# Mining data streams: estimating the number of distinct elements and the second moment

Davide Varricchio

September 2024

## Contents

## 1 Introduction

While counting the number of distinct elements and computing the second moment of a stream are usually simple tasks that can be completed by keeping count of every element seen, there are cases in which it might not be feasible to store the necessary data structure in working memory. This could happen because of the high number of elements the stream is producing, or because multiple streams are being analyzed. In any cases, the major constraint is space. The goal of this project is to implement the Flajolet-Martin (FM) algorithm[2] to approximate the number of distinct elements, and the Alon-Matias-Szegedy (AMS) algorithm[1] to approximate moments of arbitrary order. These implementations will then be tested on the Letterboxd dataset, interpreting the set of movies as a stream.

## 2 Dataset

The implemented algorithms were executed on a particular part of the Letterboxd dataset published on Kaggle[1], accessed on September 4 2024. The dataset contains information about movies, organized as follows:

- A movies.csv file that stores each movie's id and its properties (such as the name or date).
- Several other csv files that represent 1 to many relationship, where the movie id is used as foreign key.
- Images of the posters of the movies.

The FM and AMS algorithm were applied on the actors.csv file, interpreting it as a stream of records. The record structure is (movie_id,name,role), effectively storing the name of the actor that played in the movie and its role in it. The name of the actor was the field chosen for the tasks of estimating distinct elements and estimating moments. No pre-processing was needed, since there were no null values.

## 3 Algorithms

### 3.1 Flajolet-Martin Algorithm

The Flajolet-Martin Algorithm estimates the number of distinct elements in the stream by hashing them. The idea is that the more different elements

[1] https://www.kaggle.com/datasets/gsimonx37/letterboxd/data

are seen in the stream, the more different hash values are seen. As more different hash values are computed, it becomes more likely that one of these values will be "unusual". The particular property the algorithm exploits is that the hash bit string ends in some number $r$ of consecutive zeroes. Given an element $a$ and its hash $h(a)$, this number $r$ is called the tail length for $a$ and $h$. Let $R$ be the maximum tail length of any $a$ seen so far in the stream. Then $2^R$ can be used as an estimate for the number of distinct elements seen in the stream. This estimate makes intuitive sense. The probability that a given stream element $a$ has $h(a)$ ending in at least r zeroes is $2^{-r}$. Supposing that there are $m$ distinct elements, the probability that none of them has tail length at least $r$ is $(1-2^{-r})^m$. It can be rewritten as $((1-2^{-r})^{2^r})^{m2^{-r}}$. The inner expression is of the form $(1-\epsilon)^{\frac{1}{\epsilon}}$, which is approximately $\frac{1}{e}$, assuming $r$ is reasonably large. Thus, the probability of not finding a stream element with as many as $r$ zeroes at the end of its hash value is $e^{-m2^{-r}}$. This means that:

- If $m$ is much larger than $2^r$, then the probability that a tail length of at least $r$ is found approaches 1.
- If $m$ is much less than $2^r$, then the probability that a tail length of at least $r$ is found approaches 0.

From these two facts, it can be concluded that he estimate $2^R$ (where $R$ is the maximum tail length ever recorded for a stream element) is unlikely to be either much too high or much too low. To get a more precise estimate, more hash functions can be used, and the results combined. The combining process still needs some attention. A simple mean could lead to a skewed result, because of its sensitiveness to outliers: even small variations in R (which are likely given the randomness of the algorithm) lead to very different results as the estimate is $2^R$. On the other hand, a median limits the estimate to be a power of 2. The chosen technique in the implementation involves dividing the estimates in small groups, taking the average of each group and returning the median, in order to increase the chances of approaching the true value of distinct elements.

