7.400.000 tweets are large enough then nShingles = 512.000.

After conducting experiments I noticed the following: with smaller shingleLength more pairs are identified as similar and the number of nShingles do not greatly affect the algorithm as long as its number is not very small.

Another important observation for LSH is by using as nShingles a number that is less than 65.535 the signatures will not contain any number greater than 65.535, this means that instead of using an int array to save the signatures (4 bytes per cell), I could use a char array instead (2 bytes per cell), that way I could run the program for double the signature size with the same memory. Another important observation is that even if the number of nShingles is 512.000 there are very few cells in the signatures containing a number greater than 65.535 (just 0.0746%), that means even if I use a char array and substitute all numbers greater than 65.535 as 65.535 the results will be almost identical to the results of the algorithm with an int array. Nevertheless the version with an int array has been submitted as my solution, since its a more "correct" version of the algorithm.

2.2 Parameters b and r

These parameters affect the algorithm's speed, memory and accuracy. Parameters b and r are also related with the size of the signatures through the following equations $b * r = signatureSize$, obviously the larger their number, the larger the size of the signature, more memory is required to store the signatures and the algorithm also becomes slower as it takes more time to compute them. On the other hand bigger b and r will result in the algorithm being more accurate producing less False Positives. Another aspect of these parameters is that they control how candidate similar pairs are selected. Specifically regardless of the choice of b and r we create an S curve. The probability that a pair of tweets become candidate similar pair is given from the following equation $1 - (1 - s^r)^b$, with s the Jaccard similarity. Generally the bigger the b the more candidate pairs I will get, the smaller the r the more candidate pairs I will get. Furthermore the equation $(1/b)^{1/r}$ gives an approximation of the threshold.

Following you can see the results of experiments for 100.000 tweets, threshold=0.9, nShingle = 166375 ($= 55^3$) and shingleLength = 3. The bruteforce result for these setting were 6962 similar pairs.

Results for 100.000 tweets, shingleLength=3, nShingles=166375, thershold=0.9									
b	r	Signature size	$(1/b)^{1/r}$	TP	Recall $\left(\frac{TP}{TP+FN}\right)$	FP	Precision $\left(\frac{TP}{TP+FP}\right)$	Time (m:ss)	Memory
2	5	10	0.87	6710	96.38%	5540	54.62%	0:02	4 MB
3	8	24	0.87	6625	95.16%	1977	77.01%	0:04	9.6 MB
4	10	40	0.87	6704	96.29%	1740	79.39%	0:05	16 MB
8	15	120	0.87	6693	96.13%	628	91.42%	0:15	48 MB
12	18	216	0.87	6695	96.16%	368	94.79%	0:25	86.4 MB
30	25	750	0.87	6796	97.61%	221	96.85%	1:28	300 MB
45	28	1260	0.87	6750	96.95%	131	98.09%	2:31	504 MB
60	30	1800	0.87	6785	97.45%	137	98.02%	3:40	720 MB
90	33	2970	0.87	6803	97.71%	94	98.63%	6:22	1188 MB
50	2	100	0.14	6727	96.62%	856	88.71%	1:38	40 MB
33	3	99	0.31	6629	95.21%	591	91.81%	0:17	39.6 MB
20	5	100	0.55	6727	96.62%	856	88.71%	0:13	40 MB
10	10	100	0.79	6737	96.76%	947	87.68%	0:12	40 MB
7	14	98	0.87	6624	95.14%	583	91.91%	0:12	39.2 MB
6	17	102	0.90	6538	93.90%	525	92.56%	0:12	40.8 MB
5	20	100	0.92	6259	89.90%	423	93.67%	0:12	40 MB
3	33	99	0.967	5687	81.68%	70	98.78%	0:12	39.6 MB
2	50	100	0.982	5478	78.68%	10	99.82%	0:12	40 MB

Note: the memory is the estimated memory required for storing only the signatures.

