Learning-Augmented Frequency Estimation in Sliding Windows

Rana Shahout
Harvard University
rana@seas.harvard.edu

Ibrahim Sabek
University of Southern California
sabek@usc.edu

Michael Mitzenmacher
Harvard University
michaelm@eecs.harvard.edu

Abstract—We show how to utilize machine learning approaches to improve sliding window algorithms for approximate frequency estimation problems, under the "algorithms with predictions" framework. In this dynamic environment, previous learning-augmented algorithms are less effective, since properties in sliding window resolution can differ significantly from the properties of the entire stream. Our focus is on the benefits of predicting and filtering out items with large next arrival times – that is, there is a large gap until their next appearance – from the stream, which we show improves the memory-accuracy tradeoffs significantly. We provide theorems that provide insight into how and by how much our technique can improve the sliding window algorithm, as well as experimental results using real-world data sets. Our work demonstrates that predictors can be useful in the challenging sliding window setting.

I. INTRODUCTION

Stream processing plays a crucial role in various applications, such as network monitoring, intrusion detection systems, and sensor networks. Dealing with large and rapidly incoming data streams presents challenges in providing accurate responses to queries due to high computational and memory costs. Furthermore, these applications require time-efficient algorithms to cope with high-speed data streams. To that end, stream processing algorithms often build compact approximate sketches (synopses) of the input streams.

Estimating the frequency of certain items in a data stream is a fundamental step in data analysis. Several algorithms, such as Count-Min Sketch [1], Count Sketch [2] have been proposed to estimate item frequencies in the data streams.

As time passes, newer data often becomes more relevant than older data, necessitating an aging mechanism for the sketches. In financial analysis, for instance, analysts prioritize current market trends, whereas in intrusion detection systems, recent intrusion attempts are of primary concern. In both cases, outdated information loses significance over time. Retaining old data not only consumes valuable memory resources but also introduces noise, complicating the analysis of recent, relevant data. Many applications realize this by tracking the stream's items over a sliding window. The sliding window model [3] considers only a window of the most recent items in the stream, while older ones do not affect the quantity we wish to estimate. Indeed, standard approaches to the problem of maintaining different types of sliding window statistics have been extensively studied [3]–[7].

Recently, machine learning has been combined with traditional algorithms, yielding the paradigm of learning-augmented algorithms, also known as algorithms with predictions. This approach aims to improve algorithms by leveraging advice from machine learning models in the form of predictions. In the context of streaming algorithms, Hsu et al [8] introduced learning-based frequency estimation, where machine learning is utilized to predict the most frequent items, known as 'heavy hitters,' with the goal of reducing estimation errors. They utilized known hashing-based algorithms such as Count-Sketch and Count-Min Sketch, which approximately count item frequencies by hashing items to buckets. The learning process ensured that predicted heavy hitter items were not placed in the sketch but assigned dedicated counters, facilitating accurate counting. In addition, [9] described a learned-based algorithm for identifying heavy hitters, top k, and flow frequency estimation. They focus on the Space Saving algorithm and exclude predicted low-frequency items from updating the Space Saving, while ensuring that predicted heavy-hitter items remain in the structure. In this way, the Space Saving accuracy is improved.

However, these approaches do not directly translate to the sliding window model, where different items can become heavy hitters or have low frequency locally within a window as time passes. In other words, an item that is a heavy hitter for the entire stream may not be a heavy hitter in any particular data window. Moreover, to the best of our knowledge, predictions have not been applied to the problem of approximate counting in the sliding window setting. In general, sliding window variations of approximate counting and other problems are generally more challenging than variations without sliding windows, so perhaps this is unsurprising.

In this work, we aim to demonstrate the applicability of predictions to sliding window algorithms for frequency estimation problems, focusing on how natural predictions of the gap between item arrivals can lead to easy-to-implement improvements in this setting.

We focus on Window Compact Space-Saving (WCSS) [5] and present the learned-based version, LWCSS. Our primary idea is to exclude items that appear only once **in the sliding window** from being included in the data structure used for tracking item frequencies, inspired by LSS [9].