### 3.1.1 Hash functions

The hashing algorithm used for implementation is MurmurHash3 (mmh3). It suits the use case as it is a fast algorithm (non cryptographic) that has all the properties on which the FM technique relies on: it is deterministic and produces a uniform distribution of the hashes. More importantly, it can be initialized with a seed, which changes the output values. This is convenient as the algorithm can be used with different seeds to simulate different hash functions. The basic idea of MurmurHash3 is to take an input, split it into chunks, mix those chunks using bitwise operations (like shifts, rotations, and XOR), and combine them into a final hash value. The implementation of this hashing algorithm is delegated to the mmh3 python package, which provides fast computation thanks to the C implementation. The package lets also the user choose between 32 and 128 bit hashes (it also provides 64 bit hashes, but uses the 128 bit code as backend). Since the FM algorithm will be used to estimate the number of distinct actors in the Letterboxd dataset, the length of the hash values was chosen to be 32 bit, as it is more than enough to represent the true number, while being faster to compute. For reproducibility purposes, the FM implementation uses as seeds the index of the hash function (e.g. hash function 0 seed = 0, hash function 1 seed = 1 ...).

#### 3.1.2 Space Requirements

During the algorithm execution, it is not required to store the elements seen. All that is needed is a single integer for each hash function, that records the maximum tail length ever seen, effectively enabling the solution to scale really well with the size of the stream. In practice, as more and more hash function are used, the bottleneck quickly becomes the computation time of such functions.

## 3.2 Alon-Matias-Szegedy Algorithm

The $kth$ order moment is computed as $\sum_i (m_i)^k$ where $m_i$ is the frequency of element $i$ in the stream. The 0th moment is the number of distinct element and can be estimated with the FM algorithm, while the 1st moment is the length of the stream, which is trivial to compute. The AMS algorithm estimates moments of order bigger or equal than 2 by computing some number of random variables. Each variable X is composed by:

- X.element, a particular element of the stream.
- X.value, an integer that counts the absolute frequency of X.element starting from some position in the stream.

To determine X.element, a position in the stream between 1 and its length is chosen with uniform probability, and the element found there is stored. X.value will store the absolute frequency of X.element starting from that same position. The $kth$ order moment can be estimated from any variable by computing $n(v^k - (v-1)^k)$ where $n$ is the length of the stream and $v$ is X.value. This makes sense because of the expected value of the expression. Let $e(i)$ be the stream element that appears at position i in the stream, and let $c(i)$ be the number of times element $e(i)$ appears in the stream among positions i, i + 1, ... , n.

$$E(n(v^k - (v-1)^k)) = \sum_{i=1}^{n} c(i)^k - (c(i)-1)^k$$

To make sense of this formula, it is useful to change the order of the summation by grouping over all the position that have the same element $a$. For instance, concentrate on some element a that appears $m_a$ times in the stream. The term for the last position in which $a$ appears must be $1^k = 1$. The term for the next-to-last position in which $a$ appears is $2^k - 1^k$, and so on up to $(m_a)^k - (m_a - 1)^k$. The formula for the expected value can than be rewritten as:

$$E(n(v^k - (v-1)^k)) = \sum_a (1^k + 2^k - 1^k + 3^k - 2^k + ... + (m_a)^k - (m_a - 1)^k)$$

It is easy to see and to prove by induction that the formula simplifies to

$$E(n(v^k - (v-1)^k)) = \sum_a (m_a)^k$$

This means that the expected value of $n(v^k - (v-1)^k)$ for every variable is exactly the $kth$ order moment. Because of that, the final estimate can be computed as an average of every variable's estimate, yielding better results.

