SDSC3001 Course Project

A Memory-Efficient Sketch Method for Estimating High Similarities in Streaming Sets

Junyeong Heo (55960131)

Introduction

The paper I chose is titled 'A Memory-Efficient Sketch Method for Estimating High Similarities in Streaming Sets', proposing a new sketching approach called MaxLogHash. As Data streams have become ubiquitous across various domains, from financial transactions to Internet of Things (IoT) data, network traffic, call logs, and trajectory records today. These applications generate massive volumes of data, making it expensive to collect and store entire data streams, especially when computational and storage resources are limited. These issues lead to the development of memory-efficient methods for processing and analyzing streaming data.

A fundamental challenge in data analysis is computing set similarities, which is crucial for numerous machine learning applications and information retrieval. For instance, in spam filtering, we need to determine how similar a newly received email is to a dataset of known spam emails. This similarity analysis helps automatically identify and filter out unwanted emails. Similarly, this paper mentions that understanding user similarities through visited websites history or friend networks is essential for applications like link prediction and recommendation systems in computer networks and online social networks (OSNs).

The Jaccard similarity coefficient is defined as $\frac{|A \cap B|}{|A \cup B|}$ for two sets, A and B, is one of the most widely used metrics for measuring set similarity. While MinHash is a well-established technique for approximating Jaccard similarity and has been successfully applied in various applications such as similarity search, social network compression, and web spam detection, it faces significant challenges when dealing with streaming data. These applications often focus on identifying highly similar sets (i.e., similarities close to 1), requiring accurate and memory-efficient methods for estimating high similarity values in a streaming context.

Research Challenges

MinHash has emerged as a fundamental technique for estimating set similarities, effectively handling streaming data by computing sketches incrementally. However, its requirement of 32 or 64 bits per register poses significant memory constraints, particularly in large-scale applications where resource efficiency is crucial. This limitation has motivated the development of more memory-efficient variants, but each comes with challenges.

Two notable memory-efficient variations, b-bit MinHash, and Odd Sketch, have demonstrated promising results for static data but fail to handle streaming scenarios effectively. The core issue lies in their inability to compute sketches incrementally. Specifically, b-bit MinHash cannot maintain accurate lowest b bits of minimum hash values in a streaming context because $\min^{(b)}(\pi(A \cup v)) \neq \min\left(\min^{(b)}(\pi(A)), \pi^{(b)}(v)\right)$. Similarly, Odd Sketch's bitmap-based approach makes incremental updates impossible as new elements arrive in the stream, rendering it unsuitable for streaming applications.

Recent attempts have been made to address these limitations, such as HyperLogLog and HyperMinHash mentioned in the paper, and compared their result with MaxLogHash proposed in this paper. These attempts have introduced new complications while trying to balance memory efficiency, computational speed, and estimation accuracy. The paper presents and discusses a detailed comparison of MaxLogHash with these recent methods. However, this report does not cover the technical details of these approaches and their implementations but rather focuses on introducing the MaxLogHash sketching proposed in this paper.

MaxLogHash

MaxLogHash introduces the estimation of similarities between Jaccard and streaming sets through an efficient sketch method. The core idea lies in using the logarithm of hash values to reduce memory usage while maintaining accuracy significantly. Examining the key components of this method, MaxLogHash first defines a function h that maps any element v to a random number in range (0,1). We compute each element's log rank as $r(v) = \lfloor -\log_2 h(v) \rfloor$. The MaxLogHash sketch of a set A is then defined as $\text{MaxLog}(h(A)) = \max_{v \in A} r(v)$.

This approach offers three crucial advantages. First, the log-rank can be represented using only 7 bits with high probability, compared to the 32 or 64 bits required by MinHash. Second, the sketch can be computed incrementally for streaming data using $\text{MaxLog}(h(A \cup v)) = \max(\text{MaxLog}(h(A)), \lfloor -\log_2 h(v) \rfloor)$. Third, the Jaccard similarity can be estimated accurately using these compact sketches.

