Towards Identifying Frequent Items in Sliding Windows*

David DeHaan† Erik D. Demaine‡ Lukasz Golab†
Alejandro López-Ortiz† J. Ian Munro†

Technical Report CS-2003-06
March 2003

*This research is partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

†School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, {dedehaan, lgolab, alopez-o,imunro}@uwaterloo.ca

‡MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA, edemaine@mit.edu

Abstract

Queries that return a list of frequently occurring items are popular in the analysis of data streams such as real-time Internet traffic logs. While several results exist for computing frequent item queries using limited memory in the infinite stream model, none have been extended to the limited-memory sliding window model, which considers only the last N items that have arrived at any given time and forbids the storage of the entire window in memory. We present several algorithms for identifying frequent items in sliding windows, both under arbitrary distributions and assuming that each window conforms to a multinomial distribution. The former is a straightforward extension of existing algorithms and is shown to work well when tested on real-life TCP traffic logs. Our algorithms for the multinomial distribution are shown to outperform classical inference based on random sampling from the sliding window, but lose their accuracy as predictors of item frequencies when the underlying distribution is not multinomial.

1 Introduction

On-line data streams posses interesting computational characteristics, such as unknown or virtually unbounded length, possibly very fast arrival rate, inability to backtrack over previously arrived items (only one sequential pass over the data is permitted), and lack of system control over the order in which data arrive; see Babcock et al. for a survey of data management issues in stream processing [1]. A particular problem of interest concerns statistical analysis of data streams with a focus on newly arrived data and frequently appearing items. For instance, an Internet Service Provider may be interested in monitoring streams of IP packets originating from its clients and identifying users who consume the most bandwidth during a given time interval. These types of queries, in which the objective is to return a list of the most frequent items (called *top-k queries* or *hot list queries*) or items that occur above a given frequency (called *threshold queries*), are generally known as frequent item queries. However, to make such analysis meaningful, bandwidth usage statistics should be kept for only a limited amount of time (for example, one hour or a single billable period) before being replaced with new measurements. Failure to remove stale data leads to statistics aggregated over the entire lifetime of the stream, which are unsuitable for identifying recent usage trends.

A solution for removing stale data is to periodically reset all statistics. This gives rise to the *landmark window model*, in which a time point (called the landmark) is chosen and statistics are only kept for that part of a stream which falls between the landmark and the current time. Although simple to implement, a major disadvantage of this model is that the size of the window varies—the window begins with size zero and grows until the next occurrence of the landmark, at which point it is reset to size zero. In contrast, the *sliding window model* expires old items as new items arrive. In particular, *count-based* sliding windows are defined with respect to the last N items seen, while *time-based* windows include only those items which have arrived in the last t time units.

If the entire sliding window fits in memory, answering threshold queries over sliding windows is easy: we maintain frequency counts of each distinct item within the window and increment or decrement counters as new items arrive and old items are removed. However, there are many applications (e.g. monitoring Internet traffic on a high-speed link) where the stream arrives so fast that useful sliding windows are too large to fit in memory. In this case, each window must somehow be summarized and an answer must be approximated on the basis of the available summary information. This implies a trade-off between accuracy and space, which is one of the fundamental characteristics of all on-line stream algorithms.

1.1 Our Contributions

We begin by presenting a simple algorithm, FREQUENT, that identifies frequently occurring items in sliding windows conforming to an arbitrary distribution. We then describe three algorithms, MOREFREQUENTITEM, MOSTFREQUENTITEM, and OVERTHRESHOLD, for identifying frequent items within a stream adhering to a multinomial distribution in each instance of the sliding window. Our algorithms are based on the Basic Window idea of Zhu and Shasha, where the sliding window is divided into equally-sized Basic Windows for which only a small synopsis is stored [17]. MOREFREQUENTITEM and MOSTFREQUENTITEM—which store only the identity of the item that was most frequent in each Basic Window—identify the most frequent item, but estimating frequency or even bounding the error in the identification is shown to become infeasible as the number of item types grows. In contrast, OVERTHRESHOLD identifies all items over a specified threshold frequency and may be used for frequency prediction with bounded error dependent upon the allocated memory. We demonstrate that MOREFREQUENTITEM and OVERTHRESHOLD outperform classical inference based on random sampling in terms of identifying items over a fixed threshold. Finally, we test our algorithms on TCP connection logs and discover that algorithm FREQUENT works fairly well on bursty data, but the others become inaccurate when the assumption of an underlying multinomial distribution does not hold.

1.2 Roadmap

The remainder of this paper is organized as follows. Section 2 presents relevant previous work, Section 3 introduces algorithm FREQUENT, Section 4 describes algorithms MOREFREQUENTITEM and MOSTFREQUENTITEM, while Section 5 introduces algorithm OVERTHRESHOLD. Section 6 compares the prediction error of our algorithms with random sampling and inference for proportions, Section 7 contains experimental results based an a real-life Internet traffic log, and Section 8 concludes the paper with suggestions for future work.

2 Previous Work

2.1 Frequent Item Algorithms over Infinite Streams

Frequent item algorithms in the infinite stream model employ sampling, counting, and/or hashing to generate approximate answers using limited space. The main difficulty lies in finding a small set of potentially frequent items to monitor, while still being able to catch rarely occurring items that suddenly become frequent. In this context, approximation may mean a number of things: an algorithm may either return a set of items that contains all frequent flows (and some false positives), only some frequent flows (and some false negatives), identities of frequent flows but no frequency counts, or identities and approximate counts of the frequent items. Note that the terms *item types*, *item categories*, and *flows* are used synonymously throughout the remainder of the paper.

A naive counting method for answering threshold queries is to examine all items as they arrive and maintain a count for each item type to identify those that appear frequently. This method takes linear space in the worst case, where all item types are unique except for one type that occurs twice. On the other hand, reducing memory usage by random sampling may result in large variance when the sampled frequency is used as the estimator of the actual frequency. Estan and Varghese propose a sampled counting algorithm to determine a superset likely to contain the dominant flows [7]. This algorithm uses random sampling only to select whether an item is to

be examined more thoroughly; once an item is selected, all of its occurrences are counted (this idea also appears in Gibbons and Matias [10]). Another counting-sampling approximate algorithm is given by Manku and Motwani in [13], which uses a sampling rate that decreases with time in order to bound memory usage. Finally, a randomized counting-sampling algorithm is presented by Demaine et al. [6] that finds flows above a relative frequency of $1/\sqrt{nm}$ with high probability, where n is the number of incoming items observed and m is the number of available counters. This algorithm divides the stream into a collection of rounds, and for each round counts the occurrences of $m/2$ randomly sampled categories. At the end of each round, the $m/2$ winners from the current round are compared with $m/2$ winners stored from previous rounds and if the count for any current winner is larger than the count for a stored category, the stored list is updated accordingly.

