Algorithms for Data Science: Data Streams

Pablo Mollá Chárlez

September 29, 2024

Contents

1 Introduction to Data Streams

Data streams are sequences of data that arrive continuously and in-real time, requiring immediate processing or analysis. Unlike traditional databases that assume datasets are available in their entirety (offline), data streams are often available only online, meaning the data arrives incrementally and is not stored permanently. This is common in scenarios like:

- **Twitter status update**: New tweets are constantly being posted in real-time.
- **Queries on search engines**: Devices submit new queries continuously.
- **Data from sensor networks**: Devices such as weather sensors or IoT devices generate real-time data streams.
- **Telephone calls of IP packets on the Internet**: Both involve data that must be processed instantly.
- **High-speed trading data**: Financial markets generate streams of transactions data at a rapid pace.

1.1 Data Stream Model

A stream of data can be represented as an infinite sequence of items $S = \{i_1, i_2, \ldots, i_k, \ldots\}$. Data streams are characterized by being:

- **Infinite**: The streams can grow indefinitely as new data arrives.
- **Non-stationary**: The statistical properties of the stream can change over time.

The primary challenge with data streams is how to ask queries on the stream in 2 forms:

1. **Standing Queries**: The continuously monitor the stream for certain conditions.
2. **Ad-hoc Queries**: These are made on-demand for specific information about the stream.

However, some restrictions are forced:

- **Storage Space**: It's impossible to store all items in an infinite stream.
- **Processing time**: The stream must be processed in real-time; once data is passed, it's lost forever unless processed immediately.

The difference between sampling items (Reservoir Sampling) and filtering items (Bloom Filter) from a stream lies in purpose, functionality, and the nature of how they handle elements in the stream:

- **Reservoir Sampling**: The goal of reservoir sampling is to select a random sample of size s s from an infinite or unknown-sized stream of elements. It ensures that each element in the stream has an equal probability of being included in the final sample. It's used when you want to maintain a representative sample from a large stream with memory constraints.
- **Bloom Filter**: A Bloom Filter is used for efficient membership testing. It answers the question: "Have I seen this item before?" by checking if an element is possibly in a set (but with false positives) or definitely not. It helps in filtering out elements that aren't in a predefined set with very low memory requirements. It is primarily used to prevent reprocessing or duplicate detection

2 Sampling Items from a Stream

Objective: The goal is to keep a proportion **p** of the items in a stream. For instance, we might want to keep 1 in every 10 elements from a stream. Let's consider that we are working with queries on search engines.

- **First Solution**: One straightforward approach is to randomly decide for each element whether to keep it. For instance, we generate a random number between 0 and 9 for each item, and we only keep the item if we generate a 0. However, this clearly leads to an inaccurate estimation, as it doesn't correctly capture the true proportion of singletons or duplicates (of queries) in the stream. (Proof on Slides)
- **Better Solution**: To improve the accuracy, it's better to sample based on users rather than individual queries. That way, for a given sampled user, we would retain all of their queries, ensuring a more accurate representation of their behavior.

  This can be done by hashing user identifiers to integers and using the hash value to decide which users to sample. Now, assume we need to keep a sample of exactly s items (due to memory limitations). The goal is that each item in the stream should be in the sample with equal probability.

Let's see an example to understand it.

2.1 Real-World Scenario

Let's consider a real-world scenario where we are analyzing search queries submitted by users to a search engine. The stream consists of tuples in the form (user, query, timestamp), representing each user's query and the time it was made. We want to sample a proportion of users to retain all their queries, instead of sampling individual queries. Imagine the following sequence of tuples arriving in the stream:

$$\begin{aligned}(UserA, "howtobakeacake", 09:00)\\(UserB, "bestpizzarecipe", 09:01)\\(UserA, "chocolatecakeingredients", 09:02)\\(UserC, "weatherinNewYork", 09:03)\\(UserB, "howtomakepizzadough", 09:04)\\(UserA, "cakefrostingideas", 09:05)\\(UserC, "NYweatherforecast", 09:06)\end{aligned}$$

Instead of sampling queries, we aim to sample users, so that all queries from a sampled user are kept. This can be done by hashing user identifiers (like usernames or user IDs) into integers and using the hash value to decide whether to keep the user.