First I will check the effect of the size of the signature $(b * r)$. As the signature size increases the number of FP decrease, on the other hand increasing the signature size does not seem to have a big impact in the number of TP. Note that increasing the signature size also increases the time and memory required. I also notice that as long as the value $(1/b)^{1/r}$ is below the threshold it does not seem to have a large impact on the number of TP. On the other hand if the value $(1/b)^{1/r}$ is above the threshold, TP (and Recall) start to decrease, but FP also decrease which result in increased Precision. A small $(1/b)^{1/r}$ result in more run time, while larger one result in less run time, that is because fewer pairs end up hashing in the same bucket.

It appears overall that the ideal choice of parameters **b** and **r** is a combination that make a large signature size and keeps the value $(1/b)^{1/r}$ a bit below the threshold, i.e. if the threshold is 0.90 then a good choice for that value is 0.87. That way both Recall and Precision is ensured to be high.

2.3 Number of buckets for each band

First I will check how the number of buckets affect the quality of the solution

Results for 100.000 tweets, shingleLength=3, nShingles=166375, thershold=0.9, b=4, r=10		
Number of Buckets	TP	FP
10.000.000	6696	1467
1.000.000	6696	1467
100.000	6696	1467
10.000	1231	1467
1.000	6696	1467
100	6696	1470

Note: In all Java.util.Random I use a seed, therefore my algorithm is deterministic.

I see that the number of buckets do not affect the quality, even when using extreme numbers. This makes sense as it is highly unlikely for two similar documents to hash in different buckets in all bands.

Next I will check the time performance of the different bucket sizes in 1.000.000 tweets, I choose more tweets this time since the algorithm finish too fast for 100.000 making comparisons difficult.

Results for 1.000.000 tweets, shingleLength=3, nShingles=166375, thershold=0.9, b=4, r=10			
Number of Buckets	Time (m:ss)	Memory in 32 bit machine	Memory in 64 bit machine
100.000.000	0:54	392 MB	781 MB
50.000.000	0:53	202 MB	400 MB
10.000.000	0:54	49 MB	94 MB
1.500.000	0:55	19.94 MB	26.08 MB
1.000.000	0:58	12.33MB	20.86MB
100.000	1:00	4.95 MB	6.10 MB
10.000	1:04	3.92 MB	4.04 MB
1.000	2:36	3.82MB	3.83MB
100	17:32	3.81MB	3.81MB

Note: In all java.util.Random I use a seed, therefore my algorithm is deterministic. All the memory estimates have been also confirmed using the memory profiling program Visual VM.

I notice that generally the larger the array the algorithm gets faster, but there are diminishing returns. For small bucket sizes the algorithms gets very slow. A good number of buckets considering the memory requirements and the speed efficiency appears to be a number between $1.5 * numberofTweets$ to $\frac{numberofTweets}{10}$.

3 Running for the full dataset

The total memory requirements for the signature of type int is $b*r*numberOfTweets*4bytes$, for $b = 4, r = 10, numberOfTweets = 7.416.113$ the requirement is 1,18 gigabytes. Another part of the program consuming substantial amount of memory are the array lists that remember the IDs of tweets that hash to the same bucket for each band. Using $numberOfBuckets = 10.000.000$, I estimate the memory requirements for the array lists to be around 200 megabytes. There are some other things saved to memory but they are not very important memory-wise, i.e. by the time they start to consume lots of memory, other things will have already been garbage collected.

Depending on the machine I can run the algorithm for different signature sizes. In aalst machine I can run the algorithm for signature size 40 only when using the JVM arguments -Xms1750M -Xmx1750M which set the min and max memory usage as 1750 mega bytes, but when I try to run the program without the JVM arguments the program gets killed. On other machines I can run the program for larger signature sizes than 40 without needing to use any JVM arguments. On namen I could run the program for very large signature sizes, i.e. 90 which normally should have been impossible as the signatures alone would require more than 2.7 gigabytes of RAM. Because of this variation of memory among different machines I will provide you with commands that include the JVM arguments limiting the maximum memory usage to ensure the program will not get killed, since with those JVM arguments my program works in all machines. Of course you are free to remove them since the algorithm may perform a bit faster if they have more memory since the garbage collector will not work as intensively.