Demaine et al. also present a counting algorithm that uses only m counters and deterministically identifies all categories having a relative frequency above $1/(m+1)$. This algorithm is a straightforward extension of the classical *majority* counting algorithm by Fischer and Salzberg [9]. This method, however, returns a superset guaranteed to contain popular items and requires a re-scan of the data to determine the exact set of frequent items. Moreover, Manku and Motwani also show a deterministic counting algorithm that maintains a counter for each distinct item seen, but periodically deletes counters whose average frequencies since counter creation time fall below a fixed threshold. To ensure that frequent items are not missed by repeatedly deleting and re-starting counters, each frequency estimate includes an error term that bounds the number of times that the particular item could have occurred up to now.

Fang et al. present various hash-based frequent item algorithms in [8], but each requires at least two passes over the data. Moreover, the one-pass sampled counting algorithm by Estan and Varghese may be augmented with hashing as follows. Instead of sampling to decide whether to keep a counter for a flow, we simultaneously hash each item's key to d hash tables and add a new counter only if all d buckets to which a particular element hashes are large (and if the element does not already have a counter). This reduces the number of unnecessary counters that keep track of rare flows. A similar technique is used by Charikar et al. in [4] in conjunction with hash functions that map each key to the set $\{-1, 1\}$.

2.2 Sliding Window Algorithms

Zhu and Shasha introduce the concept of *Basic Windows* in order to incrementally compute simple aggregates [17]. The sliding window is divided into equally-sized Basic Windows and only a synopsis and a timestamp are stored for each Basic Window. When the timestamp of the oldest Basic Window expires, that window is dropped and a fresh Basic Window is added. This scheme works well with statistics that are incrementally computable from a set of synopses. For example, we may incrementally compute the sum of all items inside the current sliding window by replacing the sum of all elements in the oldest Basic Window with the sum of all elements in the newest Basic Window. However, results are refreshed only after the stream fills the current Basic Window. If the available memory is small, then the number of synopses that may be stored is small and hence the refresh interval is large.

In order to solve the above problem, *Exponential Histograms* (EH) are introduced by Datar et al. in [5]. Given an error bound ϵ, the EH algorithm maintains Basic Windows with exponentially varying size such that the number of windows (and hence, the amount of memory needed for synopses) is optimal. Only a synopsis and a timestamp are stored for each Basic Window. However, in contrast to [17], the EH algorithm returns results to within an error ϵ at all times. The algorithm

requires $O(\frac{1}{\epsilon}\log^2 N)$ space, where N is the size of the window, and it is proven that this bound is optimal for counting within the allowed approximation error. Unfortunately, EH may only be used with synopses that are mergeable. That is, a synopsis for the union of two Basic Windows must be computable from the two individual synopses. Nevertheless, some statistics that do not obey the additive synopsis property (e.g. variance) may be re-written into an approximate incremental formula (see [3]).

Finally, random sampling from a window of size N is addressed by Babcock, Datar, and Motwani in [2]. Two algorithms are shown: *Chain Sampling* for count-based windows and *Priority Sampling* for time-based windows.

3 Motivation and Simple Algorithm for Arbitrary Distributions

The two existing techniques for computing statistics over sliding windows (Basic Windows and EH) cannot easily handle top-k queries. For example, if each Basic Window stores counts of the top five flows, we would ignore a frequent flow that consistently places sixth. Moreover, the fact that a flow type appears in a top-k synopsis in any one Basic Window does not mean that this flow is one of the k most frequent flows in the entire sliding window (it may not appear in any other Basic Windows at all). Likewise, the frequent item algorithms for infinite streams do not present obvious opportunities for extension to the sliding window model. The counters used in the counting methods could be split and a timestamp assigned to each sub-counter; this essentially reduces to the Basic Window method with item counts stored in the synopses. Similarly, hash tables could be split in the same way, resulting in a Basic Window approach with hash tables stored in the synopses. Space usage could be improved by incorporating periodic garbage collection to remove infrequent items or items which are about to expire. However, the side effects of the former are that infrequent items that suddenly arrive in large bursts and rise in frequency above the threshold may be missed, while the latter artificially narrows the sliding window.

We propose the following simple algorithm, FREQUENT, that employs the Basic Windows approach and stores top-k synopses in each Basic Window. We fix an integer k and for each Basic Window, maintain a list of the k most frequent items in this window (we assume that a single Basic Window fits in main memory, within which we may count item frequencies exactly). Let δ_i be the frequency of the kth most frequent item in the ith Basic Window. Then $\delta = \sum_i \delta_i$ is the upper limit on the frequency of a flow that does not appear on any of the top-k lists. Now, we sum the reported frequencies for each item present in at least one top-k synopsis and if there exists a flow whose reported frequency exceeds δ, we are certain that this flow has a true frequency of at least δ. The pseudocode is given below, assuming that b is the number of elements per Basic Window.

Algorithm FREQUENT
Repeat

1. For each element e in the next b elements
   If a local counter exists for the type of element e, increment it
   Otherwise, create a new local counter for this element type and set it equal to 1
2. Add a summary S containing the identities and counts of the k most frequent items to the back of queue Q
3. Delete all local counters
4. For each type named in S
   If a global counter exists for this type, add to it the count recorded in S

Otherwise, create a new global counter for this element type and set it equal to the count recorded in S

5. Add the count of the kth largest type in S to δ
6. If $sizeOf(Q) > N/b$ then
   (a) Remove the summary S' from the front of Q and subtract the count of the kth largest type in S' from δ.
   (b) For all element types named in S', subtract from their global counters the counts recorded in S'
   If a counter is decremented to zero, delete it
   (c) Output the identity and value of each global counter $> \delta$

Algorithm FREQUENT works with arbitrary underlying distributions, but is susceptible to the false negatives problem: there may be items that have appeared on a few top-k lists, but summing up their frequencies from these top-k lists does not exceed δ—however, some of these items may be sufficiently frequent in other Basic Windows (but not frequent enough to register on the top-k lists of these other windows) that their true frequency counts exceed δ. Moreover, if k is small, then δ may be very large and the algorithm will not be able to report any frequent flows. On the other hand, if k is large and each synopsis contains items of a different type (i.e. there are very few repeated top-k "winners"), the algorithm may require a great deal of storage space, perhaps as much as the size of the sliding window. We suspect that algorithm FREQUENT will work well if there are a few very frequent flows, which will repeatedly be included in nearly every top-k synopsis. We will verify this hypothesis experimentally in Section 7.

4 Identifying the Most Frequent Item: Multinomial Distribution

In this section, we present an algorithm for identifying the most frequent item in a sliding window that conforms to a multinomial distribution. We show that the algorithm works well when only two flows are present, but becomes computationally expensive as the number of flows increases. In the next section, we modify the algorithm to instead identify flows above a given threshold, and we show that solving this problem is significantly easier. We begin by considering count-based windows and extend our approach to time-based windows in Section 5.2.1. We continue to assume that reporting the most frequent item need not be available at all times (as in the Exponential Histogram approach) but instead a slight refresh delay is permitted (as in the Basic Window approach).