### 3.2.1 Space requirements and dealing with infinite streams

The AMS algorithm requires to pick a position in the stream with uniform probability. To achieve space efficiency and to not store the entire stream, the implementation must pick the positions as they arrive. The technique used is called reservoir sampling. Supposing to have $s$ variables, the first $s$ positions are picked, providing a base case for the following inductive reasoning. Inductively, suppose $n$ stream elements were seen, and the probability of any particular position being the position of a variable is uniform, that is $s/n$. When the $n+1$ element arrives, that position is picked with probability $s/(n+1)$. If picked, one of the current $s$ variables is discarded with equal probability and it is replaced by a new variable whose element is the one at position $n+1$ and whose value is 1. If not picked, nothing changes and the $s$ variables keep their positions. To ensure the AMS algorithm correctness, in this new situation every position needs to have a probability of $s/n + 1$ of being selected. This is surely the case for position $n + 1$ by definition of the procedure, but it is also true for the probability of the first $n$ positions. Said probability is, according to the total probability law, equal to the probability of picking the first $n$ position when the $n + 1$ position is picked, plus the probability of taking the first $n$ position when the $n + 1$ position is not picked.

$$(1 - \frac{s}{n+1})\frac{s}{n} + (\frac{s}{n+1})(1 - \frac{1}{s})(\frac{s}{n})$$

In the above formula the inductive hypothesis was used to determine the probability of picking the first $n$ positions $(s/n)$. The formula simplifies to:

$$(1 - \frac{s}{n+1})\frac{s}{n} + (\frac{s-1}{n+1})(\frac{s}{n})$$

$$((1 - \frac{s}{n+1}) + (\frac{s-1}{n+1}))(\frac{s}{n})$$

$$(\frac{n}{n+1})(\frac{s}{n}) = \frac{s}{n+1}$$

Thus, showing that all the first n positions have the correct probability of being picked $(s/n+1)$ when the new element arrives. The reservoir sampling allows to store only the desired number of $s$ variables, effectively needing space for only $s$ integers and $s$ elements (which in this project's case are strings of tens of bytes), enabling the solution to scale well in the context of big data streams.

## 4 Experiments

The algorithms described in section 3 require the user to input as a parameter the desired number of hash functions/variables. Because of this, the main experiment consist of testing different values for these parameter on the data described in section 2. Different estimators were initialized with the following number of variables/hash functions: 1,16,32,64,128,256,512,1024,2048. As every actor name is seen, the variables/tail lengths are updated. Every 5000 elements, an estimate is computed for each estimator (For the AMS algorithm, only 2nd moment is estimated at this stage), as well as the true number of the 2nd moment/distinct elements for comparison. The FM algorithm implementation requires also an input parameter when computing an estimate, which is the size of the groups described in section 3.1. The estimates were computed using a group size of 4. More experiments were conducted on the estimators in their state at the end of the stream. First, the following different values of group sizes were tested for each distinct element estimator: 2, 4, 8, 16, 32,64. Then, the 2048 variables moment estimator was used to estimate the 3rd and 4th moment as a way to compare the error of the AMS algorithm with different order moments. Finally, the time to update the 2048 and 512 estimators (both FM and AMS) was computed, in order to have an idea of the frequency of the stream that could be managed with these algorithms.

### 4.1 Reproducibility

The experiments were executed both with the google colab standard free plan, and with an Intel Core i71165g7. In the former case, Only the first 100,000 rows of the actors.csv file were processed, to keep the execution time of the notebook under a reasonable limit (about 10 minutes). The AMS algorithm has some probabilistic behaviour when updating the variables stored. To ensure reproducibility, the seed 42 was used for the python pseudo-random number generator. Similarly, the FM implementation uses a hashing algorithm that can be initialized with a seed to simulate different hash functions. The seed used for each hash function is the index of such function as described at the end of subsection 3.1.1

## 5 Results

### 5.1 Flajolet-Martin implementation

As shown in figure 1, the error in the estimation of distinct elements goes up pretty much linearly as more elements are seen, regardless of how many hash functions are used. Another interesting thing is that there isn't some noticeable benefit in error reduction by using more than a certain number of hash functions, although the estimation curve gets less "bumpy" and follows better the growth of true distinct elements. Table 1 shows lower error for 1 and 16 hash functions, but this is probably due to the fact that the true distinct value at the last 5 5000 interval points is close to a power of 2. After that, there isn't noticeable improvement using more than 64 hash functions.

