Stream Analysis on the movie dataset

Course on "Algorithms for Massive Datasets", University of Milan

Jean-Baptiste Lepidi

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Experimental setup

This project uses Kaggle's Letterboxd dataset, available at https://www.kaggle.com/datasets/gsimonx37/letterboxd. The dataset was accessed on the 28$^{\text{th}}$ of May 2024 and has not been modified since at the time of redaction of this paper. For this project, we will consider only a fragment of the dataset, namely the genres, actors, and movies CSV files.

2 Algorithms and implementation

The goal of the project is to experiment on stream analysis. This will be done by carrying out 2 tasks : count the number of distinct elements and filter stream elements using a Bloom filter. All the algorithms used to carry out those tasks will have to be re implemented from scratch.

2.1 Count distinct

The goal of this task is to count the number of distinct names in the actors file. The file is organized according to a very simple structure : each tuple is composed of the id of the movie (a foreign key that references the movies

file) and the name of one actor. Therefore each movie is referenced several times in the file, in order to be associated to all its cast.

2.1.1 The Flajolet-Martin algorithm

The task of counting the number of distinct elements in a stream can seem quite easy, but when the data becomes massive and the number of distinct elements becomes very large, keeping track of all the elements that have been seen becomes impossible in terms of memory. To remedy this problem, we will use a probabilistic algorithm based on hashing called the Flajolet-Martin (FM) algorithm.

This algorithm approximates the number of distinct elements in a stream by keeping track of the longest "tail length" (number of tailing 0s in the hash value).

In terms of implementation it is quite straightforward : an integer counter to keep track of the longest tail length, a hash function and an estimator of the number of distinct elements. The chosen hash function in that instance is SHA256. The estimator for the number of elements is $\lfloor 2^m/\phi \rfloor$, with m the maximum tail length, and ϕ a correction factor. This correction factor is $\phi = 0.77351$ and its computation is detailed in the original paper by Flajolet and Martin (1).

2.1.2 Refining the algorithm

Seeing how the algorithm is probabilistic, and given how the result is approximated, we can quickly understand that it can be a bit unstable, and more importantly that the granularity is not that fine : we are dealing with powers of 2 so the distance between two consecutive estimations (one with m and with $m + 1$) can be quite large.

With this in mind, we can devise a way to refine the algorithm, by using several hash functions, in order to have multiple estimators and be able to aggregate them. For the aggregation, the mean is not that great because it is very sensitive to outliers, so we prefer the median, but since the value of the median is a value from one of the estimators, using it doesn't solve the problem of granularity. We will be using a hybrid approach, namely to use $k \times l$ hash functions, divided in l groups of k. The aggregation within the groups will be done using the median (reducing variance), and the aggregation of the group results will be done with the mean (for better granularity).

In terms of implementation, this refined version of the algorithm implies some changes :

- Instead of using one counter, we use a list of $k \times l$ counters.

- We need to use $k \times l$ different hash functions. This can be simulated by using a random seed to "initialize" the hash function.
- Since we will be using a lot of hash functions, we replace SHA-256 with a more efficient hash function, namely MurmurHash3. This function is not suitable for cryptographic purposes, but this is not a property that matter to us.

2.1.3 General processing

Other than the main algorithm, there is a need for a general processing of the data. The implementation is very simple, the idea is to simulate a stream of data by reading the actors file line by line (as opposed to loading the full file in memory and processing it). Each tuple will be processed by the FM algorithm according to the general scheme of the algorithm : hash the value, compute the number of trailing zeros, check it against the maximum number registered yet. Finally, we output the value of the estimator.

2.2 Bloom filtering

The goal of this task is to set up a Bloom filter. The idea of a Bloom filter is to have a memory efficient filter that can hold a huge amount of entries, by sacrificing a bit of accuracy since the filter is probabilistic. Our main purpose with this filter is to filter the movies on their genre, namely define a genre (in our case "Western") and be able to filter the dataset, assigning to each movie a label "Western" or "Not western".

2.3 The filter

The structure of the Bloom filter is quite simple since it is just a bitmap, and a series of hash functions. Two main parameters are to be taken into account : the size of the bitmap, and the number of hash functions used, both of which will have an impact on the performance of the filter.

Two important functions have to be implemented for this filter to work : add and query. The add function is used to add an entry to the filter. It works by hashing the value of the entry n times (with n the chosen number of hash functions) and setting the corresponding positions in the bitmap to 1. The query function hashes the value with the n hash functions and checks if all the corresponding positions in the bitmap are set to 1. In order to have an efficient implementation, just like with the FM algorithm, we will be using the MurmurHash3 hash functions that are very fast, and a seed to

simulate the use of different functions. Note that since we cannot control the output size of the hash function, we need to map the 64 bit output of the hash function, to a number between 0 and $size(bitmap) - 1$. Also note that since we are coding in Python, list elements are Python objects and therefore are quite large; in order for our filter to be memory efficient, we will be using the "bitarray" module that allows us to create actual bitmaps.

2.4 General processing

The general scheme of the processing is :

- Initialize the filter by adding to its entries all the movie IDs that are associated to the genre "Western". This is done using the genres file. Note that we could for example use the whole vector of data in the movies file but using simply the ID works perfectly well in our case.
- Then, go through the movies file and query the filter for each ID to see if the movie in question is or not a western.

It is divided in 2 main parts that both require to read a whole file from the dataset. Both parts are treated as stream processing, meaning that the initialization is done row by row, as well as the filtering, without the need to ever load the full file in memory.

Once the processing is done, we can compute the false positive rate to evaluate the accuracy of the filter.