Steps:

1. $\boxed{\text{Hash user identifiers}}$: We will apply a hash function to each user (e.g., hash(user) mod 100).

2. $\boxed{\text{Set sampling threshold}}$: Let's say we want to sample 10% of the users. We retain a user if the hash of their identifier is less than 10 (out of 100).

   - For example, hash(UserA) mod 100 = 5, hash(UserB) mod 100 = 35, and hash(UserC) mod 100 = 15.

3. $\boxed{\text{Sampling logic}}$: Based on the hash values:

   - UserA is sampled because $5 < 10$. Therefore, all of UserA's queries will be kept in the sample.
   - UserB is not sampled because $35 \geq 10$. Hence, none of UserB's queries will be included.
   - UserC is not sampled because $15 \geq 10$. None of UserC's queries will be included either.

4. $\boxed{\text{Resulting Sample}}$: After processing the stream, the only sampled user is UserA, so the sample would look like this:

$$\begin{aligned}(UserA, "howtobakeacake", 09:00)\\(UserA, "chocolatecakeingredients", 09:02)\\(UserA, "cakefrostingideas", 09:05)\end{aligned}$$

The advantages of such procedure include:

1. **Complete user behavior**: By sampling users instead of individual queries, we capture all the queries from a sampled user, allowing for more accurate analysis of user behavior patterns.

2. **Equal probability**: Each user is sampled independently with equal probability (based on the hash function), ensuring fairness.

3. **Efficient memory usage**: We can control the memory usage by adjusting the sample size based on user sampling, rather than tracking individual queries.

2.2 Reservoir Sampling Algorithm

The Reservoir Sampling Algorithm is a popular approach to maintain a random sample of s elements from a stream, ensuring each element is included with equal probability.

2.2.1 Algorithm

1. **Initialization**: Store the first s elements from the stream.
2. **Selection**: For each new element n (where $n > s$):
   - With probability $\frac{s}{n}$, keep the new element, otherwise discard it.
   - If the element is kept, replace a randomly chosen element in the current sample.

2.2.2 Proof

Let's prove it by induction by claiming as follows: The algorithm ensures that after processing n items, each element has been included in the sample with probability $\frac{s}{n}$.

- $\boxed{\text{Base case}}$: After seeing the first s elements, they are in the sample with probability $s/s = 1$. All original elements are included in the reservoir sample.
- $\boxed{\text{Inductive hypothesis}}$: After processing n elements, each element is in the sample with probability $\frac{s}{n}$.

  For the inductive step, when element $n+1$ arrives, the probability that the new element is kept is $\frac{s}{n+1}$. We know that the probability of being in the reservoir sample, under the inductive hypothesis, is $\frac{s}{n}$ and the probability of leaving due to the new arrival of the element $n+1$ is $\frac{n}{n+1}$, therefore the probability that any previously sampled element is retained in the sample at time $n+1$ is:

$$\frac{s}{n} \times \frac{n}{n+1} = \frac{s}{n+1}.$$

  This ensures that at each step, every element has an equal chance of being in the sample, which guarantees the desired property of the algorithm.

2.2.3 Python Implementation

```python
import random

def selectKItems(stream, n, k):
    # Initialize reservoir with the first k elements
    reservoir = stream[:k]

    # Replace elements with a decreasing probability
    for i in range(k, n):
        j = random.randrange(i + 1)
        if j < k:
            reservoir[j] = stream[i]

    return reservoir

# Driver Code
if __name__ == "__main__":
    stream = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
    k = 5
    result = selectKItems(stream, len(stream), k)
    print("Randomly selected items:", result)
```

3 Filtering Items from a Stream

A Bloom Filter is a probabilistic data structure used for efficiently checking whether an element might belong to a set or is definitely not in the set. It uses hashing and a fixed-size bit array to achieve memory-efficient membership tests.

