Of Mice and Models: Improving Learning-Based Frequency Estimation

Sacha Servan-Schreiber MIT CSAIL
Tal Wagner Microsoft Research
Piotr Indyk MIT CSAIL
Tim Kraska MIT CSAIL

Abstract

Efficiently estimating the frequency of elements in data streams has a wide range of important applications. Recently, learning-based algorithms have been proposed to reduce estimation errors in the "heavy hitter" setting, a common feature of real data streams where a small number of elements dominate frequency.

In this work, we develop a new learning-based algorithm for frequency estimation that provably improves on the state-of-the-art learning-based approach. Our algorithm uses distinct element sampling to estimate the frequencies of "typical" elements in the stream. In turn, this allows us to improve overall estimation by partitioning the heavy-hitters and computing their exact frequencies.

Compared to existing approaches, our algorithm improves the theoretical error guarantees and concrete estimation accuracy when evaluated on real-world data. In our empirical evaluation, we show up to $8\times$ reduction in average absolute error and over $40\times$ reduction in average relative error.

1 Introduction

Frequency estimation algorithms for data streams have many important applications ranging from networking systems [11, 26, 19] to natural language processing [14]. Frequency estimation is used when gathering statistics (e.g., frequency of search terms in a search engine), stock trading, and even in security where frequency estimation helps deter the use of weak passwords [25]. Due to the wide applicability of this problem, many space-efficient algorithms were developed. Widely-used approaches, such as Count-Min [7] and Count-Sketch [5], hash stream elements into a (small) table, and report the number of elements hashed into each entry. That is, the frequency of an element is approximated by the count(s) in the table entry that the element hashed to.

Recently, Hsu *et al.* [16] (ICLR 2019) proposed a learning-based algorithm to augment hashing-based sketches using a "hint" oracle. The oracle is instantiated as a machine-learning model trained to recognize general properties of frequent elements (a.k.a. heavy-hitters or **elephants**) in the stream. A fraction of the memory allocated to the sketch is then used to count the frequency of labeled heavy-hitters with *perfect* accuracy (by storing them separately in a table). The frequencies of all other elements (a.k.a. light-hitters or **mice**) are estimated by the hashing-based sketch, which uses the remaining memory allocated to the algorithm. Separating the heavy-hitters serves to reduce the overall estimation error, as those elements no longer collide with the elements stored by the hashing-based sketch.

While the algorithm of Hsu *et al.* [16] provably decreases the error for common distributions, such as the Zipfian, we find that its error can be decreased even further while simplifying the algorithm considerably. The reason is as follows. Since the algorithm identifies and stores (most of) the heavy hitters separately, the elements that are fed to the hashing-based algorithm are mostly infrequent, to the point that their true frequencies are comparable to (or smaller than) the estimation error guaranteed by the sketching algorithm. As a result, the space allocated to that algorithm is not used

Preprint. Under review.

effectively. Instead, we observe that the space can be better utilized by reallocating it to store more heavy elements, and use a crude, low-space estimation method for all other elements (i.e., the mice).

The above observation is the starting point of our work. Our algorithm uses almost all of the available memory for storing the exact frequencies of the heaviest elements and does not use a hashing-based sketch at all. Instead, the frequencies of the remaining elements are estimated by sampling a small number of "light" elements, computing their exact frequencies, and returning their median. Perhaps counter-intuitively, we show that for natural heavy-tailed distributions, including Zipfian, this simple idea yields asymptotically smaller error[1]. Moreover, while Hsu *et al.* [16] requires fine-tuning parameters to determine the optimal memory allocation strategy, our approach adapts to the data stream automatically and is much simpler to implement.

Contributions. Our main contribution is a novel algorithm that provably improves frequency estimation errors on heavy-tailed distributions compared to the state-of-the-art approach of Hsu *et al.* [16]. We provide theoretical and empirical justification to our approach. On real-world data, we show a $8\times$ improvement for the mean absolute error and up to $44\times$ for the mean relative error.

We also evaluate the impact of noise in the frequency oracle to determine the extent to which (accurately) learning the properties of heavy-hitters reduces estimation error. Our results provides insight into how "traditional" algorithms can benefit from learning-based approaches such as ours.

Limitations. While we empirically improve the absolute and relative error in frequency estimation, we perform worse on the *weighted* error metric. We show that this is the case in situations where the classification accuracy of the model is poor. This suggests that the approach of Hsu *et al.* [16] may be favorable when the heavy-hitters in the stream are difficult to learn and are incorrectly classified by the oracle. An additional limitation of our approach (one that is shared with [16]) is that it only applicable to large data streams (i.e., 100MB or larger) as the model space usage needs to be amortized.

2 Related Work

Related work falls into two camps: traditional streaming algorithms and learning-based algorithms.

Traditional streaming algorithms. Frequency estimation in data streams is a well studied problem with a wide array of algorithms in existence (see Cormode and Marios [6] for a summary of approaches). Approximate algorithms, based on hashing techniques, are widely used in practice. These include Count-Sketch [5], Count-Min [7], and multi-stage filters [11]. Hashing-based approaches are connected to other applications as well, namely compressed-sensing and dimensionality reduction [4, 10].

There also exists several approaches that do not make use of hashing [23, 9, 17, 22], offering a different set of tradeoffs relative to hashing-based solutions. For example, unlike hashing-based algorithms that we study in this paper, they cannot handle element deletions.

Algorithm	**Expected Error**		
	Absolute	Relative	Weighted
Hsu *et al.* [16]	$\frac{N}{B}$	$\frac{N}{B}$	$\frac{N}{B}\log(\frac{N}{B})$
Theorem 4.2	$\log(\frac{N}{B})$	$\frac{1}{2} - O(\frac{B}{N})$	$\frac{N}{B}$

Table 1: Asymptotic error comparison to Hsu *et al.* [16] over data streams following Zipfian distributions. N denotes the number of unique elements and B denotes the space allocated.

Learning-Based algorithms. Recently, there have been several proposals for combining traditional approaches with machine-learning to improve accuracy and space usage of streaming algorithms on real-world data distributions. Solutions to combinatorial optimization problems have benefited from "learning-augmented" algorithms [15, 2, 20]. Reinforcement learning has been applied to graph optimization problems [8]. Learning-based approaches have likewise been developed for database optimization [21] and several indexing data structures [18, 24].