For implementation, MaxLogHash maintains k bit-strings for each set, where each bit-string contains two components: a register $m[i]$ recording the maximum hash value and a single bit $s[i]$ indicating whether exactly one element achieves this maximum value. The method provides theoretical error bounds for estimation accuracy, demonstrating that its error is nearly independent of set cardinality. Furthermore, an optimization called MaxLogOPH reduces the processing time from $O(k)$ to $O(1)$ per element while maintaining comparable accuracy for sufficiently large sets.

Experiment

To thoroughly evaluate the performance of MaxLogHash, the experiments were conducted on both synthetic and real-world datasets, allowing comprehensive assessment under

controlled and practical scenarios in this paper. The methods used to compare the performance of MaxLogHash, which uses *k* 6-bit are MinHash with *k* 32-bit registers per sketch, HyperLogLog, (originally designed for set cardinality estimation), with *k* 5-bit registers and HyperMinHash, a more recent approach that combines a HyperLogLog sketch (using q-bit registers) with r-bit registers for each of its k components. However, since this project is focused on reproducing the solution proposed in this paper, the experiments conduced are previously explain approaches, MinHash, b-bit MinHash, and Odd Sketch.

This paper explains the synthetic datasets were designed to test the algorithm's performance across different set characteristics. The experiments used 32-bit numbers as elements, creating two types of set-pairs: balanced and unbalanced. In balanced set pairs, both sets contained the same number of elements (10,000 by default), with Jaccard similarities ranging from 0.80 to 1.00. In unbalanced set pairs, the sets had different sizes, with one set's size determined by the target Jaccard similarity, testing the algorithm's robustness to size variations. These synthetic datasets enabled controlled algorithm performance testing across different similarity values and set size relationships. For the experiment, due to the limited computing capacity of personal device, true similarities raning from 0.80 to 1.00, stepping by 0.02. The metrics used are root mean squared error (RMSE) and bias, same as the paper, but 10 independents run with different random seeds, much less than 1,000 independent runs from the paper.