However, LSS employs predictors that relate to the entire

stream, which cannot be directly applied to a sliding window. An item with low frequency in the entire stream might not be low-frequency within a particular window, potentially leading to unbounded estimation errors.

Our key insight is the potential to exclude items that appear once by employing an effective predictor for each item's next arrival time in the stream. With a perfectly accurate predictor, we could identify single-occurrence items within the sliding window by determining whether they have not appeared previously in the stream, and have a predicted next appearance time that exceeds the current window size. Predictors of next arrivals have been employed in for example caching with machine learning advice [10].

We treat the next arrival predictor as a black box and do not delve into its internal functioning; our approach can therefore be utilized with any suitable learning scheme that produces a predictor. We analyze both theoretically and empirically the performance gains offered by predictors, as well as the robustness of our approach to predictor errors.

We also explore potential future directions, such as developing a predictor capable of learning item frequencies within a given time frame and periodically retraining itself using transfer learning techniques to adapt to shifts in the data distribution across frames. Furthermore, we discuss the prospect of extending our approach to handle other types of queries in the sliding window setting.

II. PROBLEM DEFINITION

Given a *universe* $\mathcal{U}$, a *stream* $\mathcal{S} = u_1, u_2, \ldots \in \mathcal{U}^N$ is a sequence of arrivals from the universe. (We assume the stream is finite here.) We denote by f_i^W the frequency of item i over the last W arrivals.

We seek algorithms that support the following operations:

- **ADD(i)**: given an element $i \in \mathcal{U}$, append i to $\mathcal{S}$.
- **QUERY(i)**: return an estimate $\widehat{f_i^W}$ of f_i^W

Definition 1. *An algorithm solves* (W, ε)-*WFrequency for if given any Query(i) it returns* $\widehat{f_i^W}$ *satisfying*

$$f_i^W \leq \widehat{f_i^W} \leq f_i^W + W\varepsilon.$$

III. SLIDING WINDOW ALGORITHMS

Sliding window algorithms have two conceptual steps: removing outdated data and recording incoming data. These algorithms aim to avoid storing the entire window sequence, as the window size is typically large. Due to such memory limitations, approximation algorithms are desirable and often employed. Such algorithms typically do not maintain individual counters for each item, but instead employ an approximate counting algorithm to monitor item frequencies.

In this work, we choose Window Compact Space Saving (WCSS) [5] as our underlying algorithm[1]. WCSS solves (W, ε)-WFrequency when the queries are frequency queries.

---

[1] Our approach could be applied to other sliding window streaming algorithms that follow the same setup of dividing the stream into frames (e.g. [11], [12])
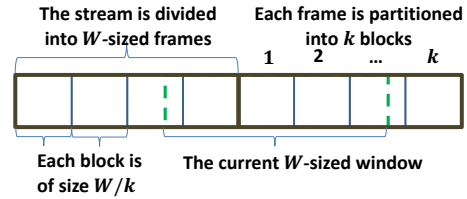
Fig. 1: WCSS algorithm overview (adapting a figure in [5])

It divides the stream into frames of size W and each frame is divided into k equal-sized blocks as illustrated in Figure 1. The query window also has size W and, importantly, it overlaps with no more than 2 frames at any given time.

WCSS counts how many times each item arrived during the last frame. When the counter exceeds a multiple of the block size ($\frac{W}{k}$, $\frac{2W}{k}$, etc), WCSS identifies it as an overflow. The algorithm keeps track of the item ID associated with each overflow, and selectively keeps only overflowed items for past blocks. To identify these overflowed items, WCSS uses an instance of the Space Saving algorithm (SS) [13] which is re-initialized as empty at the beginning of each frame. In addition, WCSS employs specific data structures to keep track of the overflowed item IDs, denoted by *overflowsRecord*.

In WCSS, *overflowsRecord* contains a collection of up to $k+1$ queues. Each queue corresponds to a block that overlaps with the current window. Within each queue, WCSS stores the IDs of items that have overflowed in the corresponding block. Whenever an item overflows, WCSS appends its ID to the current block's queue. When a block ends, WCSS removes the oldest queue from the collection. In order to estimate the frequency of an item within a given window, WCSS counts the overflow occurrences of the item and multiplies the result by $\frac{W}{k}$. Similar setups are found in works that support other queries over sliding windows, such as in [11]. In this work, *overflowsRecord* contains a hierarchical tree structure consisting of frequency tables. These tables store overflowed item IDs along with their respective frequencies within block intervals.