[1] In fact, this holds even if the frequency estimator of light elements always returns zero, although the median estimator works better in practice. Again, the reason is that for light elements, estimation error is comparable to (or higher than) the true frequency values. See Section 4.4 for a quantitative analysis of this phenomenon.

A learning-based approach for frequency-estimation in particular has been proposed by Hsu *et al.* [16] and analyzed by Amand *et al.* [1]. The algorithm is tailored towards frequency estimation of heavy-hitters in a data stream and uses a model trained to classify heavy-hitters based on their properties. Elements classified as heavy-hitters are stored separately in a "cutoff table" and their frequency computed exactly. All other element frequencies are estimated using a hashing-based approach. We refer to the algorithm of Hsu *et al.* [16] as Cutoff Count-Sketch. Compared to Cutoff Count-Sketch, our approach significantly improves accuracy, concretely and asymptotically (see Table 1).

3 Preliminaries

In this section, we cover the necessary preliminaries for understanding frequency estimation, existing hashing-based approaches, and the building-blocks for our algorithm.

3.1 Notation

The universe of elements is denoted U. To keep the notation simple, we will overload it and use U to denote the cardinality of the set U as well. We denote the number of non-zero frequencies in a data stream by N. We use $:=$ to denote assignment and use $[n]$ to denote the set $\{1 \dots n\}$. The number of elements in a set S is denoted $|S|$. An estimate of a value v is denoted by $\tilde{v}$.

3.2 Estimation Error

The algorithm considered in this paper, and frequency estimation algorithms in general, output estimated frequencies $\tilde{f}_i$ for each true frequency f_i of element $i \in U$. To measure the overall estimation error between the frequencies $\mathcal{F} := \{f_1, f_2, \dots\}$ and their estimates $\tilde{\mathcal{F}} := \{\tilde{f}_1, \tilde{f}_2, \cdots\}$, we will use the *expected* error $E_{i \in U}[e_i]$.

This leads to the following three error definitions. The expected **absolute error** is defined as

$$\mathrm{Err}_{absolute}(\mathcal{F}, \tilde{\mathcal{F}}) := \frac{1}{N} \sum_{i \in U} |\tilde{f}_i - f_i|, \tag{1}$$

the expected **relative error** is defined as

$$\mathrm{Err}_{relative}(\mathcal{F}, \tilde{\mathcal{F}}) := \frac{1}{N} \sum_{i \in U} |\tilde{f}_i - f_i| / f_i, \tag{2}$$

and the expected **weighted error** is defined as

$$\mathrm{Err}_{weighted}(\mathcal{F}, \tilde{\mathcal{F}}) := \frac{1}{N} \sum_{i \in U} |\tilde{f}_i - f_i| \cdot f_i. \tag{3}$$

We use these three error metrics to evaluate the accuracy of our algorithm and compare to existing approaches for frequency estimation. As argued by Hsu *et al.* [16], it is more natural to analyze the expected error in a learning-based setting rather than the traditional "(ϵ, δ)" error formulations, which would require tuning two objectives: ϵ and δ. We note that Equation (3) is the only metric evaluated in Hsu *et al.* [16], which is an error metric that is biased towards accuracy of heavy-hitters.

3.3 Count-Sketch & Count-Min

We briefly summarize Count-Sketch [5], a widely used hashing-based frequency estimation algorithm which we use as a baseline for comparison. Count-Sketch uses multiple hash functions to compress the stream in a way that preserves element frequency estimates.

For parameters H and B, Count-Sketch uses $2H$ pair-wise independent hash functions $h_i : U \to [B]$ and $s_i : U \to \{-1, 1\}$. An array C of size $H \times B$ is allocated and used as follows. For each element j in the stream, add $s_k(j)$ to counter $C[k, h_k(j)]$, for all for $k \in [H]$. The estimated frequency is then computed as $\hat{f}_j := \mathsf{median}(\{s_k(j) \cdot C[k, h_k(j)] \mid k \in [H]\})$.

A variant of Count-Sketch, known as Count-Min, follows a similar idea but does not use the sign hash s_i, which results in a *biased* estimation [7].

Cutoff Count-Sketch [16] (see Section 2) augments Count-Sketch and Count-Min with a learning based heavy-hitter classifier to separate heavy-hitters from all other elements. Cutoff Count-Sketch improves error by removing elements that heavily bias estimates inside the sketch.

3.4 Zipfian Distribution

In our theoretical analysis we assume that the element frequencies follow the Zipf Law. That is, if we order the elements according to their frequencies such that $f_1 \geq f_2 \geq \ldots \geq f_N$, then we have that $f_j \propto 1/j$. Zipfian distribution is a a popular and realistic model for data streams, see e.g., [5, 22].

4 Learning-Based Cutoff-Median Sketch

In this section we introduce our approach to learning-based frequency estimation. Section 4.1 provides an overview of our algorithm, Section 4.2 introduces the main technique in our idea, and Section 4.4 provides a theoretical analysis of our algorithm.

4.1 Overview

Our approach is simple: for a fixed number of buckets B, we use the frequency oracle FreqOracle to separate heavy-hitters "on-the-fly". We store B of the heaviest elements in a table and compute their *exact* frequencies. For all other elements, we approximate their median frequency through a careful distinct element sampling technique computed in a streaming fashion, which we output as the frequency estimate.

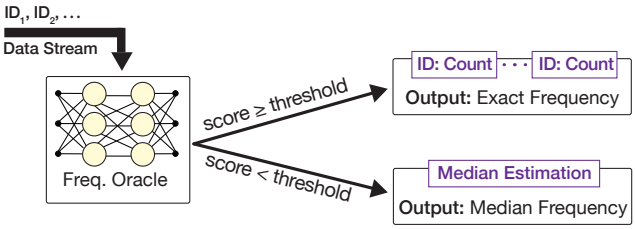

Figure 1: Block-diagram illustrating the high-level overview of our algorithm. An oracle is used to classify elements. Exact frequency estimates are reported for all elements classified as heavy-hitters. The estimated median frequency is reported for all other elements.