4.1 Two Flows

In the simplest case of only two flows, call them x and y, whose actual relative frequencies p_x and p_y sum to one, we wish to determine which of the two flows occurs in the window with higher frequency and give an estimate for that frequency. Using the Basic Window approach, a simple exact algorithm is to store counters that contain the difference of the number of x-items versus the number of y-items in each Basic Window. Summing the counters over all Basic Windows gives the difference in the observed counts, from which percentage frequencies may be obtained if the size of the sliding window is known. In what follows, we show that if the sliding window conforms to a binomial distribution, we need only record the identity of the more frequent flow in each Basic Window in order to estimate item frequencies.

4.1.1 A Simple Algorithm

The following algorithm divides the sliding window of size N into a set of n equally-sized Basic Windows, each of which is summarized by an entry in a queue. Statistics are refreshed every $b = N/n$ items.

Algorithm MoreFrequentItem

1. Initialize global counters f_x and f_y to zero
2. Repeat
   (a) Initialize local counters l_x and l_y to zero
   (b) For each element e in the next b elements
   If e is of type x, increment l_x
   Otherwise, increment l_y
   (c) Add a summary containing the type of the "winner" (larger local counter) to the back of queue Q, and increment the corresponding global counter
   (d) If $sizeOf(Q) > N/b$ then
      i. Remove the summary from the front of Q and decrement the corresponding global counter
      ii. Output the identity and value of the larger global counter

Since a single bit can identify the "winner" between two flows, MoreFrequentItem requires $O(N/b)$ space and $\Theta(1)$ amortized time.

Each time a Basic Window is filled, MoreFrequentItem outputs the identity of the item expected to be more frequent in the sliding window. Suppose that the output item is x. The algorithm also supplies a frequency f_x of the Basic Windows dominated by x. However, it is not immediately clear how f_x is related to the actual relative frequency p_x of item x.

Proposition 1 *Consider the random variable w defined as follows.*

$$w = \begin{cases} 1 & \textit{if } x \textit{ is the more frequent item in a Basic Window} \\ 0 & \textit{otherwise} \end{cases} \tag{1}$$

Then, w constitutes a Bernoulli variable.

Proof

The probability of success is the same for all Basic Windows as they all have the same size. Success occurs with probability B_x equal to the probability that flow x is more frequent than flow y within the Basic Window. That is, B_x is the probability that flow x occurs $\lceil \frac{b}{2} \rceil$ or more times in a Basic Window of size b, given by Equation (2); recall that flows x and y conform to a binomial distribution, where the probability that a given element e belongs to flow x is p_x and the probability that e belongs to flow y is $p_y = 1 - p_x$. Failure occurs with probability $1 - B_x$.

$$B_x = \sum_{i=\lceil \frac{b}{2} \rceil}^{b} \binom{b}{i} p_x^i (1 - p_x)^{b-i} \tag{2}$$

Corollary 1 *Since the probability of flow x winning in any one Basic Window is independent of its probability of winning in any other Basic Window, the sum of n Bernoulli variables w is a Binomial variable with parameters n and B_x.*

The frequency f_x output by MOREFREQUENTITEM may be used to calculate an observed relative frequency $\hat{B_x}$ that x is the winner of a Basic Window.

$$\hat{B_x} = f_x/n \tag{3}$$

This value can then be substituted in Equation (2) in order to obtain $\hat{p_x}$, the expected relative frequency of item x. Unfortunately, Equation (2) cannot be solved in closed-form for $\hat{p_x}$ (see Appendix A for partial results). Thus, numerical methods must be used in order to obtain a value for $\hat{p_x}$ for a given $\hat{B_x}$.

4.1.2 Bounding the Error

We will make use of the following result due to Hoeffding [11]. Consider a sample of n items from a Binomial distribution and an observed frequency of f. The following is Hoeffding's bound on the deviation of the observed frequency from the true frequency p.

$$\mathbf{Pr}\left\{\frac{f}{n} - p \geq \Delta\right\} \leq e^{-2n\Delta^2} \tag{4}$$

We assume that the numerical methods used to obtain $\hat{p_x}$ from $\hat{B_x}$ are not a significant source of error; therefore, the primary source of error stems from the quality of $\hat{B_x}$ as an estimate for B_x. Now, B_x is a Binomial random variable (by Corollary 1, f_x is a Binomial random variable, and B_x is simply a normalized form of f_x). Using the Hoeffding bound along with a symmetry argument gives the following.

$$\mathbf{Pr}\left\{(\hat{B_x} - \Delta) \leq B_x \leq (\hat{B_x} + \Delta)\right\} > 1 - 2e^{-2n\Delta^2} \tag{5}$$

The right-hand side is the confidence level, so by setting it equal to the desired confidence (e.g. 0.95) we can solve for Δ (note that n is fixed by the choice of b). Because B_x in Equation (2) increases monotonically with p_x, we can find lower and upper bounds for p_x by numerically computing solutions to Equation (2) for the points $B_x = (\hat{B_x} - \Delta)$ and $B_x = (\hat{B_x} + \Delta)$, respectively. This process is illustrated in Figure 1, showing B_x on the vertical axis and the bounds for p_x on the horizontal axis.

It should be noted that because the Basic Window size b occurs in the bounds of the summation in Equation (2), the choice of b has a large impact on the error in predicting p_x. As b increases, the following behaviour may be observed.

1. The prediction error Δ surrounding $\hat{B_x}$ increases because n, the number of Basic Windows used to make the prediction, decreases.

2. The graph of B_x vs. p_x degrades from a linear function to a step function centered around $p_x = 0.5$.

Figure 2 demonstrates the effect of changing b for a window of size $N = 10000$. It shows the curve $\hat{B_x}$ as a function of p_x, along with the curves $\hat{B_x} - \Delta$ and $\hat{B_x} + \Delta$ that bound the 95% confidence region. The three graphs demonstrate the following values of b: (a) 5 (b) 50 (c) 500.

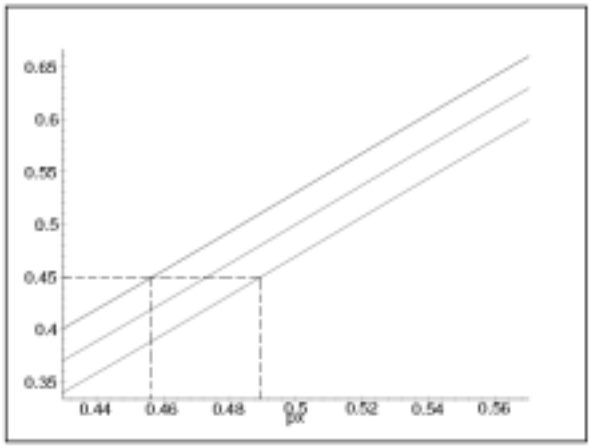
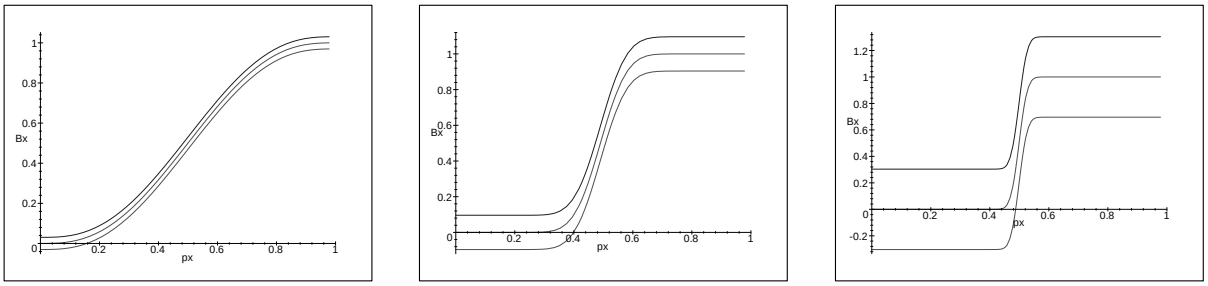