IV. LEARNED SLIDING WINDOW ALGORITHM

A. Overview

Given the specific characteristics of the sliding window setup, we want a predictor that offers a time-sensitive approach, in contrast to a predictor for whether an item is a heavy hitter over the stream, which is time-insensitive. We employ an oracle that, for a specific item i and timestamp t, predicts the next occurrence of i, or equivalently how many arrivals will occur before item i appears again. A detailed description of this predictor, outlining its structure and functionality, will be provided later in this section.

Based on this prediction, LWCSS excludes items predicted to next appear again beyond the frame size from being inserted in the SS instance. These items are referred to as *non-essential items*. As the SS instance is reset at the beginning of each frame, we obtain a "cleaner" SS instance, as proposed in [9].

Note that the *non-essential items* are not consistently ignored in every occurrence; rather we consider each arrival of the item and take an appropriate action based on the predictor. Note that if we had perfect predictions for non-essential items, then not including them in the Space Saving instance allows us to minimize memory overhead with no cost in terms of accuracy.

However, machine learning methodologies are inherently imperfect, and they may exhibit errors, including substantial and unexpected errors. Therefore, as suggested in the algorithms with predictions literature (see, e.g., [14]), algorithms based on machine learning predictions must demonstrate sufficient robustness to handle errors that may occur. In particular, the notion of robustness that has become common is that the performance of algorithms using predictions should not be significantly inferior to that of conventional online algorithms that do not rely on predictions, even if predictions are inaccurate. Unless some mitigating structure is added, ignoring items predicted as non-essential can lead to unbounded errors. For example, if a heavy hitter of the current frame is predicted incorrectly as a non-essential item, this item is excluded from the sliding window algorithm, violating the error guarantee (Definition 1).

We therefore apply the idea of keeping a Bloom filter of predicted non-essential items previously ignored during the current frame to ensure robustness, suggested by [9]. The Bloom filter, like the Space Saving instance, is flushed at the beginning of each frame.

There are two sources of estimation error when some arrivals of item i are predicted to exceed the window size. The first source of error arises when item arrivals are predicted incorrectly to exceed the window size. Ideally, these arrivals should have been captured by the Bloom filter and inserted back into the SS. However, when we insert item i into the Bloom filter for the first arrival that exceeds the window size, we do not add it to the SS. Thus, if the first arrival is mispredicted, we may miss one insertion (otherwise, it is correct not to insert this arrival into the SS). Second, the Bloom filter flushing at the beginning of a new frame causes an underestimation. Since the queried intervals can overlap with at most two frames, this imposes an underestimation error of no more than 1. To compensate for these two sources of error occurring together, we add 2 to the query.

Importantly, because the predictor is concerned with the item's next arrival, the number of *single occurrence items* within the frame exceeds the number of single occurrence items across the entire stream and depends on the frame size as shown in Figure 3.

B. Next occurrence oracle

In what follows, we use a slightly different approach that allows us to utilize a simpler predictor and still obtain strong performance. Rather than try to predict the next arrival exactly, we predict whether the next arrival is larger than W or not. If the next arrival is larger than W, then it clearly lies outside the frame (Figure 2). This allows us to have our predictor perform a binary classification, rather than solve a regression problem.
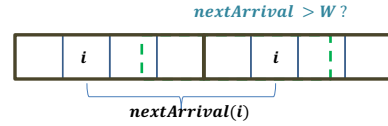

Fig. 2: Next arrival prediction in sliding window setting.
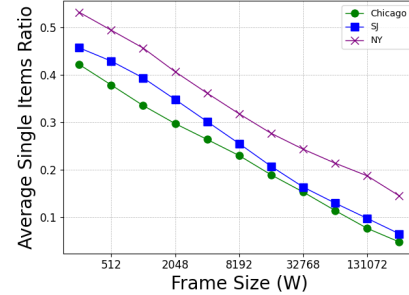

Fig. 3: Average single items ratio vs. frame size using real-world traces (described in Section V).