Using the oracle to classify heavy-hitters is also the main idea for Cutoff Count-Sketch [16]. However, we show that our median-estimation significantly improves over Cutoff Count-Sketch. The rationale behind this improvement is two-fold:

1. We can dedicate nearly all the space to computing exact frequencies of heavy-hitters.
2. For all other elements, we can efficiently estimate their median frequency, which we show is the optimal[2] frequency estimate (see Theorem 4.2).

In Section 4.2 we describe our technique for estimating the median frequency of elements, and present our full algorithm for frequency estimation in Section 4.3.

4.2 Streaming estimate of the median frequency

As explained in Section 4.1, the crux of our approach relies on accurately estimating the median frequency of elements not stored in the cutoff table. Doing so efficiently (using little space and in a single pass over the stream) is non-trivial. Our technique to achieve this exploits a dynamically computed sample of unique elements. Frahling *et al.* [13] show that it is possible to sample a random element in a single pass over the stream. We extend this to efficiently sampling a random subset of elements (along with their frequencies) from the stream. We then show that using such a subset, we can compute an estimate for the median frequency of *all* elements by outputting the median frequency in the subset. Our approach is natural in hindsight and may be of independent interest.

The dynamic sampling data structure of Frahling *et al.* [13], referred to henceforth as DS, has the following functionality:

- DS.Add(i): processes a new element from the stream.

[2]The median frequency is optimal for the absolute error metric under the Zipfian distribution. See our theoretical analysis in Section 4.4

- DS.Sample(): returns a random element from the stream along with its frequency in $\mathcal{S}$.

To formalize the idea behind our median estimation technique, let $q(i)$ denote the rank of the ith element when the stream elements are ranked by their frequencies in descending order. Thus, if i_{max} is the most frequent element then $q(i_{max}) = 1$, if i_{min} is the least frequent one then $q(i_{min}) = N$, and if i_{med} is the median-frequency element then $q(i_{med}) = \lfloor \frac{1}{2} N \rfloor$.

Lemma 4.1 (median approximation). *Fix any $\epsilon > 0$ and $n := O(\log(NU)/\epsilon^2)$. Then,*

- *using* DS *to sample a random set of n elements from the stream requires $O(\log^2(NU)/\epsilon^2)$ bits of storage space,*
- *with probability $1 - 1/N^{O(1)}$, Algorithm 1 (median estimation) returns an element i such that $(1-\epsilon)\frac{1}{2}N \leq q(i) \leq (1+\epsilon)\frac{1}{2}N$.*

Put simply, Lemma 4.1 states that Algorithm 1 returns an element whose frequency rank is a $(1 \pm \epsilon)$-approximation of the median rank (which is $\lfloor \frac{1}{2} N \rfloor$). If the element frequencies are sufficiently smooth, as they are in the Zipfian case, then the frequency of the returned element is close to the median frequency; see Appendix D for proofs.

Algorithm 1: Median Estimation

Input: Stream $\mathcal{S}$ of elements $i_1, i_2, \ldots$
Output: Estimated median frequency $\tilde{f}_{med}$ of $i_1, i_2, \ldots$
Parameters: Number of samples n.

```
Procedure EstimateMedian(S, n)
    for each element i in the stream do
        DS.Add(i)
    {(i'_j, f'_j) | j ∈ [n]} ← DS.Sample(n)
    return median({f'_1, ..., f'_n})
```

4.3 Our Algorithm: The Cutoff-Median Sketch

We present the full algorithm in Algorithm 2 and depict the block-diagram of the algorithm in Figure 1. At a high level, we use the frequency oracle (denoted FreqOracle) to predict the frequency of each element in the data stream. We then either increment the count of the element in the cutoff table if classified as a heavy-hitter and discard the element otherwise. The estimated frequency for discarded elements is computed by Algorithm 1 which outputs the median frequency as the estimate.

4.4 Analysis in the Zipfian case

In this section we analyze the theoretical guarantees provided by our approach and compare it to Hsu *et al.* [16] and the subsequent theoretical analysis of the algorithm of Aamand *et al.* [1]. Our analysis is set in a simplified setting where the input distribution is Zipfian, and the oracle identifies the heavy-hitters without error. This is the same theoretical setting considered in prior work [16, 1].

Theorem 4.2. *Suppose that the true frequency of each element i is N/i. Let $B > 0$ be the number of heavy-hitters whose frequencies are computed exactly in Algorithm 2 by storing them in a table. Then, Algorithm 2 has the following error bounds:*

1. *The expected absolute error is $O(\log(N/B))$.*
2. *The expected weighted error is $O(N/B)$.*
3. *The expected relative error is $\frac{1}{2} - O(\frac{B}{N})$.*

The error bounds of Theorem 4.2 significantly improve over prior work [16, 1] across all error metrics — see Table 1 for a comparison. Concretely, these bounds correspond to a 2-40$\times$ improvement in estimation error compared to [16] (see Section 5 for empirical comparison).

Theorem 4.2 is proven in full in Appendix D. In the remainder of this section, we focus on the analysis for the expected absolute error of our algorithm versus that of Hsu *et al.* [16]. Both algorithms use B counters to store the exact heavy-hitter frequencies, and B' additional memory to estimate the remaining element frequencies.

Algorithm 2: Cutoff-Median Sketch

Input: Stream $\mathcal{S}$ of items $i_1, i_2, \ldots$

Output: Estimated frequencies $\hat{f}_1, \ldots, \hat{f}_N$ for all N distinct items in the stream.

Parameters: Frequency oracle $\mathsf{FreqOracle}(\cdot)$ and B.