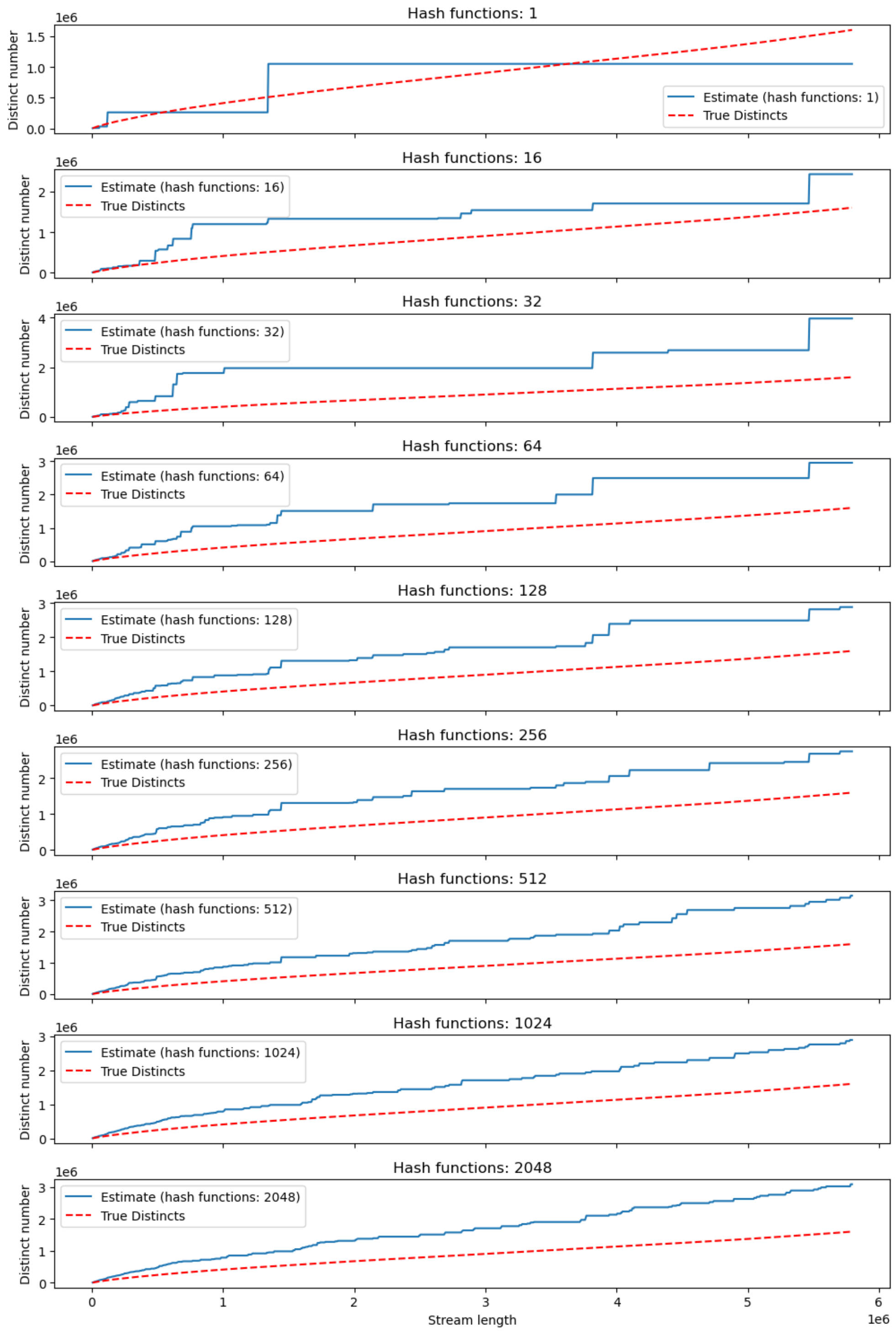

Figure 1: Distinct element estimation as more elements are seen, per hash function number