Figure 1: Solving for $\hat{p_x}$ numerically.



(a) (b) (c)

Figure 2: Effect of Basic Window size on inference error.

The observation that B_x as a function of p_x degrades to a step function with increasing b is crucial for characterizing the effect of Basic Window size on prediction error. For small values of b, algorithm MoreFrequentItem predicts a wide range of values for p_x, while for large values of b, the useful prediction range for p_x is very small. However, the prediction error immediately about the point $p_x = 0.5$ remains tight as b grows. The net effect is that as the choice of Basic Window size ranges from 1 to N, MoreFrequentItem's usefulness as a frequency predictor diminishes, but its accuracy as a Boolean test for identifying the majority item remains. Since the algorithm's space usage is inversely proportional to b, we conclude that there is a direct tradeoff between space and the accuracy of the frequency prediction, but the simple identification of the majority item does not illustrate this tradeoff.

4.2 Multiple Flows

Algorithm MoreFrequentItem from the previous section may be modified as follows to identify the most frequent item among d flows.

Algorithm MostFrequentItem
Repeat

1. For each element e in the next b elements
   If a local counter exists for the type of element e, increment it
   Otherwise, create a new local counter for this element type and set it equal to 1
2. Add a summary S containing the type of the "winner" (largest local counter) to the back of queue Q
3. Delete all local counters
4. If a global counter exists for the type named in S, increment it
   Otherwise, create a new global counter for this element type and set it equal to 1
5. If $sizeOf(Q) > N/b$ then
   (a) Remove the summary from the front of Q and decrement the corresponding global counter. Delete the counter if its size reaches zero
   (b) Output the identity and value of the largest global counter

The space requirement of algorithm MostFrequentItem consists of two parts: the working space needed to create a summary for the current Basic Window, and the storage space needed for the summaries of the Basic Windows. In the worst case, the working space requires $min(b, d)$ local counters of size $\log b$. For storage, there are N/b summaries each requiring $\log d$ bits. There are also at most N/b global counters of size $\log(N/b)$. This gives a total space bound of $O(\min(b, d)\log b + \frac{N}{b}\log d + \frac{N}{b}\log\frac{N}{b})$. The time complexity of MostFrequentItem is $O(b)$ for each pass through the outer loop. Since each pass consumes b arriving elements, this gives $O(1)$ amortized time per element.

The largest weakness of this algorithm lies in the intractability of using the output value f_i in order to estimate the relative frequency p_i of the most frequent item i. In fact, even just bounding the error on the identity of i is intractable for large d. Consider the case of three flows x, y, and z. In the two-flow case, B_x in Equation (2) was constructed by summing the probabilities of all possible cases where x was in majority within a Basic Window. These cases were easily identified as exactly those where x occurred at least $\lceil\frac{b}{2}\rceil$ times. However, in the three-flow case the test $count(x) \geq \frac{b}{3}$ is a necessary by not sufficient criterion for identifying a majority by x, because x's majority also depends on its count being greater than both y and z. This gives rise to the equation

$$B_x = \sum_{i=\lceil\frac{b}{3}\rceil}^{b} \binom{b}{i} p_x^i \left\{ \sum_{j=0}^{b-i} \binom{b-i}{j} p_y^j p_z^{b-(i+j)} - \sum_{j=i+1}^{b-i} \binom{b-i}{j} \left[p_y^j p_z^{1-(i+j)} + p_y^{1-(i+j)} p_z^j \right] \right\} \quad (6)$$

with analogous equations existing for B_y and B_z. In order to compute p_x given estimates for B_x, B_y and B_z, we must solve a non-linear system of two equations and two unknowns (the third equation is eliminated by rewriting p_z in terms of p_x and p_y).

In the general case of d flows, to estimate p_i we must solve a non-linear system of $d-1$ equations and $d-1$ unknowns, where the number of terms within each equation grows combinatorially in d. Even if we restrict the problem to simply bounding the prediction error in the identification of i as the most frequent item, we cannot translate the width of the Hoeffding-bounded error surrounding $\hat{B}_i$ to a range surrounding $\hat{p}_i$ without solving the entire system.

Because the *Most Frequent Item Problem* is a simplification of the more general *Top-k Problem*, the above results demonstrate that it is infeasible to extrapolate a solution to the top-k problem with bounded error using only a set of sub-solutions (top-k lists for portions of the total window) and the assumption of a multinomial distribution.

5 Threshold Queries: Multinomial Distribution

5.1 The Algorithm

The complexity involved in using algorithm MostFrequentItem is due to the interdependence among flows inherent in the concept of a winner for each Basic Window. Because of the dependencies involved in the creation of the stored synopses, we cannot use the synopses to solve for the relative frequency of one flow without simultaneously solving the entire system. Clearly, if we wish to solve for the frequencies of only selected flows, we must eliminate the inter-flow dependencies that exist within the stored synopses.

One way to introduce independence is to replace the concept of *winner* (implying comparison among peers) with *achiever* (implying comparison against an external standard). As a consequence, rather than each Basic Window resulting in exactly one winner, each Basic Window may result in the recognition of zero or more achievers. The following algorithm employs a user-defined threshold $1/m$ to create a synopsis for each Basic Window.

Algorithm OverThreshold
Repeat

1. For each element e in the next b elements
   If a local counter exists for the type of element e, increment it
   Otherwise, create a new local counter for this element type and set it equal to 1
2. Add a summary S containing the element types of all local counters $\geq b/m$ to the back of queue Q
3. Delete all local counters
4. For each type named in S
   If a global counter exists for this type, increment it
   Otherwise, create a new global counter for this element type and set it equal to 1
5. If $sizeOf(Q) > N/b$ then
   (a) Remove the summary S' from the front of Q
   (b) Decrement the global counters for all element types named in S'
   If a counter is decremented to zero, delete it
   (c) Output the identity and value of all global counters greater than some threshold τ

The space complexity of OverThreshold is at most m times worse than that of MostFrequentItem, with a worst case bound of $O(\min(b,d)\log b + \frac{mN}{b}\log d + \frac{mN}{b}\log\frac{N}{b})$ where d is the total number of flows in the system. The time complexity is $O(\min(m,b)+b)$ per iteration of the outer loop, which still yields $O(1)$ amortized time.