Procedure $\mathsf{CutoffMedian}(\mathcal{S})$
  $T, W \leftarrow \emptyset$; $\mathsf{thresh} \leftarrow 0$
  for each element i in the stream **do**
    $w_i \leftarrow \mathsf{FreqOracle}(i)$ `// inference`
    if $w_i > \mathsf{thresh}$ **then**
      $T[i] \leftarrow T[i] + 1$ `// update count`
      $W[i] \leftarrow w_i$ `// associated score`
      if $|T| > B$ **then**
        $\mathsf{thresh} \leftarrow \min(W)$
        $j \leftarrow \operatorname{argmin}(W)$
        Remove j from T and W

Procedure $\mathsf{EstimateFrequency}(i)$
  $w \leftarrow \mathsf{FreqOracle}(i)$ `// inference`
  if $w > \mathsf{thresh}$ **then**
    return Exact frequency f computed from T
  else
    return Estimated median frequency $\tilde{f}_{med}$ returned by Algorithm 1

Our algorithm uses a single estimate m for all elements not stored in the cutoff table. Therefore, the expected absolute error (assuming the oracle identifies the heavy-hitters correctly) is

$$\frac{1}{N} \sum_{i \in \text{mice}} |f_i - m|.$$

This expression is minimized when m is the median frequency for all elements not stored in the cutoff table. This is our motivation for dedicating the additional B' memory for a median estimation subroutine in Algorithm 2. In the Zipfian case $f_i = N/i$ we have $q(i) = i$, so Algorithm 1 returns an element i' such that $(1-\epsilon)i_{med} \le i' \le (i+\epsilon)i_{med}$, where i_{med} is the true median element. We then get,

$$\frac{1}{N} \sum_{i \in \text{mice}} |f_i - m| = \sum_{i=B}^{N} \left| \frac{1}{i} - \frac{1}{i'} \right| = \sum_{i=B}^{N} \left| \frac{1}{i} - \frac{1}{(1 \pm \epsilon) i_{med}} \right| \le \sum_{i=B}^{N} \left| \frac{1}{i} - \frac{1}{i_{med}} \right| + O(\epsilon) \cdot \sum_{i=B}^{N} \frac{1}{i_{med}}.$$

Since the median is optimal for the first summand, we can replace it with zero and upper bound by $\sum_{i=B}^{N} \frac{1}{i} = O(\log(N/B))$. The second summand equals $O(\epsilon) \cdot \frac{2(N-B)}{N+B}$ since $i_{med} = \lfloor \frac{1}{2}(N+B) \rfloor$, and is thus dominated by the first summand. Thus the overall error is $O(\log(N/B))$.

We now compare this bound to that of Cutoff Count-Sketch [16] which uses the remaining B' memory for the Count-Sketch. This results in expected additive error for element i equal to $\frac{N}{B'} \log(1 + \frac{B'}{B})$ in the Zipfian case, as per the proof of Theorem 4.3 of Aamand *et al.* [1]. Consequently, the overall expected absolute error is

$$\frac{1}{N} \sum_{i=B}^{N} \frac{N}{B'} \log\left(1 + \frac{B'}{B}\right) = \frac{N-B}{B'} \log\left(1 + \frac{B'}{B}\right),$$

larger than ours by a factor nearly linear in $O(N/B')$.

It is also worth noting that the optimal setting in the latter bound is $B' \approx B$, meaning that the Cutoff Count-Sketch [16] needs to allocate a constant fraction of its space budget to the elements not stored in the cutoff table. Our algorithm uses a fixed budget of $B' = O(\log^2(UN)/\epsilon^2)$ space for these elements (Lemma 4.1), allowing us to allocate most of the space budget to the heavy-hitters.

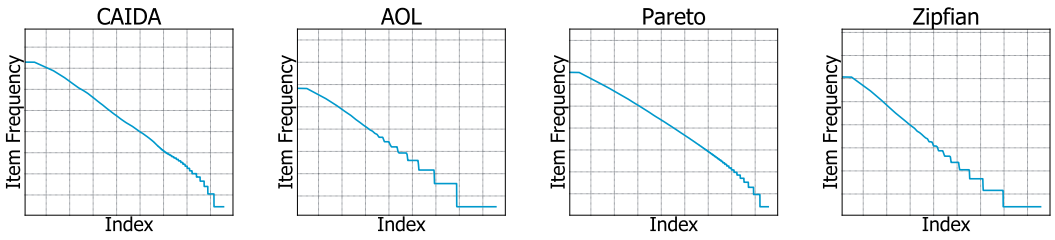

Figure 2: Log-log scaled frequency vs. sorted index of items in each dataset we use in our evaluation.

5 Empirical Evaluation

In this section we empirically evaluate our algorithm on real-world and synthetic datasets to compare our approach with Count-Sketch, Count-Min, and the learning-based Cutoff Count-Sketch [16]. All the code used to evaluate these algorithms is open-source.[3]

Baseline Comparison. We compare our approach (Cutoff-Median Sketch; Algorithm 2) to existing frequency estimation algorithms. Hsu *et al.* [16] only evaluate Cutoff Count-Sketch on the *weighted* error metric (which favors correct estimation of heavy-hitters but not other elements). We additionally evaluate all approaches on absolute and relative error (see definitions in Section 3), which capture the overall estimation accuracy of elements in the stream; not just heavy-hitters.

Models. We use the pre-trained models of Hsu *et al.* [16] [4]. Using the same models allows us to accurately compare the performance improvement of our algorithm over Cutoff Count-Sketch; which, to the best of our knowledge, is the only learning-based frequency estimation algorithm other than ours. We summarize the model training methodology of Hsu *et al.* [16] in Appendix A. We emphasize that the models learn the *properties* of elements in general, *not* the elements themselves.

5.1 Datasets

We use four datasets which have a skewed distribution in element frequency. We provide the element frequency distribution (log-log scale) of each dataset in Figure 2.

CAIDA Internet Packet Traffic Dataset. The CAIDA 2016 Anonymized Internet Traces dataset [5] consists of internet traffic data from a Tier-1 ISP. The traffic data was collected in 2016 from a backbone link between Chicago and Seattle. Each recording session consists of approximately an hour's worth of internet traffic. Each minute contains roughly 30 million packets and 1 million unique packet flows. The distribution of packet flows follows a Pareto distribution, with a few flows dominating the packet traffic as can be seen in Figure 2.

AOL Search Queries Dataset. To show utility of our algorithm for frequency estimation in query optimization, we evaluate it on the AOL query log dataset consisting of 21 million search queries issued by 650 thousand users. The data contains 3.8 million queries collected from 650 thousand users over a 90 day period. Each query is a search phrase consisting of one or more words. The search queries follow a Zipfian distribution with a small number of search terms (e.g., "google") dominating search frequency and a high number of unique queries (e.g., "austin certificates of occupancy"). We note that the AOL dataset is known to contain personally identifiable information [3] but has been in the public domain for over a decade and used to benchmark a number of prior streaming algorithms.

Synthetic Zipfian & Pareto Datasets. To empirically analyze the impact of classification errors, we generate two synthetic datasets sampled from the Zipfian and Pareto distributions, respectively. For the Zipfian dataset, we use parameter $\rho = 2.5$ which approximates the distribution of the AOL

[3] Available at `https://github.com/sachaservan/cutoff-median-sketch`.