Table 1: Mean squared error of the last 5 distinct count estimates for each estimator

| hash functions | MSE |
|---|---|
| 1 | 299,275,518,122.0 |
| 16 | 687,587,800,029.2 |
| 32 | 5613,589,784,336.4 |
| 64 | 1831,948,701,046.8 |
| 128 | 1658,838,817,347.6 |
| 256 | 1338,388,853,725.2 |
| 512 | 2323,098,317,789.2 |
| 1024 | 1642,117,424,912.4 |
| 2048 | 2127,654,604,150.8 |

### 5.1.1 Group sizes

The plot 2 shows that there is not an improvement in increasing the group size. The best result is obtained with a group size of 2, while the book [4] suggests that a good number is the base 2 log of the expected number of distinct elements. This may happen because the true number of distinct actor names in the file is 1,600,665, which is pretty close to $2^{20}$ and $2^{21}$. Also, increasing the hash function number yields similar results (at least for group sizes less than 32). This is probably the effect of how the groups and the hash functions are computed. For instance, when 1024 hash functions are used, half of the seeds are repeated: seeds 0 to 1023 are used, but when using 512 hash functions the range of seeds is 0 to 511. Also, since the groups are computed by splitting the tail lengths list from left to right, also half of the groups will be the same. Finally, taking the median of the group averages will lead to similar results across the same group size, even with a different number hash function.

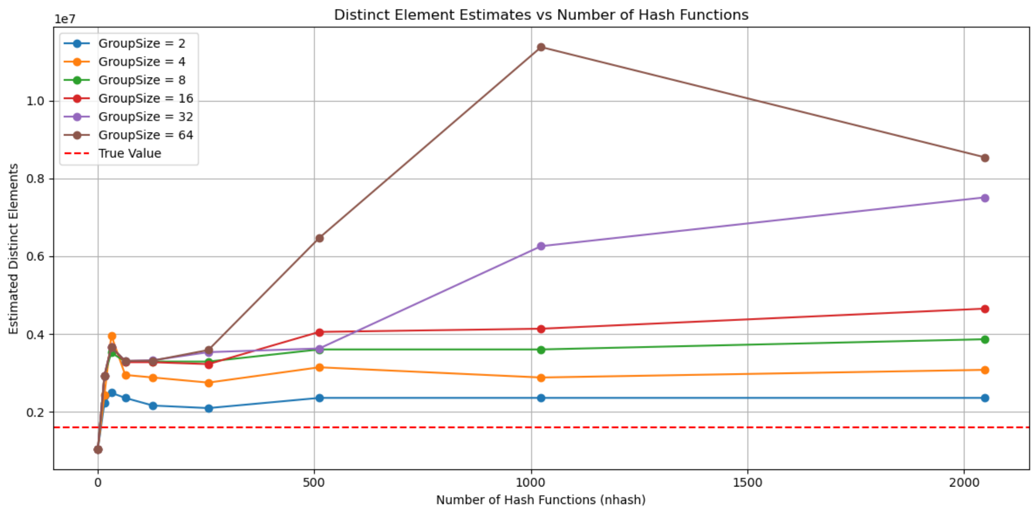

Figure 2: Distinct element estimation at the end of the stream with different group sizes, as hash function number increases

### 5.1.2 Computation time

The FM algorithm takes linear time in the number of hash functions to update the tail lengths when a new element is seen, as well as to compute an estimate. Despite this, using a lot of hash functions does not necessarily imply a better estimate, so one could use a small number of them and still have an acceptable result. While the following table shows execution times in the order of milliseconds, it is to be noted that the FM algorithm was run in a single thread, and more performance could be gained by computing the hash functions in parallel.

Table 2: Execution time to update the tail lengths when a new element is seen

| Hash functions | Colab | Intel i7 1165g7 |
|---|---|---|
| 512 | $\approx$ 1 ms | $\approx$ 0.2 ms |
| 2048 | $\approx$ 19 ms | $\approx$ 1 ms |

The execution times for the estimate computation is around the same order of magnitude.