To construct the next occurrence oracle, we trained a neural network to predict if an item will show up again within a specified period of time (the next W items in the stream). By converting the problem into a two-category classification problem, we simplify the prediction problem. The oracle is thus tasked to classify whether the next appearance of the item falls within our specified window size, tagged as 1, or exceeds this limit, indicated as 0. Our model is a Recurrent Neural Network (RNN) that utilizes Long Short-Term Memory (LSTM) cell. We chose this framework for its effectiveness in processing sequential data as demonstrated in prior research [15]; however, we emphasize that other predictors could be used.

For our test cases, we focus on networking problems, and items are source-destination IP address pairs. The network begins by transforming the source and destination IP addresses into indices, which are then converted into embedding vectors. As a result of this process, IP addresses are represented in a dense and expressive manner. Our model uses these embeddings as input to the LSTM layer. In the LSTM layer, the sequence of IP address embeddings is processed, in order to capture temporal dependencies. After the LSTM produces the final hidden states, the results are passed to a fully-connected (dense) layer that outputs one value. This output value, after being passed through a sigmoid activation function, represents the model's prediction of whether the next appearance of an item falls within the specified window size W or not. This prediction is generated based on the sequence of IP addresses provided as input. The model is trained using Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss) and the Adam optimizer. After training, the model's performance is evaluated using the F1 score on a separate test dataset. Again, the implementation of such

a predictor is our tailored approach, and it is just one of many possible options. Depending on the requirements, our design may be replaced or enhanced with any other effective prediction technique.

C. Robustness Result

Consider a sliding window algorithm $\mathcal{A}$ that follows the same setup of dividing the stream into frames. The WCSS algorithm is an example of such an algorithm. In the following, we prove the robustness of our approach in the general case using algorithm $\mathcal{A}$. We show that our proposed algorithm is robust, in the sense that it cannot perform significantly worse than the corresponding algorithm that does not use predictions.

Theorem 1. *Let $\mathcal{A}$ be an algorithm for $(W, \varepsilon - \frac{2}{W})$-WFrequency. Then LWCSS solves (W, ε)-WFrequency.*

Proof. For any item i and any window of size W, we may underestimate the count of item i by at most two. This is because we may undercount an item once when it is predicted not to occur within the next W steps and it is not already a positive from the Bloom filter. As each window of length W can intersect two frames, and the Bloom filter is reset to empty at frame boundaries, we can undercount an item at most twice over any W consecutive steps.

We have

$$\widehat{Q_i^W} \triangleq \mathcal{A}_{(W, \varepsilon - \frac{2}{W})}.\text{QUERY}(i) + 2. \tag{1}$$

That is, our estimate is the query result for i from $\mathcal{A}$, which is an algorithm for $(W, \varepsilon - \frac{2}{W})$-WFrequency, with at most 2 instances of i removed from the stream $\mathcal{A}$ processes and an extra count of 2 added back in. It follows the smallest possible return value is

$$(Q_i^W - 2) + 2 = Q_i^W,$$

and the largest possible return value is

$$\left(Q_i^W + \left(\varepsilon - \frac{2}{W}\right)W\right) + 2 = Q_i^W + \varepsilon W,$$

proving the claim. □

V. EVALUATION

In this section, we present an empirical study and compare WCSS and LWCSS.

Experimental Setup. We implemented WCSS and LWCSS in Python 3.7.6. The evaluation was performed on an AMD EPYC 7313 16-Core Processor with an NVIDIA A100 80GB PCIe GPU, running Ubuntu 20.04.6 LTS with Linux kernel 5.4.0-172-generic, and TensorFlow 2.4.1. We extend WCSS by incorporating a predictor, which is built using an LSTM network, and compare it against traditional WCSS using real-world datasets.

