"Online" Algorithms (Data Streams): Ideas and code

Pietro Colaguori 1936709

November 2023

1 Ideas

Online algorithms are algorithms that process data in a sequential order, making decisions without having the entire dataset available at once. In the context of data streams, where data is continuously generated and processed in real-time, online algorithms play a crucial role. These algorithms are designed to handle large datasets that cannot fit into memory and must be processed on the fly.

One common application of online algorithms is in the realm of data stream processing, where the algorithm must process data as it arrives and make decisions based on the current information. Online algorithms are memory-efficient and have a constant or logarithmic space complexity, making them suitable for handling massive streams of data.

I will proceed by discussing the concept of online algorithms in the context of data streams, later I will also provide snippets of code.

2 Code

1. **Number of Elements**: we keep track of all the different elements seen by using a hash table or a probabilistic data structure, like Count-Min Sketch. Below I will show an example script in which I implement this idea using the DS Count-Min Sketch, the output, as expected, will be "Estimated count of item 1: 3".

```python
from collections import defaultdict

class CountMinSketch:
    """
    Count-Min Sketch is a probabilistic data structure used for estimating the frequency of elements in a data stream.
    It uses a fixed amount of memory and provides approximate counts with a small error rate.
```

```python

    Attributes:
        width (int): The width of the sketch table.
        depth (int): The depth of the sketch table.
        sketch (list): The sketch table used for counting.

    Methods:
        hash_func(x, i): Hashes the element x using the i-th hash function.
        update(x): Updates the sketch table by incrementing the count for element x.
        estimate(x): Estimates the frequency of element x by finding the minimum count across all hash functions.

    Usage:
        sketch = CountMinSketch(width, depth)
        sketch.update(x)
        estimated_count = sketch.estimate(x)
    """

    def __init__(self, width, depth):
        """
        Initializes a CountMinSketch object.

        Args:
            width (int): The width of the sketch table.
            depth (int): The depth of the sketch table.
        """
        self.width = width
        self.depth = depth
        self.sketch = [[0] * width for _ in range(depth)]

    def hash_func(self, x, i):
        """
        Hashes the element x using the i-th hash function.

```

```python
        Args:
            x: The element to be hashed.
            i (int): The index of the hash function.

        Returns:
            int: The hashed value of x.
        """
        return hash((x, i)) % self.width

    def update(self, x):
        """
        Updates the sketch table by incrementing the count for element x.

        Args:
            x: The element to be updated.
        """
        for i in range(self.depth):
            j = self.hash_func(x, i)
            self.sketch[i][j] += 1

    def estimate(self, x):
        """
        Estimates the frequency of element x by finding the minimum count across all hash functions.

        Args:
            x: The element to be estimated.

        Returns:
            int: The estimated frequency of element x.
        """
        min_count = float('inf')
        for i in range(self.depth):
            j = self.hash_func(x, i)
            min_count = min(min_count, self.sketch[i][j])
        return min_count

# Example usage
stream = [1, 2, 3, 1, 2, 4, 5, 1, 2, 3]
cms = CountMinSketch(width=100, depth=5)
```

```python
    for item in stream:
        cms.update(item)
    count = cms.estimate(1)
    print(f"Estimated count of item 1: {count}")
```

In order to make the script as clear as possible I added varbose comments.

2. **Sliding Window**: Keep track of a fixed size window containing the latest elements of the data stream, this idea is very powerful and has many application, for instance in Telecommunications with the implementation of a packet-switched network, in which it is needed to keep track inside the buffers of the routers of a fixed size window of packets. To create an interesting script it would be better to implement some sort of data processing on the contents of the window, in this case I decided to compute the average value of the sliding window.

```python
class SlidingWindow:
    def __init__(self, window_size):
        self.window_size = window_size
        self.window = []

    def add_data(self, data):
        self.window.append(data)
        # Slide of one position if the window
           is full
        if len(self.window) >
           self.window_size:
            self.window.pop(0)

    def process_window(self):
        # Perform processing on the window
           data
        # Example: Calculate the average of
           the window
        window_sum = sum(self.window)
        window_avg = window_sum /
           len(self.window)
        return window_avg

# Example usage
window_size = 5
stream = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sliding_window = SlidingWindow(window_size)
```

```python

    for data in stream:
        sliding_window.add_data(data)
        window_avg = sliding_window.process_window()
        print(f"Window Average: {window_avg}")
```

Naturally, as easiliy verified, the output in this case will be:

```
    Window Average: 1.0
    Window Average: 1.5
    Window Average: 2.0
    Window Average: 2.5
    Window Average: 3.0
    Window Average: 4.0
    Window Average: 5.0
    Window Average: 6.0
    Window Average: 7.0
    Window Average: 8.0
```