However, a Bloom Filter can have false positives (incorrectly indicating whether an element is in the set) but guarantees no false negatives (if it says an element is not in the set, it is definitely not).

3.1 Algorithm

1. **Bit Array Initialization**

   - **Array** B: The Bloom Filter starts with a bit array B of size n bits, all initialized to 0.

   $$B = [0, 0, 0, 0, ..., 0] \quad (n \text{ bits})$$

   - **Hash Functions** $h_1, h_2, \dots, h_k$: A collection of k independent hash functions is used, each mapping elements to an index in the bit array B. Each hash function takes an input element and maps it to one of the n bit positions.

   $$h_i(x) : \text{element} \to \{0, 1, 2, ..., n-1\}$$

2. **Adding Elements to the Bloom Filter**: When adding an element (key) x to the Bloom Filter:

   - Each of the k hash functions is applied to x, producing k positions in the bit array B.
   - The bits at these positions are set to 1. If a bit is already set to 1 by a previous element, it remains 1.

   $$B[h_1(x)] = 1, \quad B[h_2(x)] = 1, \quad ..., \quad B[h_k(x)] = 1$$

   Let's say we are inserting an item x into a Bloom Filter with $n = 10$ bits and $k = 3$ hash functions. Assume the hash functions give the following indices for x:

   $$h_1(x) = 2$$
   $$h_2(x) = 4$$
   $$h_3(x) = 7$$

   After inserting x, the bit array looks like this:

   $$B = [0, 0, 1, 0, 1, 0, 0, 1, 0, 0]$$

3. **Checking Membership (Querying the Bloom Filter)**: To check if an element y is in the Bloom Filter:

   - Apply the same k hash functions to y, producing k bit positions.
   - Check the bits at those positions in the bit array:
     - If all the bits at the hashed positions are 1, the Bloom Filter returns "might be in the set" (there is a possibility the item is in the set, but there could be a false positive).
     - If any bit is 0, the Bloom Filter returns "definitely not in the set" (the item is not in the set).

Suppose we want to check if y is in the filter:

$$\begin{aligned} h_1(y) &= 3 \\ h_2(y) &= 7 \\ h_3(y) &= 9 \end{aligned}$$

We check bits 3, 7, and 9 in the bit array. If all are 1, we say "maybe in the set"; if any bit is 0, we say "definitely not."

Comment: Bloom Filters do not support deletion directly, once a bit is set to 1, it cannot be reset to 0, since this would affect other elements that also hash to that bit. There are variations like counting Bloom filters that can handle deletions by maintaining a count of how many elements have set each bit, but the basic Bloom Filter does not support this.

3.2 Probability of False Positives

A false positive occurs when the Bloom Filter incorrectly reports that an element might be in the set, even though it was never added. This happens because multiple elements can set the same bits in the bit array, causing hash collisions. The probability of a false positive P_{fp} is defined as follows:

$$\boxed{P_{fp} = \left(1 - e^{-\frac{km}{n}}\right)^k}$$

Where:

- P_{fp}: Probability of a false positive.
- m: Number of elements inserted into the Bloom Filter.
- n: Size of the bit array (number of bits).
- k: Number of hash functions.
- e: Euler's number (approximately 2.71828).

3.3 Optimal Number of Hash Functions

The optimal number of hash functions k that minimizes the probability of false positives is given by:

$$\boxed{\mathrm{k} = \tfrac{n}{m} \cdot \ln(2)}$$

Where:

- k: Optimal number of hash functions.
- n: Size of the bit array (number of bits).
- m: Number of elements inserted into the Bloom Filter.
- $\ln 2$: Natural logarithm of 2 (approximately 0.693).