Assignment 1: Streaming Algorithms

Objective

The objective of this assignment is to implement a novel streaming algorithm under the constraints of limited memory. In particular, this assignment serves as a practical exercise to approximate the distinct values, median and most frequent value in streaming data to understand the concepts in the course and their practical application.

In the assignment, a stream of integers representing yearly incomes of individuals (in 10s of thousands, e.g., 1 represents $10,000; 234 represents $2,340,000) is fed. The goal is to summarize the stream in three ways:

a) the approximate distinct number of incomes seen,
b) the median income, and
c) the most frequent value of income.

Data

Two versions of the data are prepared and provided for the assignment -

(1) trial_incomes.csv: a small trial version with only 1000 integers to use while developing the methods, and
(2) test_incomes.csv.zip: a test that goes over 1 million integers to test the implemented methods on a larger dataset.

Starter code

Additionally, a starting template code is provided. Python version 3.7 or later would work with the starter code. It includes all the data science, machine learning, or statistics libraries that are necessary.

The starter code reads the input file and set limits on memory (to constrain working with data as a stream). It iterates over elements in the stream and calls each of the three methods passing the value along.

a) Task 1A: "def task1ADistinctValues",
b) Task 1B: "def task1BMedian",
c) Task 1C: "def task1CMostFreqValue"

Purpose of the assignment is to implement above three methods such that they return the approximate value, and only use the 100 elements array (technically, a deque object with a maxsize but it operates as an array) provided to them as a memory. 100 elements array may only contain ints or floats as values.

During streaming, the current size of the array will be printed. It should remain < 8,000. At each of the following interactions: 10, 10^2, 10^3, ... , 10^6 it prints the current calculation from the method along with the memory usage.

Task 1A. Approximate the count of distinct incomes

Overview

Flajolet Martin Algorithm, also known as FM algorithm, is used to approximate the number of unique incomes in a data stream.

Pseudo Code of FM algorithm:

1. Selecting a hash function h so each element in the set is mapped to a string to at least log n bits.
2. For each element x, r(x) = length of trailing zeroes in h(x)
3. R = max(r(x))
4. Distinct elements = 2^R

FM algorithm is sensitive to the parameters of hash function and thus, the results from FM algorithm can vary significantly depending on the hash function values. Since it is allowed to store up to 100 elements at a time in memory as "memory1a" deque object, we will use 100 hash functions to optimize the accuracy of FM algorithm.

After which, we could use the mean of the results from 100 functions as an approximation for the count of distinct incomes. However, averaging would be susceptible to outliners. Therefore, we use the median of mean approach to average the estimates resulting from all hash functions.

Hash functions

First, we look at how to determine the hash function.

The hash function follows below pattern:

$$h(x) = ax + b \bmod c$$

where a is an odd numbers and c is the capping limit of hash range ($2^{\wedge k}$).

Explanation on choice of values for hash function

When a is even and b is odd, the hash function always returns odd numbers which causes the trailing zeros to be 0 all the time. When a is even and b is even, the hash function could return the same value for two different inputs of x. Hence, we would use odd numbers for a. For c, we will keep 2^{64} as a constant value; which would be more than sufficient to cover the maximum income data.

In order to generate the values for a and b, we use 'i' which has a value from 0 to 99 (as 'i' will a dynamic value and would be keeping the number of loop when looping through 100 elements in "memory1a").

i	a = (2* i) + 1	b = i	H(x) = ax + b mod c
0	1	0	1x + 0 mod c
1	3	1	3x + 1 mod c
2	5	2	5x + 2 mod c

3	7	3	7x + 3 mod c
4	9	4	9x + 4 mod c
5	11	5	11x + 5 mod c
7	15	7	15x + 7 mod c
8	17	8	17x + 8 mod c
9	19	9	19x + 9 mod c
10	21	10	21x + 10 mod c
..	..	..	..
..	..	..	..
20	41	20	41x + 20 mod c
30	61	30	61x + 30 mod c
40	81	40	81x + 40 mod c
50	101	50	101x + 50 mod c
60	121	60	121x + 60 mod c
70	141	70	141x + 70 mod c
80	161	80	161x + 80 mod c
90	181	90	181x + 90 mod c
99	199	99	199x + 99 mod c

Table 1: Illustrations of 100 Hash Functions

Memory Utilization

Every income value will pass through 100 hash functions, giving 100 trailing zeros. Current 100 trailing zeros are compared with previous 100 trailing zeros to determine the higher number of trailing zeros. Then, the higher number of trailing zeros are stored in "memory1a". Therefore, "memory1a" store 100 highest number of trailing zeros from 100 hash functions on the data seen so far.