## 5.2 Alon-Matias-Szegedy implementation

The figure 5.2 shows that the precision of the estimate grows as more variables are used, even though there are aren't noticeable improvements in using 2048 variables instead of 1024, as seen in table 3.

Table 3: Mean squared error of the last 5 2nd moment estimates for each estimator

| Variables | MSE |
|---|---|
| 1 | $\approx 3.21 \times 10^{16}$ |
| 16 | $\approx 2278 \times 10^{12}$ |
| 32 | $\approx 2598 \times 10^{12}$ |
| 64 | $\approx 222 \times 10^{12}$ |
| 128 | $\approx 4286 \times 10^{12}$ |
| 256 | $\approx 188 \times 10^{12}$ |
| 512 | $\approx 98 \times 10^{12}$ |
| 1024 | $\approx 6 \times 10^{12}$ |
| 2048 | $\approx 23 \times 10^{12}$ |

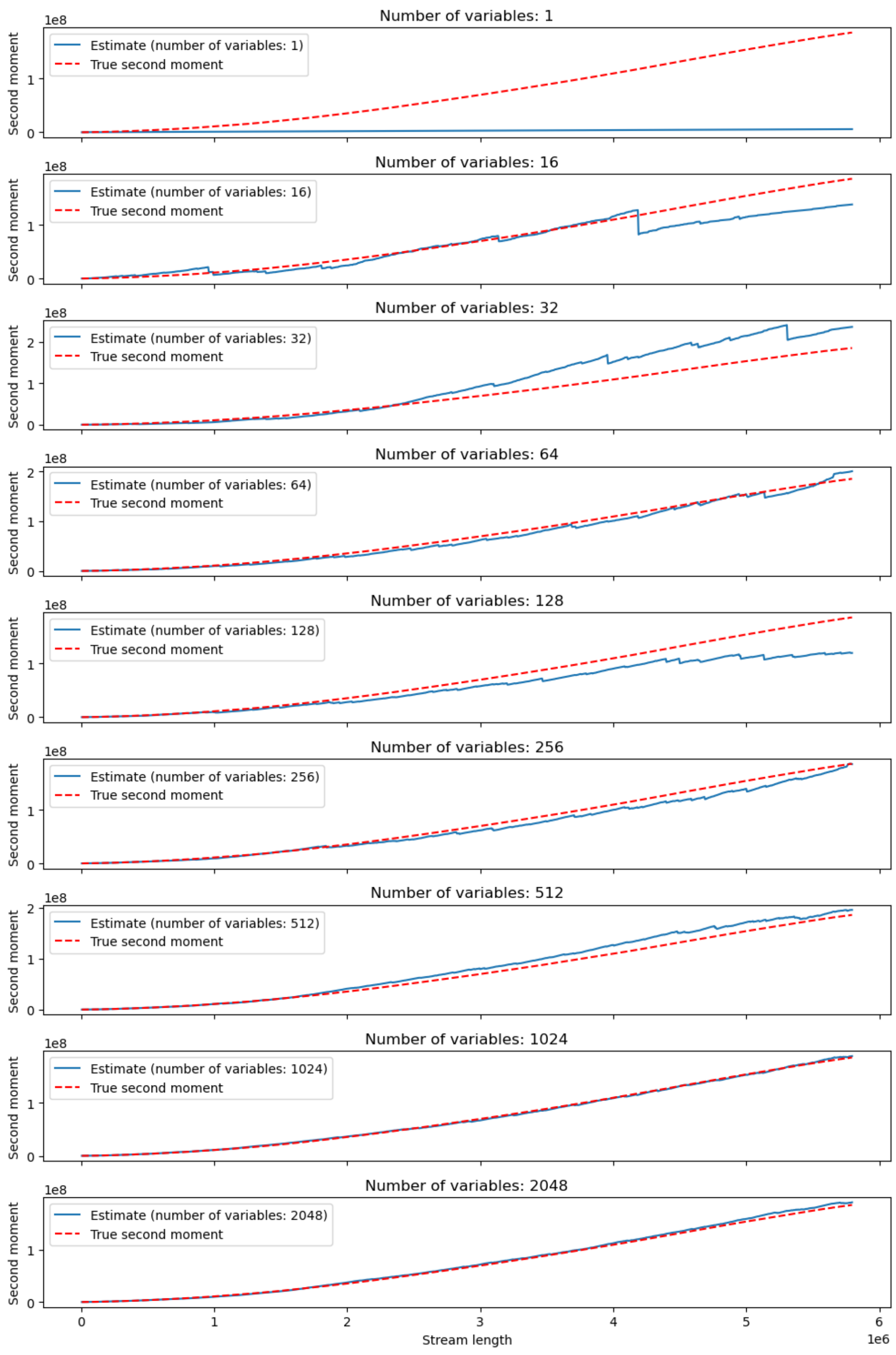

Figure 3: 2nd moment estimation as more element are seen, per number of variables used

### 5.2.1 Higher order moments

As shown in table 4, there is clearly an increase in error as the order of the moment that is being estimate increases. This result is also discussed in the original paper by Alon, Matias and Szegedy [1]

Table 4: 2nd, 3rd and 4th moment estimate vs true value, with 2048 variables, at the end of the stream

| Moment order | True | Estimate |
|---|---|---|
| 2 | 185,519,056 | 190,737,295 |
| 3 | 20,853,698,024 | 25,035,464,731 |
| 4 | 5,011,663,909,156 | 10,124,079,362,426 |

### 5.2.2 Computation time

Updating the variables when a new element arrives and computing an estimate takes linear time in the number of variables, but the estimate is still pretty accurate even with only 1024 variables. The execution times are in the range of the millisecond, but the same reasoning made for the FM implementation about parallelization can be applied.

Table 5: Execution time to update variables when a new element is seen

| Hash functions | Colab | Intel i7 1165g7 |
|---|---|---|
| 512 | $\approx$ 0.5 ms | $\approx$ 0.2 ms |
| 2048 | $\approx$ 1 ms | $\approx$ 0.8 ms |