3 Experiments and results

3.1 Count distinct

For the count distinct problem, several experiments were conducted, all on the complete actors file (no sampling) : running the simple FM algorithm without the correction factor, with the correction factor, running the refined version of the algorithm using 5 groups of 7 hashes and finally comparing the performances in terms of accuracy, execution time, and memory with its successor, the Hyperloglog (HLL) algorithm.

The results of the experiments are summarized in the following table :

	Execution time (s)	Error rate (%)
FM without correction	26.6	30.74
FM with correction	26.7	10.46
Refined FM (35 hashes)	225.3	7.45
HLL (accuracy 0.02)	23.7	3.36

Table 1: Summary of the experimental results : execution time and accuracy

Recall that in the results for the first 2 entries, the SHA-256 algoirthm was used, and that the execution time with a better fitted hash function would be way lower.

As it is made abundantly clear by the table, the HLL algorithm outperforms the FM algorithm by a lot, in terms of execution time and performance. We can see that the improvement obtained by using the correction factor is significant. However, even using a faster hash function for the refined FM, the added execution time is very important, and the improvement in terms of accuracy is not incredible.

The memory consumption of the algorithms is quite easy to compute. For the simple FM algorithm, the only thing we have to keep track of is one integer counter, so the memory consumption is that of one integer. For the refined version, we need to keep track of $k \times l$ integers. For the HLL algorithm, the memory consumption depends on the required accuracy. Referring to the original paper (2), we can compute the memory consumption in the following way : $m \times log_2(w)$, with m being the number of registers, and w the number of bits on which we need to count the number of tailing zeros (therefore, if $w = 8$, then there is at most 8 trailing zeros, and the memory needed is $log_2(8) = 3$ bits). The number of registers can be computed using the following formula : $m = (\frac{1.04}{\epsilon})^2$, where ϵ is the error rate specified by the user. w can be computed as $64 - b$, with b the number of bits needed to address the registers (namely $b = log_2(m)$). This formula comes from the fact that the HLL+ algorithm uses a 64 bit output hash function and that this output is split in two, the first b bits being the address of the register, and the rest (w) on which we count the trailing zeros.

Knowing all this, we can compute the memory consumption for our algorithms :

	Memory consumption (bytes)
Simple FM	28
Refined FM (35 hashes)	980
HLL (accuracy 0.02)	2918

Table 2: Summary of the experimental results : memory consumption

Note that here, what is supposed to be the size of an integer is very large (28) because we are using Python and that an integer in Python is a fully fledged object.

Overall, we can see that the HLL algorithm is not as memory efficient as the FM algorithm, even though the memory consumption is very low especially considering the accuracy of the results. A question we could ask is how much memory would be needed by the refined FM algorithm to attain the same accuracy (without even taking into account the execution time).

3.2 Bloom filtering

The experiments on the Bloom filtering activity are quite straight forward. The first thing we needed to do is estimate the size needed for the bitmap, and the number of hash functions, and once done we can simply initialize the filter, query it, check its accuracy, and modify the parameters as needed. Once again, the experiments have been conducted on the full dataset in order to have an idea of the scaling potential of the method.

After a first execution, we know that there are about 9000 western movies. Using n hash functions that means that at most $9000 \times n$ positions are set to 1 in the bitmap. We want to avoid collisions as much as possible so we want our bitmap to be relatively scarce. The chosen estimate was to have about 20 to 25% of the bitmap full (in the worst case where all the hashes give a different result). A first test was done using 5 hash functions, and therefore a size of 200000 bits. This lead to a false negative rate of about 3%, which is not bad but can be improved. To improve this result, we can increase the number of hash functions, and therefore the size of the bitmap accordingly. Using 10 hash functions, and a size of 500000 bits (to keep the same kind of scarcity), we get to a false positive rate of 0 !

The size of the bitmap and number of hash functions may be a bit over estimated, but 62.5KB is a very respectable memory consumption, and since we are using a fast hash function the whole processing takes very little time even with 10 functions. Overall, the initialization of the filter takes around 1-2 seconds, and the processing of the whole movies file takes less than 10 seconds.

4 Scaling up

Let's quickly analyze how the implemented algorithms scale with the size of the dataset.

4.1 Flajolet-Martin

For the count distinct problem, the full dataset of about 5.5 million names can be processed by the basic FM algorithm in less than 30 seconds, using a slow hash function (SHA-256). The execution time increases a lot when using the refined version, especially if the number of hashes is very high, since we process each line $k \times l$ times instead of 1. The execution time can quickly become very problematic when using a huge dataset. An increase of one order of magnitude (50M instead of 5M) would result in an execution time close to half an hour ! There are probably some improvements to be applied to the processing scheme, as well as some limitations of the language : in fact, Python is quite a slow language, and running the same algorithm is C could lower significantly the execution time.

In terms of memory, the algorithm scales very well since the memory consumption does not depend at all on the number of tuples that have to be processed.

The best choice would be to implement the HLL or HLL+ algorithms to solve this problem since the execution time and performances are way better, and the memory consumption is still fine.

4.2 Bloom Filtering

The Bloom filter scales very well in terms of execution time : as we have seen, both the initialization and the querying are pretty fast. However it is important to note that the performances of this filter entirely depends on the distribution of the dataset. If all the movies in the dataset were westerns, then the bitmap would have to be way larger, and the initialization/processing time would too !

References

[1] Philippe Flajolet, G. Nigel Martin, *Probabilistic Counting Algorithms for Data Base Applications*, April 1985

[2] Philippe Flajolet, Éric Fusy, Olivier Gandouet, Frédéric Meunier, *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*, 2007