We now proceed to resolve two issues related to algorithm OverThreshold. Firstly, we identify the relation between the frequency f_x output by the algorithm and the true relative frequency of the flow p_x. Secondly, we investigate how to calculate the required value of τ used in step 6(c) of the algorithm. We first note that, as in section 4.1, we can define a Bernoulli random variable

$$w = \begin{cases} 1 & \text{if } count(x) \geq b/m \text{ in a Basic Window} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

whose probability of success B_x is given by the sum of the probabilities for all scenarios where x exceeds the threshold. In the construction of Equation (2) in section 4.1 we exploited the fact

that "majority" in the two flow case is equivalent to surpassing a threshold of $\lceil b/2 \rceil$. In this more general case of an arbitrary threshold, our new equation for B_x becomes the following.

$$B_x = \sum_{i=\lceil \frac{b}{m} \rceil}^{b} \left(\begin{array}{c} b \\ i \end{array} \right) p_x^i (1-p_x)^{b-i} \tag{8}$$

Observe that unlike Equation (6), this equation relates B_x to p_x without dependence on the relative frequencies of any other flows.

To address the first issue, note that each output f_x induces a value $\hat{B}_x = f_x/n$ which is an approximation for the true B_x of Equation (8). The frequencies $f_i \forall i = 1, \ldots, d$ are expected to follow a Multinomial distribution with parameters n, B_1, B_2, ..., B_d, so the marginal distribution for f_x (and hence $\hat{B}_x$) follows a Binomial distribution. Therefore, we can directly apply the Hoeffding bound from Equation (5) to quantify the error in this approximation. The result is that the observations made in section 4.1.2 regarding the effect of b on the shape of the curve and the error in prediction all directly apply, with the generalization that the step function centers around the point $p_x = 1/m$ rather than $p_x = 1/2$. Figure 3 demonstrates the curve associated with the values $N = 10000$, $b = 50$, and $m = 10$.

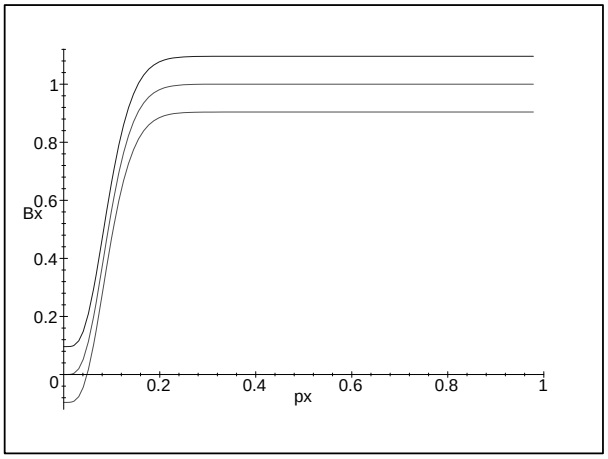


Figure 3: Effect of Threshold Parameter on Inference Error

The fact that the accuracy of frequency prediction centers around the relative threshold $1/m$ used to create the synopses implies that the desired reporting threshold should be very close to $1/m$ (or, more likely, $1/m$ should be chosen very close to the actual desired reporting threshold). Let us assume that $1/m$ is also the desired reporting threshold. Then, the value for τ should be the expected value for B_x when p_x equals the reporting threshold $1/m$, which can be calculated by substituting $1/m$ for p_x in Equation (8). This value for τ gives the most likely list of flows that have a relative frequency over $1/m$; however, the solution may contain either false negatives (high frequency flows not identified) or false positives (low frequency flows incorrectly identified). By adding (subtracting) the value Δ to (from) τ, we can guarantee with the confidence level associated with Δ that the solution contains no false positives (negatives), with the tradeoff that the solution is more likely to contain false negatives (positives).

5.2 Possible Extensions

5.2.1 Handling Time-based Sliding Windows

Our algorithms have been designed with count-based sliding windows in mind as the Basic Window approach requires uniformly sized divisions of the window. To extend our work to time-based windows, Basic Windows could be modified to allow different sizes that span equal time intervals. Our algorithms for the multinomial distribution would still work due to the following result from probability theory.

Theorem 1 *A Poisson trial a_i is a success with probability p_i and failure with probability $1 - p_i$. Suppose that A is the sum of n independent Poisson trials a_i with probabilities p_i for $1 \leq i \leq n$. Hoeffding's theorem states that A may be upper-bounded by a Binomial random variable B with parameters n and $p = \frac{1}{n}\sum_{i=1}^{n} p_i$.*

Unfortunately, Hoeffding's bound for the sum of Poisson trials is known to be (potentially much) looser than the bound on the sum of Bernoulli trials. Alternatively, we may use Chernoff's bound for Poisson trials (see, e.g. [15]).

5.2.2 Top-k Estimation using Counts

Recall that our algorithms compute lists of items that occur with frequencies exceeding a user-defined threshold. The following is a possible extension of our techniques to compute a list of the k most frequent items. Consider the general case of d distinct flows and some threshold τ. In addition to storing the boolean information of whether or not an item exceeded the threshold in a given Basic Window, we also store the counts of all the items above the threshold. After computing the list of all the items that exceed the threshold in the entire window, if there are more than k such items, then we increase the threshold slightly and eliminate all the items whose counts do not exceed the new threshold. We continue this procedure until there are exactly k items left.

The above suggests a more general approach of assigning different thresholds for different items. That is, for flows x, y, z and w, we could choose to include flow x on our above-the-threshold lists only if its relative frequency is above 0.4 and include other flows if their frequencies are above 0.35. This would be an appropriate strategy if we knew that flow x is slightly more popular than the other flows. This method could be improved by incorporating feedback from recent sliding windows and deciding whether to increase or decrease thresholds for various flows. This idea, however, is beyond the scope of this paper as it is more akin to probabilistic counting from data synopses than to frequent item queries.

5.2.3 Reducing Space Usage

We propose two extensions of our algorithms that aim at reducing space usage: randomly sampling items to be stored in the synopses and deleting parts of older synopses if a particular item has already exceeded the global threshold. In the first approach, if an item exceeds the threshold in a given Basic Window, we flip a biased coin and store the item with probability h and ignore it with probability $1 - h$. This scheme reduces space and does not affect the running time, but it does introduce an additional source of error. This demonstrates an interesting tradeoff between using space in order to straighten out the error curve (as in Figure 2, improving the range of p_x that can be predicted) and using space to tighten the prediction error within the usable part of the curve.