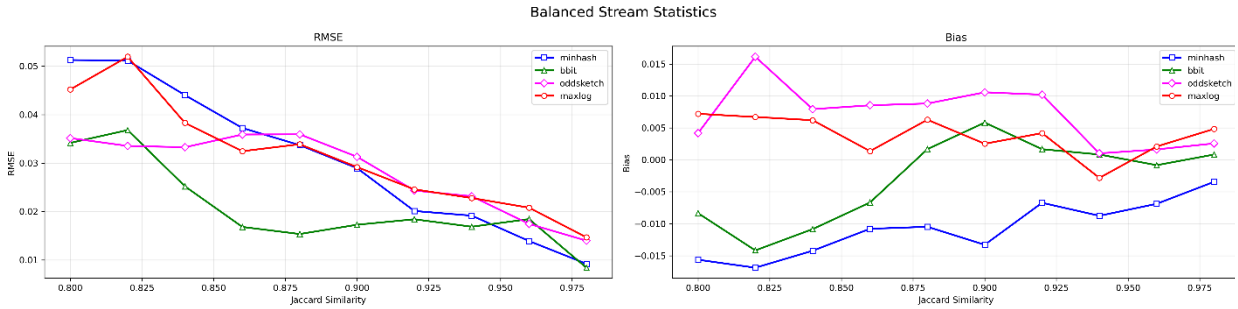

Figure 1. Balanced stream statistics

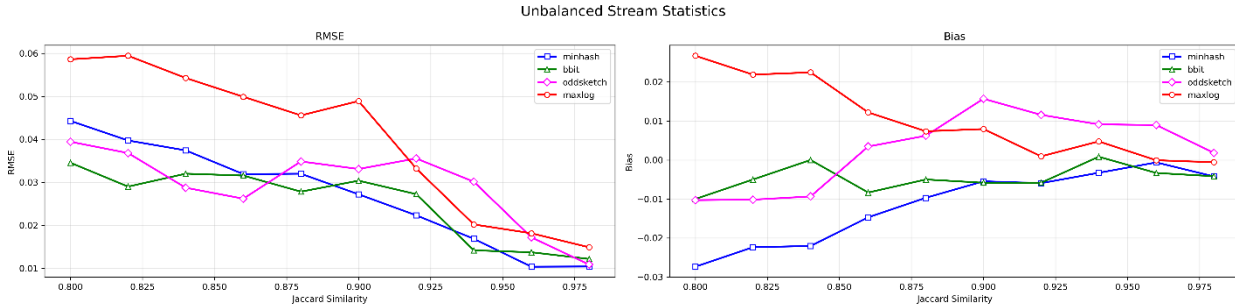

Figure 2. Unbalanced stream statistics

The result of balanced and unbalanced stream shows that the implementation of sketching method, especially the MinHash and MaxLogHash is successfully reproduced, showing the similar trend. Looking at both RMSE and Bias metrics, all methods demonstrate improved performance in terms of memory usage as introduced in the paper, presenting similar result of accuracy.

In the meantime, this paper used two real-world datasets, MUSHROOM and CONNECT, to evaluate the algorithm's practical performance. These datasets represent transaction records where items with uniform sizes for each transaction appear together. MUSHROOM contains 8,124 records with 119 distinct items, having 23 items for each record, and this results in 186,852 item-record pairs. CONNECT is larger, with 67,557 records with 43 items per record, 127 distinct items, and 2,904,951 item-record pairs.

Method	mean	median	std	rmse	bias
MinHash	0.292648	0.25	0.166855	0.058571	0.044202
b-bit	0.331533	0.291667	0.159565	0.03919	0.005318
Odd Sketch	0.344002	0.306853	0.171544	0.065526	-0.00715
MaxLog	0.393489	0.350132	0.149196	0.083587	-0.05664

For the experiment, only MUSHROOM data is analyzed randomly selecting 1000 records and compare the result of true similarity with the results of each sketch methods. The comparison of different sketching methods for the mushroom dataset reveals their relative performance in estimating Jaccard similarities. When analyzing the accuracy through Root Mean Square Error (RMSE), B-bit MinHash demonstrates superior performance with the lowest RMSE of 0.039, indicating it provides the most accurate estimates compared to true Jaccard similarity. This is followed by MinHash and OddSketch with moderate RMSE values of 0.059 and 0.066 respectively, while MaxLogHash shows the highest error with an RMSE of 0.084. Looking at systematic bias, B-bit MinHash again shows excellent performance with a near-zero bias of 0.005, suggesting balanced estimation. In contrast, MinHash tends to overestimate with a positive bias of 0.044, while MaxLogHash shows significant underestimation with a negative bias of -0.057. OddSketch maintains relatively neutral performance with a slight negative bias of -0.007. Overall, B-bit MinHash emerges as the most reliable method for this dataset, but it is not unacceptable result as the figure 2 from the original paper showed there is no significance difference between MinHash and MaxLogHash.

References

[1] Wang, Pinghui, Yiyan Qi, Yuanming Zhang, Qiaozhu Zhai, Chenxu Wang, John C. Lui, and Xiaohong Guan. "A Memory-Efficient Sketch Method for Estimating High Similarities in Streaming Sets." *ArXiv*, (2019). Accessed November 30, 2024. https://doi.org/10.1145/3292500.3330825.

[2] Qi, Y. (2019). qyy0180/MaxLogHash: Official implementation of "A Memory-Efficient Sketch Method for Estimating High Similarities in Streaming Sets". GitHub. https://github.com/qyy0180/MaxLogHash

[3] Shiroyama, H. (2012). hajimes/mmh3: Python wrapper for MurmurHash3, a fast non-cryptographic hash algorithm. GitHub. https://github.com/hajimes/mmh3