Datasets. Our real datasets comprise CAIDA Internet Traces [16] obtained from multiple sources: (1) equinix-chicago 2016 high-speed monitor is located at an Equinix datacenter in Chicago and is connected to a backbone link of a Tier 1 ISP between Chicago, IL and Seattle, WA (2) equinix-sanjose 2014 monitor (SJ) is positioned at an Equinix

TABLE I: Traces from [16] used in the evaluation

Trace	#Items	#Uniques
Chicago	88,529,637	1,650,097
NY	63,284,829	2,968,038
SJ	188,511,031	2,922,904

TABLE II: Predictor Accuracy

Metric	**Value**
F1 Score (Chicago)	81.3%
F1 Score (NY)	87.3%
F1 Score (SJ)	83.5%

datacenter in San Jose, CA, connected to a backbone link of a Tier 1 ISP between San Jose, CA and Los Angeles, CA (3) The equinix-nyc 2018 monitor is located at an Equinix datacenter in New York. These traces are summarized in Table I.

For the synthetic data, we considered the Zipf distribution where the items are selected from a bounded universe, and the frequencies of an item with rank R is given by $f(R, \alpha) = \frac{constant}{R^\alpha}$, where α is a skewness parameter. For predictions, we employed a synthetic predictor that computes the true count for each item, and then adds an error which is drawn i.i.d. from a normal distribution with mean parameter 0 and standard deviation $\sigma = 1$.

Metrics. In the approximation error evaluation, we use Root Mean Square Error (RMSE). When an item is encountered, we query its frequency estimation immediately and calculate the RMSE. For performance of operations, we use updates or queries per second.

A. Accuracy Comparison

Figure 4 shows the accuracy (RMSE) as a function of memory for WCSS and LWCSS setting $W = 2^{10}$. This evaluation includes three real datasets: Chicago, NY, and SJ, and utilizes a pre-trained LSTM predictor. The predictor's performance on each dataset is summarized in Table II. The results demonstrate that LWCSS outperforms WCSS in terms of accuracy across all three datasets since the filtering of single-occurrence items enhances the SS's accuracy which is crucial for overall accuracy. Figure 4 demonstratea the feasibility of our approach using the next arrival predictor.

Figure 5a shows the RMSE as a function of window size using the Chicago dataset. As seen, as the window size increases, the RMSE also increases. Additionally, the performance improves as the window is smaller because the ratio of single items becomes larger, as shown in Figure 3.

B. Performance Comparison

Figures 5b and 5c show the performance of WCSS and LWCSS in terms of the update and query time respectively using the Chicago dataset and setting $W = 2^{13}$. The query performance of the two algorithms appears closely aligned due to their consistent operations. The update performance of
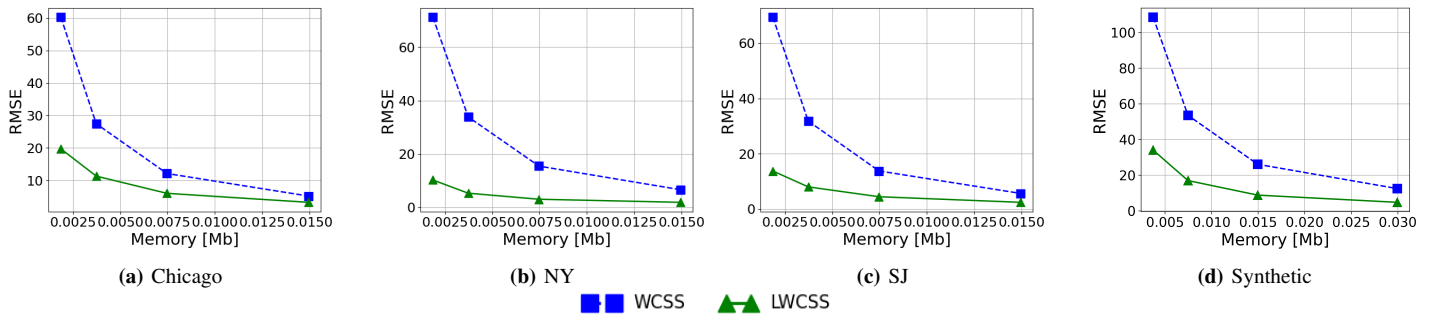
(a) Chicago **(b)** NY **(c)** SJ **(d)** Synthetic

WCSS ■■ LWCSS ▲▲

Fig. 4: Accuracy comparison of WCSS and LWCSS vs. Memory (Megabytes) using real datasets (Chicago, NY and SJ)