The second improvement essentially cuts down on redundant information and works as follows. Suppose that an item would have to occur on at least 20 out of 100 top-k lists in order to exceed a given threshold. Suppose further that flow x occurs on 60 such lists. If we removed every second occurrence of flow x from the top-k lists, we would still have 30 such occurrences and we would still conclude that x exceeds the threshold (although we could not even attempt an estimation of the true frequency of x). However, this would introduce error for skewed data as the window slides. A better solution would be to remove flow x from the 30 oldest lists on which it occurs, which does not introduce any error into future windows. This reduction in space comes at a cost of increased processing time to locate the oldest items to delete.

6 Comparison with Random Sampling

We are interested in comparing the accuracy in identifying high frequency flows between our algorithms (MoreFrequentItem and OverThreshold) and classical inference for proportions. Let $\hat{p}$ be the sample proportion (observed count divided by the sample size n). The interval within which the true proportion p lies may be calculated as follows.

$$p \in [\hat{p} - z^* S, \hat{p} + z^* S] \tag{9}$$

The value of z^* is the percentile of the standard normal distribution that corresponds to the given confidence level ($z^* = 1.960$ for 95% confidence, $z^* = 2.576$ for 99% confidence), while S is the standard error of the sample given by the following equation.

$$S = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \tag{10}$$

This inference method relies on the normal approximation to Binomial distributions and should be used only if $np \geq 5$ and $n(1-p) \geq 5$. Moreover, we introduce the finite population correction factor, which is used when sampling is performed from a finite population. In this scenario, the population size is equal to the sliding window size because we have assumed that each sliding window conforms to a multinomial distribution. With the correction for finite population, assuming sample size n and sliding window (population) size N, the standard error becomes the following.

$$S = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \sqrt{\frac{N-n}{N-1}} \tag{11}$$

In our experiments, the error metric is taken to be the maximum expected error when the sample proportion is equal to the threshold at the 95% confidence level. For instance, in the two-flow majority case, we compare the range of p that MoreFrequentItem returns when $\hat{B}_x = 0.5$ with the confidence interval predicted by Equations (9) and (11) for $\hat{p} = 0.5$. We fix N, the size of the sliding window, to be 10000 and investigate the consequences of increasing the Basic Window size b, (or equivalently, decreasing k, the number of Basic Windows). To ensure fairness, we allow the random sampling algorithm to utilize the same amount of memory that our algorithms require in the worst case. Furthermore, we undercharge the random sampling algorithm by ignoring the space costs associated with maintaining a windowed random sample (see Babcock, Datar, and Motwani [2] for more details regarding these costs).

6.1 Performance of Algorithm MoreFrequentItem

We begin the performance comparison by considering algorithm MoreFrequentItem in the role of an identifier of the majority between two flows. Figure 4 shows the error as a function of b. The upper curve corresponds to classical inference, the lower curve to MoreFrequentItem.

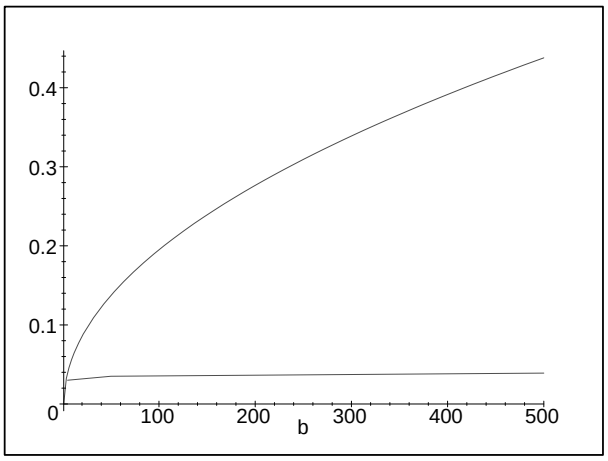


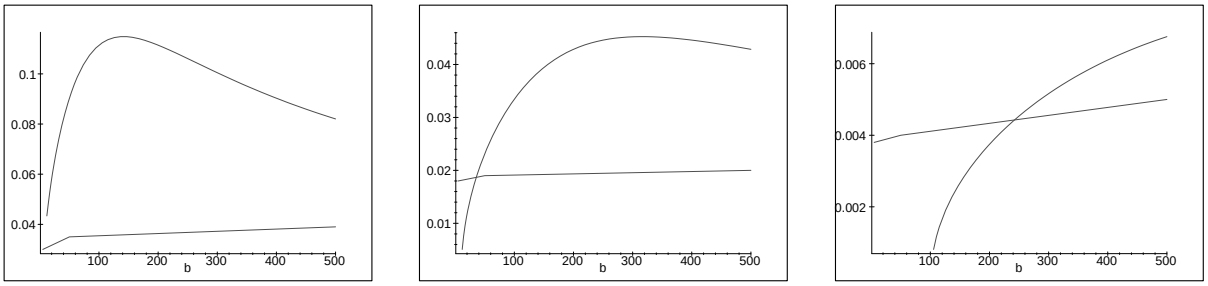
Figure 4: Prediction Error of MoreFrequentItem and Random Sampling

It can be seen that MoreFrequentItem outperforms classical inference for all values of b. For instance, if $b = 100$, the algorithm's error is only one-fifth of the error in random sampling. As the value of b approaches 400, our algorithm's advantage in minimizing the error reaches one order of magnitude. As seen in Figure 2 in Section 4, increasing b has little effect on the approximation error of MoreFrequentItem at the decision point, while at the same time reducing the space requirements (and unfortunately, increasing the refresh delay). In contrast, random sampling performs increasingly poorly as b gets large because the ratio of the sample size to the population size decreases. We conclude that MoreFrequentItem is the superior algorithm for the examined parameters.

6.2 Performance of Algorithm OverThreshold

We now compare the worst-case performance of algorithm OverThreshold with classical inference for many flows with three threshold values: 0.5, 0.1, and 0.01. The value of N remains fixed at 10000 and the confidence level is still 95%. We assume that the number of distinct flows d is at least as large as b. Results are shown in Figure 5 for threshold values of (a) 0.5 (b) 0.1 and (c) 0.01. The (approximately) linear function is the error of OverThreshold, while the sharp curve corresponds to the error of random sampling.

We first note that the error in classical inference is no longer a monotonically increasing function of b. This is so because the space complexity of algorithm OverThreshold depends on b (in the worst case, we need to store the entire current Basic Window in memory since we assumed that $d \geq b$) and on $\frac{mN}{b}$ (the number of synopses stored times the maximum number of items that may possibly exceed the given threshold of $\frac{1}{m}$). Thus, as b increases, our algorithm must allocate more working storage for the current Basic Window, which allows the classical inference algorithm to use



(a) (b) (c)

Figure 5: Prediction Error of OverThreshold and Random Sampling

a larger sample size. This explains why the error in random sampling eventually begins to decrease as b increases, as seen in Figures 5(a) and 5(b).