Approximating count of distinct incomes

Every time the function is called to return the approximate count of distinct incomes, 100 highest number of trailing zeros in "memory1a" are bucketed into 10 groups to get 10 means and then, the median of these 10 means is returned as the approximate count of distinct incomes.

Summary

In summary, the followings have been performed to improve the accuracy of the algorithm:

1) **100 hash function to get 100 highest trailing zeros from each function** (maximum elements we can store in the memory is 100)
2) **Means by Bucketing** (100 highest trailing zeros in memory are split into 10 buckets with 10 elements in each bucket, and then we calculate the means from each bucket, resulting in 10 means)
3) **Median of means** (10 means are sorted into an ascending order and the median of means is picked as the approximate value for the count of distinct incomes)

Results

Running the Task 1A with trial_income.csv gives the following result:

Stream element at	Current memory size of memory1a	Distinct values
10	4480	16
10^2	3952	34
10^3	4480	111

Table 2: Results of Task1a from trial_Income.csv

Running the Task 1A with test_income.csv gives the following result:

Stream element at	Current memory size of memory1a	Distinct values
10	4480	10
10^2	3952	35
10^3	3952	128
10^4	3952	494
10^5	3952	2272
10^6	3952	8192
10^7	3952	30573

Table 3: Results of Task1a from test_Income.csv

Conclusion

Cross checking the results for trial_income.csv ensures that as the data becomes larger, the more accurate the result become. Therefore, we can conclude that FM algorithm is useful is approximating the distinct value from streaming data.

Stream element at	Distinct values from program	Expected Actual Distinct values	Over-estimation by
10	16	10	0.60
10^2	34	24	0.41
10^3	111	79	0.41

Table 4: Comparison of actual output vs expected output for trial_income.csv

Task 1B. Approximate the median of the incomes

Overview

Assuming that income dataset follows a Pareto type 1 distribution, we use the following function to approximate the median of the incomes.

$$\hat{\alpha} = \frac{n}{\sum_{i=1}^{n} \ln x_i}$$

which was derived from maximum likelihood estimation. In the equation, n stands for total number of elements seen and x is the element at a time.

Memory Utilization

We store two values in the memory “memory1b”

(1) n = to count the number of income records processed
(2) s = summation of ln (income) records seen so far

Approximating the median of the incomes

When the function is called to return the median of the incomes seen so far, we simply plug in the count “n” and summation “s” into the equation and return the answer.

Results

Running the Task 1C with trial_income.csv gives the following result:

Stream element at	Current memory size of memory1c	Median
10	2780	0.51
10^2	2780	0.74
10^3	2780	0.74

Table 5: Results of Task1b from trial_Income.csv

Running the Task 1C with test_income.csv gives the following result:

Stream element at	Current memory size of memory1c	Median
10	2780	0.48
10^2	2780	0.75
10^3	2780	0.76
10^4	2780	0.75
10^5	2780	0.75

10^6	2780	0.74
10^7	2780	0.74

Table 6: Results of Task1b from test_Income.csv

Task 1C. Approximate the mode of the incomes

Proposed Algorithm

With the allowance of 100 elements in the memory "memory1c", in order to find the most frequently seen income value (mode) thus far in the stream, we could use the following algorithms

<u>Pseudo Code step-wise solution:</u>

1. Store the income value and its seen count in the memory. Since we have 100 elements/slots in the memory, we will have maximum 50 pairs of income and its seen count.

Position	0	1	2	3	4	5	…	…	98	99
Value	10	1	34	1	2	4	…	…	…	…

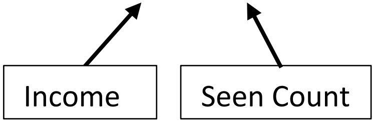


Figure 1C.1: Illustration of 100 elements in the memory

2. For each income value, check if the income already exists in the memory. If income already exists in the memory, simply increase its counter and decrease all the other counters. Remove any pairs with seen count value as zero.