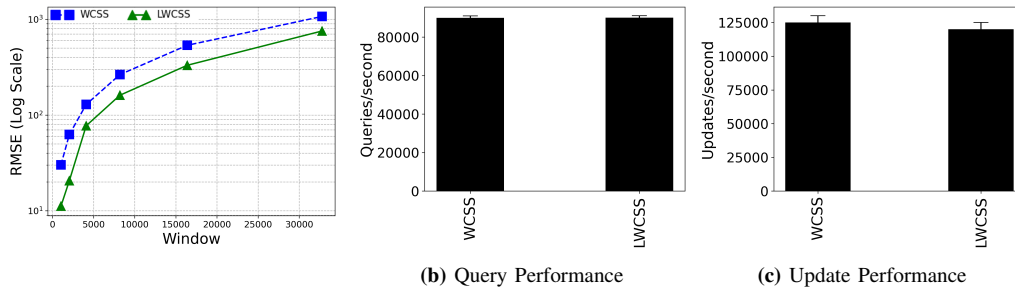**(b)** Query Performance **(c)** Update Performance

Fig. 5: Query and update performance using Chicago dataset and setting $W = 2^{13}$.

LWCSS is slightly worse than WCSS due to the additional operation of inserting single occurence items into the Bloom filter.

VI. FUTURE WORK AND EXTENSIONS

A. An alternative approach: frequency prediction within time frames

A potential direction for future work is to explore an alternative approach that learns item frequencies within each given time frame. By predicting the frequencies of each item in the frame, we can apply previous work introduced by [8] to the sketch that tracks item frequencies within each frame. However, a challenge with this approach is that we need to retrain the predictor in each frame. One way to overcome this computational inefficiency in a streaming setting is: Instead of training a new predictor for each frame, we could employ a single predictor and retrain it periodically to adapt to shifts in the data distribution across frames.

Specifically, we can perform the retraining process after observing items for M frames. The value of M will be determined based on the extent of accuracy drop in the predictor; smaller values of M should be set if the predictor incurs a higher accuracy drop. This periodic retraining aims to strike a balance between maintaining prediction accuracy and minimizing computational overhead. To enhance the efficiency of the retraining process, we can utilize recent advances in transfer learning within deep learning [17]. In particular, we can avoid retraining the model from scratch and shorten the training cycle by selectively retraining only specific layers in

the model using information from recent frames. The main intuition behind this approach is that in many neural networks, the initial layers capture general features, while the later layers focus on specific features that are more dependent on the problem at hand (in our case, these later layers are sensitive to distribution shifts). During the retraining process, we can freeze all hidden layers in the LSTM and fully-connected networks and then selectively retrain only those layers connected to the input and output of each network. As long as the dimensions of the frozen layers remain constant at any point in time, it is acceptable to reuse them. By preserving the learned general features and selectively updating the task-specific layers, we can leverage the knowledge gained from previous frames while efficiently adapting to the current data distribution.

As a future direction, we will compare this frame-based frequency prediction approach with our existing method using the next arrival predictor. This analysis could provide insights into the trade-offs between the two approaches and guide the selection of the appropriate technique based on the specific window size.

B. General learned sliding window framework

Stream processing encompasses three fundamental tasks: membership queries, frequency queries, and heavy hitter queries. Membership queries determine whether a given item is present within the sliding window. Frequency queries report the occurrence count of a specific item. Heavy hitter queries identify all items whose frequencies exceed a predetermined

threshold. Our research focuses on frequency queries over data streams within a sliding window setting. We recognized that the work proposed by [12] introduced a generic framework, termed Sliding Sketches, which can be applied to existing solutions for the above three tasks, enabling them to support queries over sliding windows. This work employed a similar approach of dividing the stream into frames (referred to as segments in their work). Consequently, our approach could intuitively be applied to this general framework.