Our second observation deals with the degradation in the worst-case performance of algorithm OverThreshold (relative to random sampling) for very small threshold values. Clearly, a smaller threshold value allows more items to exceed the threshold in a given Basic Window, thereby increasing the upper bound on the sizes of our synopses. Nevertheless, as seen in Figure 5(a), our algorithm outperforms random sampling when the threshold is 0.5 and b is smaller than roughly 2500 (i.e. one quarter of the sliding window size). In Figure 5(b), we see that when the threshold is lowered to 0.1, our algorithm performs better for b ranging between approximately 25 and 2500. In Figure 5(c), further decreasing the threshold to 0.01 leads to a value of b ranging roughly between 250 and 2500 for which algorithm OverThreshold is more precise than random sampling.

It should be noted that these results represent the worst-case behaviour of algorithm OverThreshold, where the maximal number of items that could exceed the threshold in a given Basic Window do and must be recorded in the synopses, and where there are at least b distinct items in the sliding window. Relaxing either of these conditions leads to a relative improvement in the performance of our algorithm versus random sampling. In the "best" case of only two flows, we only require one counter in order to decide which flow was more frequent within the current Basic Window. Thus, the amount of memory available to store a random sample is smaller and our algorithm enjoys a greater relative advantage (see Figure 4).

7 Experimental Results

7.1 Experimental Setup

We now test our algorithms on Internet traffic data obtained from the Internet Traffic Archive [12]. We use a trace that contains all wide-area TCP connections between the Lawrence Berkeley Laboratory and the rest of the world between September 16, 1993 and October 15, 1993. The trace was created by Paxson and analyzed by Paxson and Floyd in [16]. The total number of connections in the trace is 782279, with the ten most popular protocol types shown in Table 1. We consider each of the 53 protocol types present in the trace to be a distinct item type. In each experiment, we randomly choose one hundred starting points for windows, each of size 10000 items, and execute our algorithms over those windows. For comparison, we also run a brute-force algorithm that calculates the true frequencies of all item types.

Protocol type	# of connections	% of connections
smtp	272643	34.9
nntp	148498	19.0
ftp-data	112891	14.4
telnet	89238	11.4
domain	33402	4.3
ftp	32872	4.2
finger	24901	3.2
gopher	11587	1.5
www	9070	1.2
printer	7755	1.0
other	39422	5.0
TOTAL	782279	100.0

Table 1: Ten most popular protocols in the TCP trace used in our experiments.

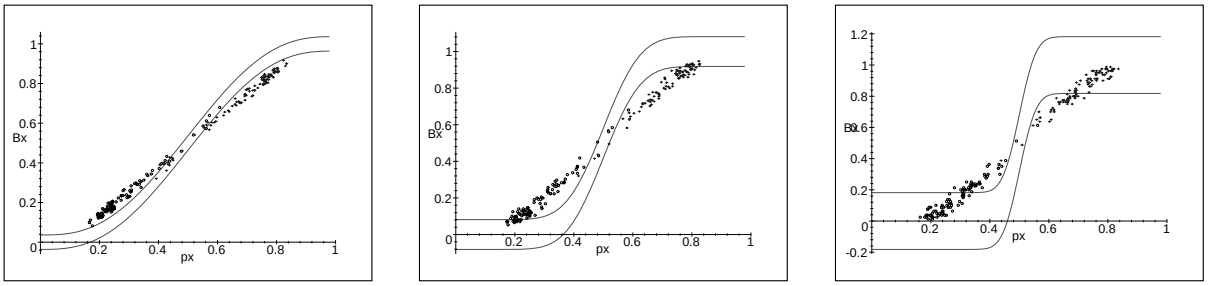
k	Avg. count of kth item	Avg. # correct	Avg. # false negatives
1	11.3	0	0
2	5.7	1.1	0.3
3	3.5	2.4	0.2
4	2.0	3.5	0.2
5	1.2	4.5	0.1
6	0.76	5.6	0.8

Table 2: Experimental results for algorithm FREQUENT

7.2 Performance of Algorithm Frequent

We begin by testing algorithm FREQUENT over sliding windows composed of four hundred Basic Windows of size 25. We vary k, the size of the synopses, from one to six. We measure the average count of the kth most frequent item per Basic Window (by dividing δ by the number of Basic Windows), the average number of flows that were correctly identified by our algorithm as being over the threshold δ, and the average number of false negatives. Results are shown in Table 2.

As expected, the average count (per Basic Window) of the kth most frequent item drops as k increases. In fact, when $k = 6$, the sixth most frequent item's frequency is below one, meaning that in some Basic Windows, there are only five distinct types and the sixth type has frequency zero (ties are broken arbitrarily). Now, if we only store the winner in each Basic Window ($k = 1$), δ is large and we cannot reliably identify any popular items. There are also no false negatives because there are actually no flows whose frequency exceeds δ. Since the average count of a winner is 11.3 and there are 25 items per Basic Window, a flow would have to occur with frequency 11.3/25, i.e. over 45% of the time. However, as seen in Table 1, the most frequent flow occurs with frequency of 35%. When $k = 2$, there are either one or two flows that exceed 5.7/25, or 23% frequency and our algorithm almost always identifies the most frequent flow. There may be occasional false negatives. When $k = 3$, there are usually two or three items that exceed δ and again, our algorithm always



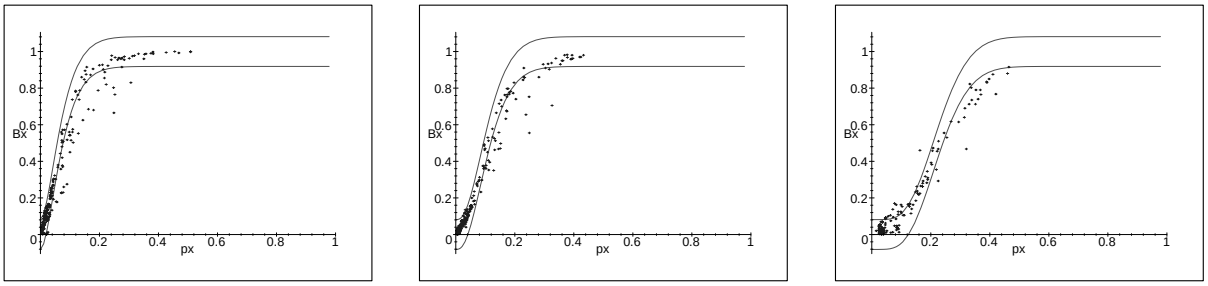
(a) (b) (c)

Figure 6: Accuracy of algorithm MoreFrequentItem with our experimental data.

correctly identifies the two most frequent flows. In some cases, the third flow is also identified and in others, it is not reported. When $k = 4$, there are three, four, and sometimes five flows that now exceed δ. At least the top three are always identified and occasionally, the fourth or fifth flows are left out. When k reaches five, the average count of the fifth most popular item in a Basic Window is just over one and there are very few false negatives: our algorithm correctly identifies the top four and sometimes the top five flows in the correct order. However, increasing k to six means that the sixth most frequent item sometimes has frequency zero and there are now up to eight item types whose frequency exceeds δ. In this case, the probability of false negatives increases and our algorithm may not identify frequent flows in correct order, i.e. a flow may have a lower reported frequency than another, but a higher true frequency. We conclude that algorithm Frequent works surprisingly well in identifying the heaviest flows within data resembling a power law distribution (i.e. several heavy flows and many very light flows). We note, however, that the algorithm would perform much worse when faced with a distribution that is more uniform.