## 6 Conclusions

In this work, the FM and AMS algorithm were implemented and tested with different parameters on the actors' names in the Letterboxd dataset. While the FM algorithm achieved looser estimates, the AMS one was pretty accurate, at least for the second moment. In both cases, the best estimates could be achieved with as little as 1024 hash functions/variables, effectively storing only tens of Kilobytes, instead of the possibly infinite stream.

## 6.1 Possible improvements

In the case of high frequency streams, the algorithms' execution time is crucial to avoid losing elements. The FM and AMS are pretty fast even if implemented in a single thread manner, but they are easily adaptable to a multi-thread environment: The hash functions' computation can be easily parallelized, as every tail length yields a different independent estimate. The AMS variables are also independent and the main bottleneck is updating each one of them, by increasing their value if the element seen is equal to the one stored in the variable.

While the FM algorithm managed to estimate well the order of magnitude of the number of distinct element, more precise approaches could be required depending on the use case. Different variations of the algorithms exist, as the HyperLogLog algorithm [3], which are more advanced and should give better results.

## References

[1] Noga Alon, Yossi Matias, and Mario Szegedy. "The space complexity of approximating the frequency moments". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing.* 1996, pp. 20–29.

[2] Philippe Flajolet and G Nigel Martin. "Probabilistic counting algorithms for data base applications". In: *Journal of computer and system sciences* 31.2 (1985), pp. 182–209.

[3] Philippe Flajolet et al. "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm". In: *Discrete mathematics & theoretical computer science* Proceedings (2007).

[4] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive data sets.* Cambridge university press, 2020.