As mentioned before a good choice for the number of nShingles for single length = 3 is $3^{80} = 512.000$.

The choice of b is going to be 4 and the choice of r is going to be 10, because I want the value $(1/b)^{1/r}$ to be slightly below the threshold and because there is significant variation of available memory among the different machines as discussed before, but my algorithm with signature size 40 runs in all of them.

The number of buckets I choose is 10.000.000 as then the memory requirements are still low and because the efficiency of the algorithm does not improve a lot above that number.

You can run the algorithm as follows:

java -XX:+UseCompressedOops -Xmx1750M -Xms1750M MyLSHRunner -threshold 0.9 -maxFiles 7416113 -inputPath /cw/bdap/assignment2/tweets -outputPath output.txt -shingleLength 3 -nShingles 512000 -b 4 -r 10 -numberOfBuckets 10000000

The results of the LSH algorithm for the above configuration in eeklo.cs.kotnet.kuleuven.be using the time command are the following:

LSH for the full data-set.	
Runtime	6 minutes 25 seconds
# Identified similar pairs	1.797.735
# TP	1.458.684
# FP	339.051
Recall	Expected to be around 96.5%
Precision	81.1%

The #TP, #FP, Recall, and Precision have been confirmed by running the 2-pass algorithm later with the same parameters and same seed for the Random number generator.

You can run the other 2-pass LSH - brute force algorithm I proposed in the 1st chapter that result in 0 False Positives using the following:

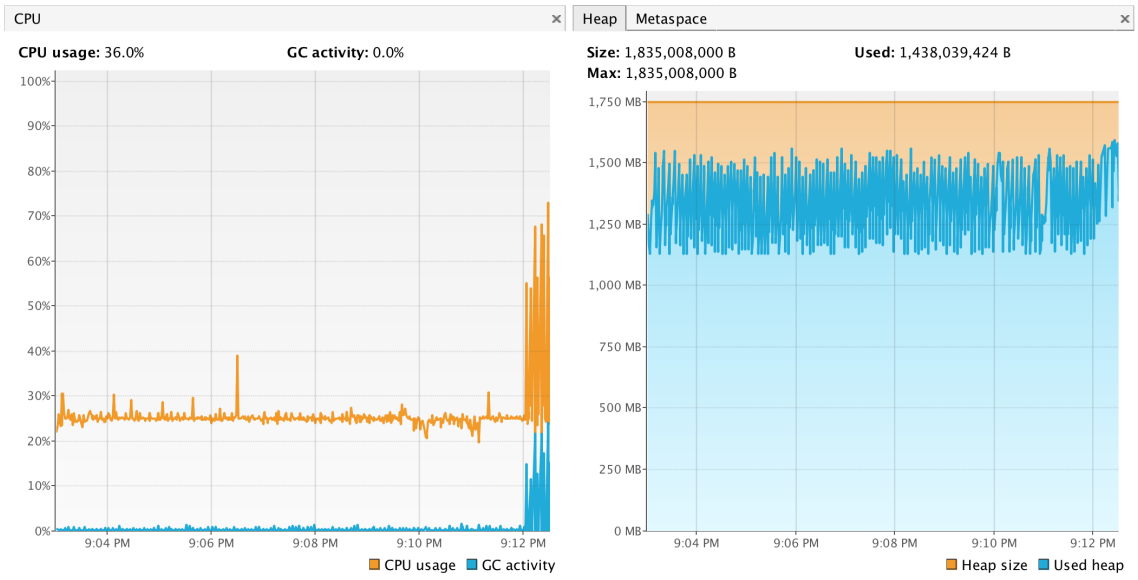
java -XX:+UseCompressedOops -Xmx1750M -Xms1750M MyLSHRunner2Pass -threshold 0.9 -maxFiles 7416113 -inputPath /cw/bdap/assignment2/tweets -outputPath output.txt -shingleLength 3 -nShingles 512000 -b 4 -r 10 -numberOfBuckets 10000000