[4] The models are available under MIT license at `https://github.com/chenyuhsu/learnedsketch`

[5] Dataset available on request at `https://catalog.caida.org/details/dataset/`

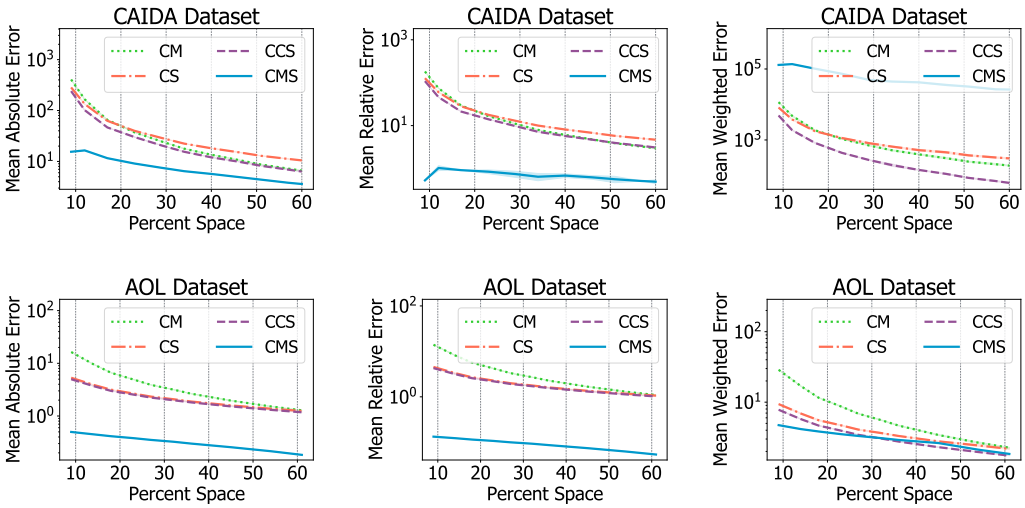

Figure 3: Evaluation of Count-Median Sketch (CMS; Algorithm 2) on the the CAIDA and AOL datasets. Comparison provided to Count-Sketch (CS), Count-Min (CM), and Cutoff Count-Sketch (CCS) on absolute, relative, and weighted error.

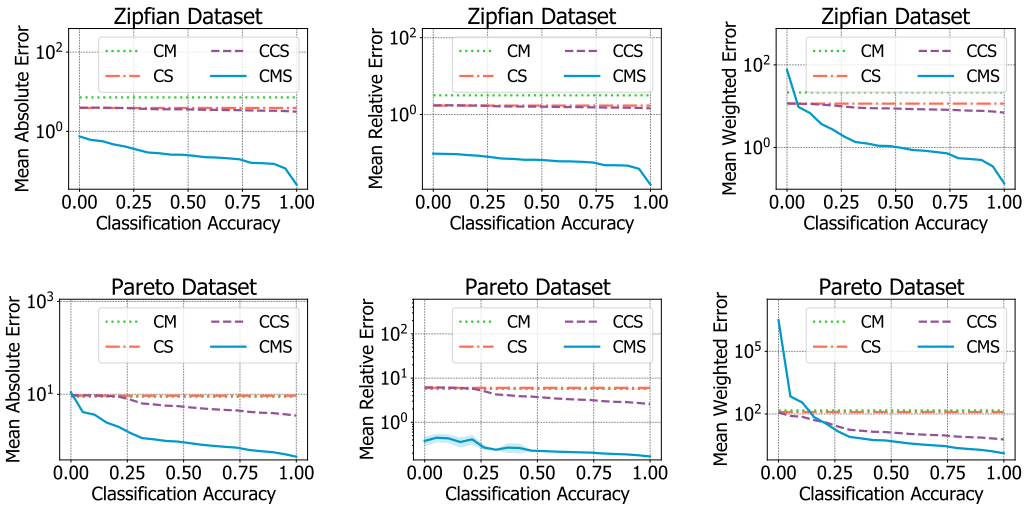

Figure 4: Evaluation of Algorithm 2 (CMS) on the synthetic Zipfian and Pareto datasets with a varying degree of noise in the oracle classification (ranging from random to perfect classification). Space usage for all algorithms is fixed at 25% (see Appendix C for additional experiments). Comparison provided to Count-Sketch (CS), Count-Min (CM), and Cutoff Count-Sketch (CCS) with respect to the mean absolute, relative, and weighted error metrics.

data. For the Pareto distribution, we use the parameter $\alpha = 1$ which approximates the distribution of the CAIDA dataset (see Figure 2). We then model a noisy classification oracle by introducing controlled noise. We describe the details of our synthetic oracle in Section 5.3.

5.2 Methodology

We report the average error over 10 independent trial runs with a 95% confidence interval (even though the variance was mostly very low across trials).

For evaluating Cutoff Count-Sketch, we follow the same methodology as followed by Hsu *et al.* [16]. Specifically, for the internet traffic dataset we use the first seven minutes of traffic data to train the

model and find the optimal cutoff parameter. We then evaluate Cutoff Count-Sketch on a validation dataset consisting of the 50th minute of traffic. For the AOL data, we use the first five days of queries for training and the 80th day of queries for the validation dataset. The choice of training and validation data replicates the setup of Hsu *et al.* [16] for evaluating Cutoff Count-Sketch exactly [6].

Space usage. For both real-world datasets, we amortize the space used by the model across the data (50 minutes for Internet Traffic; resp. 80 days for AOL) since the trained models are *general* and can be used indefinitely. This also matches the model space amortization in [16].

We assume that each bucket in the cutoff tables requires 8 bytes (4 bytes to store the item ID and 4 bytes to store the count). For the median estimation algorithm (Algorithm 1), we set n (the number of samples to take) to be 0.05% of the dataset size, which we find to be a good heuristic. For median estimation, each sample requires 640 bytes of space to (in addition to a constant overhead of 64 bytes for counting distinct elements in the stream [13]). In total, this translates to $640n + 64$ bytes of space allocated towards instantiating the median sketch.

The space usage reported in section 5.3 includes all the space necessary for storing the cutoff table, the median sketch, and the (amortised) model.