VII. Related Work

The problem of estimating item frequencies over sliding windows was first studied in [18]. To estimate frequency within a $W\varepsilon$ additive error over a window of size W, their algorithm requires $O(\varepsilon^{-1}\log^2\varepsilon^{-1}\log W)$ bits of memory. This memory requirement was later optimized to $O(\varepsilon^{-1}\log W)$ bits as highlighted in [19]. Hung and Ting in [20] further refined this by improving the update time to a constant and locating all heavy hitters in the optimal $O(\varepsilon^{-1})$ time. The WCSS algorithm, as introduced in [5], also provides frequency estimates in constant time.

Algorithms with predictions is, as we have stated, a rapidly growing area. The site [21] contains a collection of over a hundred papers on the topic. The idea of using predictions to specifically improve frequency estimation algorithms appears to have originated with [8], where they augmented a learning oracle of the heavy hitters into frequency estimation sketch-based algorithms. Later [22] and [9] explored the power of such an oracle, showing that it can be applied to a wide array of problems in data streams. All these papers use neural networks to learn certain properties of the input. We, however, differ from those papers because we consider the sliding window setup, in which properties derived from window resolution can differ significantly from those derived from the entire stream, and therefore, other predictions are required.

VIII. Conclusion

We have presented a novel method to improve sliding window algorithms for approximate frequency estimation by incorporating a learning model that filters out low-frequency "noisy" items. Past research on learning-augmented algorithms does not perform well in the sliding window settings due to variations between the properties of the sliding window resolution and the entire data stream.

We have demonstrated the benefits of our design both analytically and empirically.
Code Availability: All code is available online [23].

References

[1] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[2] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.

[3] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM journal on computing*, vol. 31, no. 6, pp. 1794–1813, 2002.

[4] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2004, pp. 286–296.

[5] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[6] O. Papapetrou, M. Garofalakis, and A. Deligiannakis, "Sketching distributed sliding-window data streams," *The VLDB Journal*, vol. 24, pp. 345–368, 2015.

[7] L.-K. Lee and H. Ting, "A simpler and more efficient deterministic scheme for finding frequent items over sliding windows," in *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2006, pp. 290–297.

[8] C.-Y. Hsu, P. Indyk, D. Katabi, and A. Vakilian, "Learning-based frequency estimation algorithms." in *International Conference on Learning Representations*, 2019.

[9] R. Shahout and M. Mitzenmacher, "Learning-based heavy hitters and flow frequency estimation in streams," *arXiv preprint arXiv:2406.16270*, 2024.

[10] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," *Journal of the ACM (JACM)*, vol. 68, no. 4, pp. 1–25, 2021.

[11] R. B. Basat, R. Friedman, and R. Shahout, "Stream frequency over interval queries," *Proceedings of the VLDB Endowment*, vol. 12, no. 4, pp. 433–445, 2018.

[12] X. Gou, L. He, Y. Zhang, K. Wang, X. Liu, T. Yang, Y. Wang, and B. Cui, "Sliding sketches: A framework using time zones for data stream processing in sliding windows," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1015–1025.

[13] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Database Theory-ICDT 2005: 10th International Conference, Edinburgh, UK, January 5-7, 2005. Proceedings 10*. Springer, 2005, pp. 398–412.

[14] M. Mitzenmacher and S. Vassilvitskii, "Algorithms with predictions," *Communications of the ACM*, vol. 65, no. 7, pp. 33–35, 2022.

[15] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 489–504.

[16] P. Hick, "CAIDA Anonymized Internet Trace, equinix-chicago," 2016.

[17] Y. Bengio, "Deep Learning of Representations for Unsupervised and Transfer Learning," in *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, 2012.

[18] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *ACM PODS*, 2004.

[19] L. Lee and H. F. Ting, "A simpler and more efficient deterministic scheme for finding frequent items over sliding windows," in *ACM PODS*, 2006.

[20] R. Y. S. Hung, L. Lee, and H. Ting, "Finding frequent items over sliding windows with constant update time," *Inf. Proc. Let'*, 2010.

[21] https://algorithms-with-predictions.github.io/, "Algorithms with predictions github repository."

[22] T. Jiang, Y. Li, H. Lin, Y. Ruan, and D. P. Woodruff, "Learning-augmented data stream algorithms," in *International Conference on Learning Representations*, 2019.

[23] "Open source code," https://anonymous.4open.science/r/LWCSS-A283.