7.3 Performance of Algorithm MoreFrequentItem

We have edited the trace and retained only two protocols (smtp and ftp-data) in order to test algorithm MoreFrequentItem. Results are shown in Figure 6. The two curves represent the lower and upper frequency prediction ranges for 99% confidence, while the crosses and circles represent the values of B_x and the corresponding true values of p_x of the winning and losing flow, respectively. Figure 6 (a) corresponds to bucket size of five, Figure 6 (b) to bucket size of 25, and Figure 6 (c) to bucket size of 125. In all cases, we see that the S-shaped prediction curve of our algorithm (which assumes an underlying binomial distribution) does not match the approximately linear prediction curve of the experimental data. In particular, the winning flow's frequency is consistently underestimated (the observed values lie to the right of the prediction curves) because of the burstiness of the data. That is, if a flow wins in a particular Basic Window, it might occur in this window exclusively. On the other hand, the frequency of the losing flow is often overestimated because the losing flow may not occur at all in the Basic Windows in which it does not win. We conclude that algorithm MoreFrequentItem should be used (especially as a frequency predictor) only if it is known that the underlying distribution is (or may be approximated as) binomial.



(a) (b) (c)

Figure 7: Accuracy of algorithm OverThreshold with our experimental data.

7.4 Performance of Algorithm OverThreshold

Results of experiments with algorithm OverThreshold are shown in Figure 7. The two curves represent the lower and upper frequency prediction ranges for 99% confidence and the crosses represent the actual (p_x, B_x) pairs generated from the test data. We fix the Basic Window size at 25 and test threshold values of 0.08 (Figure 7 (a)), 0.12 (Figure 7 (b)), and 0.24 (Figure 7 (c)). As before, in many cases the frequencies of the heavy flows are underestimated due to the burstiness of the data. Notably, some of the lighter flows are not as bursty as the most popular types and are well approximated by our multinomial algorithm. Again, we conclude that algorithms MoreFrequentItem and OverThreshold should not be used if the underlying data cannot be approximated by a multinomial distribution.

8 Conclusions

We presented algorithms for computing threshold queries over sliding windows using limited memory. We considered the general case, in which item types conform to an arbitrary distribution and presented a simple algorithm that works well with real data that are bursty and contain a small set of very popular item types. We also narrowed down our focus to data conforming to a multinomial distribution and devised algorithms for answering threshold queries (and to some extent for inferring the actual frequencies of items) in this model. These algorithms were later shown to outperform classical inference from a windowed random sample, but turned out to perform poorly with bursty data.

Our future work includes analyzing algorithm Frequent and proving an upper bound on the size of the top-k synopses required to guarantee a certain level of precision. If the underlying data conform to a power law distribution, we suspect a correlation between k and the power law coefficient. Moreover, this work may also be considered as a first step towards solving the more general problem of reconstructing a probability distribution of a random variable given only an indication of its extreme-case behaviour.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. Models and Issues in Data Streams. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 2002.

[2] B. Babcock, M. Datar, R. Motwani. Sampling from a Moving Window over Streaming Data. In *Proc. 13th SIAM-ACM Symposium on Discrete Algorithms*, 2002.

[3] B. Babcock, M. Datar, R. Motwani, L. O'Callaghan. Sliding Window Computations over Data Streams. Technical Report, Stanford University, April 2002.

[4] M. Charikar, K. Chen, M. Farach-Colton. Finding frequent items in data streams. In *Proc. 29th International Colloquium on Automata, Languages and Programming*, 2002.

[5] M. Datar, A. Gionis, P. Indyk, R. Motwani. Maintaining Stream Statistics over Sliding Windows. In *Proc. 13th SIAM-ACM Symposium on Discrete Algorithms*, 2002.

[6] E. Demaine, A. Lopez-Ortiz, J. Ian Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In *Proc. 10th European Symposium on Algorithms*, 2002.

[7] C. Estan, G. Varghese. New Directions in Traffic Measurement and Accounting. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, 2001.

[8] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J. Ullman. Computing Iceberg Queries Efficiently. In *Proc. 24th Int. Conf. on Very Large Databases*, 1998.

[9] M. Fischer and S. Salzberg. Finding a majority among N votes: Solution to problem 81-5 (Journal of Algorithms, June 1981). In *Journal of Algorithms*, 3(4):362–380, December 1982.

[10] P. Gibbons, Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *Proc. ACM Int. Conf. on Management of Data*, 1998.

[11] W. Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. In *American Statistical Association Journal*, 58:13–30, 1963.

[12] The Internet Traffic Archive. http://ita.ee.lbl.gov/index.html.

[13] G. Manku, R. Motwani. Approximate Frequency Counts over Data Streams. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002.

[14] Maple Version 8, http://www.maplesoft.com.

[15] R. Motwani, P. Raghavan. Randomized Algorithms. Cambridge University Press, 1995.

[16] V. Paxson, S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. In *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[17] Y. Zhu, D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002.

Appendix

Relating B_x and p_x

The following calculations have been performed in Maple 8.00 [14]. Here, we show that the closed form solution of Equation (2) is impractical to compute. We begin by restating Equation (2), that is, the probability of flow x winning a particular Basic Window in the case of only two flows.

$$B_x = \sum_{i=\lceil \frac{b}{2} \rceil}^{b} \left(\begin{array}{c} b \\ i \end{array} \right) p_x^i (1-p_x)^{b-i} \tag{12}$$

Solving the summation in Equation (12), we obtain

$$B_x = \left(\begin{array}{c} b \\ \lceil \frac{b}{2} \rceil \end{array} \right) p_x^{\lceil \frac{b}{2} \rceil} (1-p_x)^{b-\lceil \frac{b}{2} \rceil} H\left([\lceil \frac{b}{2} \rceil - b, 1], 1 + \lceil \frac{b}{2} \rceil, \frac{p_x}{p_x - 1} \right) \tag{13}$$

where the generalized Hypergeometric function $H([n_1, ..., n_j], [d_1, ..., d_m], z)$ is defined as

$$H(\mathbf{n}, \mathbf{d}, z) = \sum_{k=0}^{\infty} \frac{\prod_{i=1}^{j} \frac{\Gamma(n_i+k)}{\Gamma(n_i)} z^k}{\prod_{i=1}^{m} \frac{\Gamma(d_i+k)}{\Gamma(d_i)} k!} \tag{14}$$

where the Gamma function $\Gamma(z)$ is

$$\Gamma(z) = \int_0^{\infty} e^{-t} t^{z-1} dt \tag{15}$$

Maple is unable to analytically solve for p_x in Equation (12). This is also the case for multiple flows and an arbitrary threshold—the only difference is that $\lceil \frac{b}{2} \rceil$ is replaced by τb where $0 \leq \tau \leq 1$ is the threshold.