5.3 Results

Inference performance. Inference takes 2.8 microseconds per item on a single GPU; without any optimizations. More aggressive, optimization, specialized hardware, and advances in GPU technology can drastically help in reducing inference time [16].

Absolute error. The absolute error metric measures the overall frequency estimation error. We first measure the average absolute error of Algorithm 2 compared to Count-Sketch and Cutoff Count-Sketch. The average absolute error per item is decreased by a factor of 2 to $8\times$ over Count-Sketch. When compared to Cutoff Count-Sketch, our algorithm improves the average absolute error by a factor of 1.72 to $8\times$.

Relative error. The average relative error measures the prediction accuracy of mice, which, in a sense, is the direct opposite to the measurement performed in [16]. On average, the relative error per item is improved by a factor of 9 to $57\times$ over Count-Sketch and 6 to $44\times$ over Cutoff Count-Sketch.

Weighted error. For the weighted error metric, which measures the prediction accuracy of heavy-hitters, Algorithm 2 is occasionally worse on the real-world datasets when compared to both Count-Sketch and Cutoff Count-Sketch. However, we note that on the Zipfian-distributed datasets (AOL and Synthetic Zipfian), our approach is on-par with (and occasionally outperforms) Cutoff Count-Sketch.

Impact of classification accuracy. We examine the effect of heavy-hitter classification accuracy on the overall performance of Algorithm 2 and Cutoff Count-Sketch. We fix the space usage to 25% and vary the noise level of the oracle as follows. On one hand, we build a *random* oracle, where each classification is *uniformly random*. This models a "worst-case" setting where the oracle is completely useless to learning-based approaches. At the other extreme, we model a *perfect* oracle which always classifies heavy-hitters *exactly*. For everything in between, we model a "noisy" oracle as follows. We randomly assign each element to W buckets (we vary W between $0.1N$ and $10N$). The oracle then outputs the *total* frequency of all elements that collided to the same bucket as element i. Notice that this has the effect of modeling a noisy oracle that favors the estimation of heavy-hitters, and mimics the estimation error we have observed in the models trained on the real-world datasets.

We report results in Figure 4. Surprisingly, our algorithm outperforms both vanilla Count-Sketch and Cutoff Count-Sketch on absolute and relative error metrics even when the oracle is very noisy (and even random). For the weighted error metric, we find that our approach is inferior to Count-Sketch and Cutoff Count-Sketch *when instantiated with a highly noisy oracle*. However, our approach quickly improves on the weighted error metric as the level of noise decreases.

[6] See Appendix A for additional details on the training and evaluation methodology of Hsu *et al.* [16]

These results suggest that existing algorithms for frequency estimation of heavy-tailed datasets are far from optimal, even in the "traditional" (non learning-based) setting. We believe that these results open up an avenue for future work to explore better algorithms for frequency estimation.

References

[1] Anders Aamand, Piotr Indyk, and Ali Vakilian. (learned) frequency estimation algorithms under zipfian distribution. *arXiv preprint arXiv:1908.05198*, 2019.

[2] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to branch. In *International Conference on Machine Learning*, 2018.

[3] Michael Barbaro, Tom Zeller, and Saul Hansell. A face is exposed for aol searcher no. 4417749. *New York Times*, 9(2008):8, 2006.

[4] Emmanuel J Candès, Justin Romberg, and Terence Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on information theory*, 52(2):489–509, 2006.

[5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[6] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment*, 1(2):1530–1541, 2008.

[7] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[8] Hanjun Dai, Elias Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6351–6361, 2017.

[9] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360. Springer, 2002.

[10] David L Donoho. Compressed sensing. *IEEE Transactions on information theory*, 52(4):1289–1306, 2006.

[11] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3):270–313, 2003.

[12] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[13] Gereon Frahling, Piotr Indyk, and Christian Sohler. Sampling in dynamic data streams and applications. *International Journal of Computational Geometry & Applications*, 18(01n02):3–28, 2008.

[14] Amit Goyal, Hal Daumé III, and Graham Cormode. Sketch algorithms for estimating point queries in nlp. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1093–1103. Association for Computational Linguistics, 2012.

[15] He He, Hal Daume III, and Jason M Eisner. Learning to search in branch and bound algorithms. In *Advances in neural information processing systems*, pages 3293–3301, 2014.

[16] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. Learning-based frequency estimation algorithms. 2018.

[17] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)*, 28(1):51–55, 2003.

[18] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 489–504. ACM, 2018.

[19] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114. ACM, 2016.

[20] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. In *International Conference on Machine Learning*, 2018.

[21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.

[22] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.

[23] Jayadev Misra and David Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

[24] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In *Advances in Neural Information Processing Systems*, 2018.

[25] Stuart Schechter, Cormac Herley, and Michael Mitzenmacher. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–8. USENIX Association, 2010.

[26] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, pages 29–42, 2013.

A Model Training Methodology

In this section, we summarize the model training methodology of Hsu *et al.* [16] for convenience of the reader. We point to [16] for more details.

Internet Traffic (CAIDA) dataset. The CAIDA dataset contains internet traffic flows between two backbone ISPs recorded on a minute-by-minute basis. The flows are dynamic, changing one minute to the next, however, the IP addresses of the source and destination are more stable: data-centers, companies, and universities tend to generate the most traffic making it possible to identify heavy-hitters. Overall, the traffic flows follow a Pareto distribution (see Figure 2).

The frequency oracle consists of a trained neural network that predicts the packet counts (in log scale) for each flow. The model takes as input the source and destination IP addresses (and ports) of each packet and predicts the frequency of the packet in the flow. Two recurrent neural networks (RNNs) are used to encode the source and destination IP addresses, respectively. Each IP RNN (64 hidden units) takes the IP address bit-by-bit (the IP address bits are hierarchical, with the high-order bits governing a larger region of the IP space), starting with the most significant bit. The final state of the RNN is a feature vector for an IP address. A two-layer fully-connected network (16 and 8 hidden units) is used to encode the ports (one for the source and one for the destination). The final output is the concatenation of the protocol type (e.g., HTTP) and encoded IP and port feature vectors.