Income value = 2
Before:

Position	0	1	2	3	4	5	…	…	98	99
Value	10	1	34	1	2	4	…	…	…	…

After:

Position	0	1	2	3	4	5	…	…	98	99
Value	10	0	34	0	2	5	…	…	…	…

Seen Count decreased by 1
Seen Count increased by 1

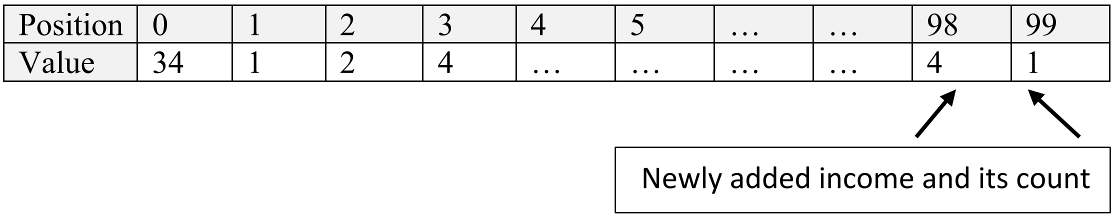
Figure 1C.2: Illustration of updating the income pair in the memory

3. If the income does not exists in the memory, add the income and its count as '1' to the memory.

Income value = 4
Before:

Position	0	1	2	3	4	5	…	…	98	99
Value	10	1	34	1	2	4	…	…	…	…

After:



Position	0	1	2	3	4	5	…	…	98	99
Value	34	1	2	4	…	…	…	…	4	1

Figure 1C.3: Illustration of adding a new income pair to the memory

Conditions of adding a new pair to the memory
Every time a new value pair is added to the memory, we would need to remove one of the existing pairs in the memory provided that the memory does not have any empty slots. To improve the accuracy, we would lay out conditions on where in the memory a new pair should be added:

- When there is an empty slot in the memory, it should be added in the empty slot
- Otherwise, we find an income pair whose seen count is the lowest minimum value in the memory and replace it with a new pair

4. Mode = the income with the highest seen count in the memory

Memory Utilization

Because of the memory space constraint (an array with 100 elements), we can only store up to 50 pairs to approximate the frequently seen income. Therefore, we set the criteria to remove or replace the values from the memory when following conditions are met.

- Once the seen count value is zero, we remove the income and its seen count from the memory, thereby, making space for new pairs to be stored in the memory.

- When there is a new income value to be added to the memory, we find the income with the least last seen value and replace it with the new income value.

Note: Every time an income record is processed, we decrease the seen count of income and its seen count in the memory except for the income we are processing. Eventually, some of the seen count value will become zeros.

Approximating the mode of the incomes

Every time the function is called to return the approximate mode of the incomes seen so far, we return the income value with the highest seen count from the memory.

Results

Running the Task 1C with trial_income.csv gives the following result:

Stream element at	Current memory size of memory1c	Mode
10	2992	2
10^2	2800	4
10^3	2824	4

Table 8: Results of Task1c from trial_Income.csv

Running the Task 1C with test_income.csv gives the following result:

Stream element at	Current memory size of memory1c	Mode
10	3328	3
10^2	2800	3
10^3	3376	3
10^4	3376	3
10^5	3352	1
10^6	2848	3
10^7	3376	3

Table 9: Results of Task1c from test_Income.csv

Limitations and improvement

If the mode of the dataset appears too far apart in the data i.e. it is seen after every 70 records or so, our proposed solution would not return the accurate result as the value mode would have already been removed from the memory. This must be the reason why the mode is returning the mode value as "1" for stream element at 10^5. However, it would mostly be able to approximate the mode for dataset with Pareto distribution since in Pareto distribution, the data is heavily tailed on one end and therefore, we can mostly assure that appearance of the mode value would not be as disperse as in that of normal distribution.

Currently, the program is returning the first element found in the memory when there are no modes in the dataset. This could be corrected by having a rule stating if all the elements have last seen count as 1, the mode return would be none.

Conclusion

Validating the results for trial_income.csv ensure that as the data becomes larger, the more accurate the result becomes. Therefore, we can conclude that above proposed solution can approximate the mode of the dataset with Pareto distribution. However, a more thorough experiments or simulations would be required to prove that it will also work with a large dataset and see how much deviation it has from the actual results.

Stream element at	Modes from program	Expected Actual Mode	Error rate
10	2	None	100%
10^2	4	4	0
10^3	4	4	0

Table 10: Comparison of actual output vs expected output for trial_income.csv