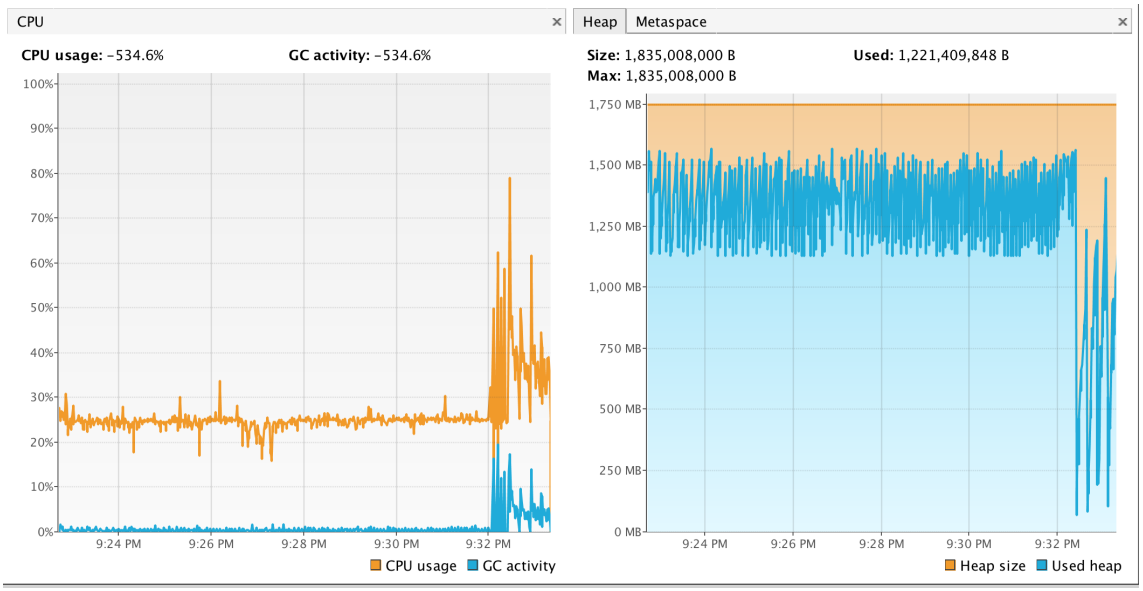
The results of the 2-pass LSH - brute force algorithm for the above configuration in eeklo.cs.kotnet.kuleuven.be using the time command are the following:

2-pass LSH - brute force algorithm for the full data-set.	
Runtime	6 minutes 48 seconds
# Identified similar pairs	1.458.684
# TP	1.458.684
# FP	0
Recall	Expected to be around 96.5%
Precision	100%

I see that the 2-pass LSH brute force algorithm is just 6% slower but identifies all False Positives. Furthermore this algorithm can be fine tuned to work even faster for a better selection of b and r but it is outside of the scope of this report.

Two screen shots from the memory profiling program VisualVM follow, when the program is run in my machine (please ignore the time stamps as my computer is slow). The first screen shot is the memory usage of LSH and the second screen shot is the memory usage of the 2-pass LSH - brute force.




CPU
CPU usage: -534.6%
GC activity: -534.6%
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
9:24 PM
9:26 PM
9:28 PM
9:30 PM
9:32 PM
CPU usage
GC activity
Heap
Metaspace
Size: 1,835,008,000 B
Used: 1,221,409,848 B
Max: 1,835,008,000 B
1,750 MB
1,500 MB
1,250 MB
1,000 MB
750 MB
500 MB
250 MB
0 MB
9:24 PM
9:26 PM
9:28 PM
9:30 PM
9:32 PM
Heap size
Used heap