AOL dataset. The AOL search query dataset consists of user-submitted queries over the course of multiple days. The queries are relatively consistent across days, with popular search terms such as "google" appearing more frequently than search terms such as "austin certificates of occupancy". The oracle is constructed by training a neural network to predict the number of times each query appears. The model consists of an RNN (256 hidden units) with Long Short-Term Memory (LSTM) cells. The RNN takes as input each query phrase decomposed into characters. The character set consists of all lower-case English letters, numbers, punctuation marks, and a special symbol for all other (unknown) characters. The character IDs are embedded as vectors before being input to the RNN. The final state of the RNN is then fed to a fully-connected layer (32 hidden unit) that predicts the query frequency.

B Additional Optimizations

B.1 Insertion-Only Median Estimation

While our median-estimation algorithm (Algorithm 1) supports a stream where elements are inserted *and* deleted, in this section we consider a space optimization that may be of practical interests in data streams where elements are inserted but never deleted. Such streams are called *insertion-only* streams and appear in many real-world settings where frequency estimation is required (e.g., data analysis and statistics).

We present the insertion-only variant of unique element sampling in Algorithm 3, which can be used in pace of Algorithm 1 for efficiently sampling unique elements in the stream. The algorithm requires estimating the total number of distinct elements in the stream, which we assume can be done efficiently using a Flajolet-Martin sketch [12], hence forth referred to as FM-sketch, having the following functionality:

- $\mathsf{FM.Add}(i)$: processes an element from the stream.
- $\mathsf{FM.Report}$: returns the estimated number of distinct elements in the stream.

Proposition B.1. *Algorithm 3 samples a unique element and its count in the stream $\mathcal{S}$ with probability $1-\delta$ for $\delta > 0$.*

Proof: If the FM-Sketch reports the correct number of distinct elements in the stream, then the correctness of Algorithm 3 follows by inspection. This happens with probability at least $1-\delta$ by the guarantees of [12]. □

Proposition B.2. *Algorithm 3 requires $O(n \cdot \log^2(U) + \log^2(U/\delta)/\epsilon^2)$ space for sampling n unique elements in the stream.*

Proof: The FM-Sketch requires space $O(\log^2(U/\delta)/\epsilon^2)$ for parameters $\epsilon, \delta > 0$. In addition, the arrays C and F each take up $\log^2 U$ bits to represent. The hash functions collectively take up $\log^2 U$

Algorithm 3: Insertion-Only Unique Element

Input: Stream $\mathcal{S}$ of elements $i_1, i_2, \ldots$
Output: Unique element and count (i, c)
Parameters: Element universe U and $k := \lceil \log U \rceil$.

Procedure UniqueElement($\mathcal{S}$)
  $C \leftarrow$ array of k counters.
  $W \leftarrow$ array of k counters initialized to $\perp$.
  Random hash functions $h_1, \cdots h_k$ with $h_j : U \to 2^j$ for $j \in [k]$
  for each element i in the stream **do**
    FM.Add(i)
    for $j = 1$ **to** $\log U$ **do**
      if $h_j(i) = 0$ **and** $W[j] = \perp$ **then**
        $C[j] \leftarrow C[j] + 1$
        $W[j] \leftarrow i$
      else if $W[j] = i$ **then**
        $C[j] \leftarrow C[j] + 1$
  $j \leftarrow \lceil \log$ FM.Report$\rceil$
  return $(W[j], C[j])$

bits. Because the FM-sketch can be reused for multiple samples (it is just an estimate for the total number of elements in the stream), we need $O(n \cdot \log^2(U) + \log^2(U/\delta)/\epsilon^2)$ space to sample n unique elements. □

Remark B.3. Using Algorithm 3 and assuming a universe of 4 bytes, we get that each sample requires 256 bytes of space (excluding the space requirement for the FM sketch), which significantly reduces the space requirement compared to the dynamic stream setting (see Section 5.2).

B.2 Amortized Space-Usage for Median Estimation

We observe that Algorithm 1 (and the insertion-only variant; Algorithm 3) computes the *exact frequency* for the n randomly sampled elements in the stream of discarded items. As a result, we can output the exact frequencies for all n elements used in Algorithm 1 when estimating the frequencies of those elements in Algorithm 2 "for free", which can be of practical interest.

C Additional Experiments

We run our synthetic dataset experiment with a larger space fraction (50%) allocated to all algorithms. In this setting, we see even greater improvement in our approach compared to both Cutoff Count-Sketch and traditional hashing-based approaches (see Figure 4 for comparison with 25% space). Indeed, on the Zipfian dataset, we see that as the space is allocated is sufficiently large and classification accuracy sufficiently good to classify all heavy-hitters (with high probability), our approach achieves perfect accuracy since the median estimation perfectly estimates the frequency of all tail elements.

D Deferred Proofs

In this section we analyze the expected error of our algorithm, and of [16], under the assumption of Zipfian frequencies $f_i = N/i$, for three error measures: absolute, relative, and weighted error.

D.1 Analysis of Our Algorithm: Proof of Theorem 4.2

As in Section 4.4, in the Zipfian case the median mice element if $i_{med} = \lfloor \frac{1}{2}(N + B) \rfloor$. The median sketch reports an element i' such that $(1 - \epsilon)i_{med} \leq i' \leq (1 + \epsilon)i_{med}$. Note that as a consequence,

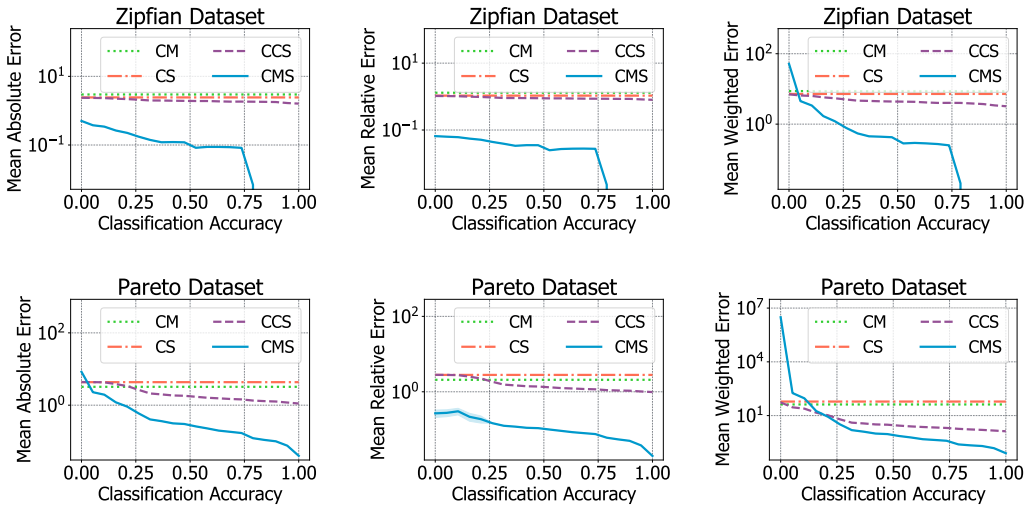

Figure 5: Evaluation of Algorithm 2 (CMS) on the synthetic Zipfian and Pareto datasets with a varying degree of noise in the oracle classification (ranging from random to perfect classification). Space usage for all algorithms is fixed at 50%. Comparison provided to Count-Sketch (CS), Count-Min (CM), and Cutoff Count-Sketch (CCS) with respect to the mean absolute, relative, and weighted error metrics. On the Zipfian dataset, classification accuracy above 0.75 leads to *perfect* accuracy in our approach as all heavy-hitters are stored in the table and the median estimation is exact for all light elements.

since $1/(1 \pm \epsilon) = 1 \pm O(\epsilon)$, for every i we have,

$$\left|\frac{1}{i} - \frac{1}{i'}\right| \leq \left|\frac{1}{i} - \frac{1}{i_{med}}\right| + O(\epsilon) \cdot \frac{1}{i_{med}}.$$

Absolute error. The expected absolute error bound, $\frac{1}{N}\sum_{i=B}^{N}\left|\frac{N}{i} - \frac{N}{i'}\right| = O(\log(N/B))$, was shown in Section 4.4.

Weighted error. The expected weighted error can be bounded as follows:

$$\begin{aligned}\frac{1}{N}\sum_{i=B}^{N}\frac{N}{i}\cdot\left|\frac{N}{i} - \frac{N}{i'}\right| &= \sum_{i=B}^{N}\frac{N}{i}\cdot\left|\frac{1}{i} - \frac{1}{i'}\right| \\ &\leq \sum_{i=B}^{N}\frac{N}{i^2} + \frac{N}{i'}\sum_{i=B}^{N}\frac{1}{i} \\ &\leq \sum_{i=B}^{N}\frac{N}{i^2} + (1+O(\epsilon))\cdot\frac{N}{i_{med}}\sum_{i=B}^{N}\frac{1}{i}.\end{aligned}$$

For the first summand, we have

$$\sum_{i=B}^{N}\frac{N}{i^2} \leq N\sum_{i=B}^{\infty}\frac{1}{i^2} = O\left(\frac{N}{B}\right).$$

For the second summand, we have $\sum_{i=B}^{N}\frac{1}{i} = O(\log(N/B))$, and $N/i_{med} = N/\lfloor\frac{1}{2}(N+B)\rfloor = O(1)$ (since $B < N$). The total error is thus $O(N/B)$.

Relative error. In this case it is simpler to analyze a variant of our algorithm which simply estimates all mice frequencies as 1 (instead of N/i', which is bounded between 1 and $2+O(\epsilon)$). Since $1 \leq N/i$ for every i, the expected relative error equals:

$$\frac{1}{N}\sum_{i=B}^{N}\frac{i}{N}\cdot\left|\frac{N}{i} - 1\right| = \frac{1}{N}\sum_{i=B}^{N}\frac{i}{N}\cdot\left(\frac{N}{i} - 1\right) = \frac{1}{N}\sum_{i=B}^{N}1 - \frac{1}{N}\sum_{i=B}^{N}\frac{i}{N}.$$

The first summand equals $1 - \frac{B}{N}$. Since $\sum_{i=B}^{N} i \leq \sum_{i=1}^{N} i = \frac{1}{2}(N^2 - N)$, the negative term is at most $-\frac{1}{2} + \frac{1}{2N}$. The total relative error is thus at most $\frac{1}{2} - \frac{B-0.5}{N}$.

D.2 Analysis of Cutoff Count-Sketch

Here we rely on the following lemma, which is a slight modification of Theorem 4.3 from [1] (the proof is very similar and is omitted here).

Lemma D.1. *Suppose we use a CountSketch with B' buckets to store elements with true frequencies N/i, for $i = B, \ldots, N$. Then the expected additive error of CountSketch per item is $O\left(\frac{N}{B'} \log\left(1 + \frac{B'}{B}\right)\right)$.*

The absolute error of the algorithm of [16] is bounded in Section 4.4. The bound stated there is

$$\frac{1}{N} \sum_{i=B}^{N} \left| \tilde{f}_i - \frac{N}{i} \right| = O\left(\frac{N - B}{B'} \log\left(1 + \frac{B'}{B}\right) \right),$$

which becomes $O(N/B)$ if we choose $B \approx B'$.

The weighted error has already been bounded in [16, 1]. Since we use somewhat different notation and scaling (in particular, our Zipfian frequencies are N/i whereas theirs were $1/i$), let us repeat the calculation:

$$\frac{1}{N} \sum_{i=B}^{N} \frac{N}{i} \cdot \left| \tilde{f}_i - \frac{N}{i} \right| \leq O\left(\frac{N}{B'} \log\left(1 + \frac{B'}{B}\right) \right) \cdot \sum_{i=B}^{N} \frac{1}{i} = O\left(\frac{N}{B'} \log\left(1 + \frac{B'}{B}\right) \cdot \log\left(\frac{N}{B}\right) \right),$$

where we have used Theorem D.1. Choosing $B \approx B'$, the bound becomes $O(\frac{N}{B} \log(\frac{N}{B}))$.

For the relative error, again using Theorem D.1, we have:

$$\frac{1}{N} \sum_{i=B}^{N} \frac{i}{N} \cdot \left| \tilde{f}_i - \frac{N}{i} \right| \leq O\left(\frac{N}{B'} \log\left(1 + \frac{B'}{B}\right) \right) \cdot \frac{1}{N^2} \cdot \sum_{i=B}^{N} i = O\left(\frac{N}{B'} \log\left(1 + \frac{B'}{B}\right) \right),$$

which again becomes $O(N/B)$ if we choose $B \approx B'$.