Fundamentals of Analyzing and Mining Data Streams

Graham Cormode

AT&T Labs–Research, 180 Park Avenue, Florham Park, NJ 07932, USA

Abstract. Many scenarios, such as network analysis, utility monitoring, and financial applications, generate massive streams of data. These streams consist of millions or billions of simple updates every hour, and must be processed to extract the information described in tiny pieces. This survey provides an introduction the problems of *data stream monitoring*, and some of the techniques that have been developed over recent years to help mine the data while avoiding drowning in these massive flows of information. In particular, this tutorial introduces the fundamental techniques used to create compact summaries of data streams: sampling, sketching, and other synopsis techniques. It describes how to extract features such as clusters and association rules. Lastly, we see methods to detect when and how the process generating the stream is evolving, indicating some important change has occurred.
Keywords: data streams, sampling, sketches, association rules, clustering, change detection.

1 Introduction

In recent years there has been growing interest in the study and analysis of *data streams*: flows of data that are so large that it is usually impractical to store them completely data. Instead, they must be analyzed as they are produced, and high quality results guaranteed, no matter what outcomes are observed as the stream progresses. This tutorial surveys some of the key ideas and techniques that have been developed to analyze and mine such massive data streams. See [13] for a longer survey from an algorithmic perspective.

Motivation for studying data streams comes from a variety of areas: scientific data generation, from satellite observation to experiments on subatomic particles can generate terabytes of data in short amounts of time; sensor networks may have many hundreds or even thousands of nodes, each taking readings at a high rate; and communications networks generate huge quantities of *meta-data* about the traffic passing across them. In all cases, this information must be processed and analyzed for a variety of reasons: to monitor a system, analyze an experiment, or to ensure that a service is running correctly. However, given the massive size of the input, it is typically not feasible to store it all for convenient access. Instead, we must operate with resources much smaller than the size of the input ("sublinear"), and still guarantee a good quality answer for particular computations over the data.

From these disparate settings we can abstract a general framework within which to study them: the streaming model. In fact, there are several variations of this model, depending on what form the input may take and how an algorithm must respond.

Models: Arrivals only, or Arrivals and Departures. The basic model of data streams is an arrivals-only one. Here, the stream consists of a quantity of tuples, or items, which describe the input. Typically each tuple is a simple, small object, which might indicate, for example, the identity of a particular object of interest, and a weight or value associated with this arrival. In a network, the observation of a packet could be interpreted as a tuple indicating the intended destination of the packet, and the size of the packet payload in bytes. For another application, the same packet could be interpreted as a tuple whose identity is the concatenation of the source and destination of the packet, with a weight of 1, indicating that it is a single packet. Typically, we can interpret these streams as defining massive implicit vectors, indexed by item names, and whose entries are (usually) the sum of the associated counts (although many other interpretations may be possible). A richer model allows departures: in additional to positive updates to entries in this implicit vector, they may be negative. This captures more general situations in which earlier updates might be revoked, or observations for which negative values are feasible. In either case, the assumption is that each tuple in the input stream must be processed as it is seen, and cannot be revisited later unless it is stored explicitly by the stream algorithm within its limited internal memory.

Randomization and Approximation. Within these models, many natural and fundamental questions can be shown to require space linear in the input to answer exactly. For example, to test whether two separate

streams are the same (i,e. they encode the same number of occurrences of each item) requires us to store space linear in the number of distinct items, which could be immense. To be able to make progress, we typically allow *approximation*: returning an answer that is correct within some small fraction, ϵ of error; and *randomization*: allowing our algorithms to make random choices and to fail with some small probability δ. Algorithms which use both randomization and approximation we refer to as (ϵ, δ) approximations.

Update time, query time and space usage. To evaluate algorithms that operate on streams, we typically look at their behavior with respect to three additional features:

- *Update time:* the time to process each stream update.
- *Query time:* the time to use the information stored to answer the question of interest.
- *Space Usage:* the amount of memory used by the algorithm to keep information.

Typically, these three are measured in terms of parameters of the stream: the number of tuples, n and the number of different items m; and the parameters ϵ and δ. To be an effective streaming algorithm these measures, particularly the space used, should be sublinear in m and n, and ideally poly-logarithmic (i.e. $O((\log m \log n)^c)$ for some constant c.

2 Streaming, sketches and summaries

In this section we outline two fundamental approaches to coping with streaming data: drawing a representative sample, and creating a compact "sketch" of the stream.

2.1 Random Sampling: reservoir and minwise

Many mining algorithms can be applied if only we can draw a representative sample of the data from the stream. The question is, how to ensure such a sample is drawn uniformly, given that the stream is continuously growing? For example, if we want to draw a sample of 100 items and the stream has length only 1000, then we want to sample roughly one in ten items. But if a further million items arrive, we must ensure that the probability of any item being sampled is more like one in a million. If we retain the same 100 items, then this is very skewed to the prefix of the stream, which is unlikely to be representative.

Several solutions are possible to ensure that we continuously maintain a uniform sample from the stream. The idea of reservoir sampling dates back to the eighties and before [15]. It is easiest to describe if we wish to draw a sample of size 1. Here, we initialize the sample with the first item from the stream. We replace the current sample with the ith item in the stream (when it is seen) by throwing some random bits to simulate a coin whose probability of landing "heads" is $1/i$, and keeping the ith item as the sampled item if we observe heads. It is a simple exercise to prove that, after seeing n items, the probability that any of those is retained as the sample is precisely $1/n$. One can generalize this technique to draw a sample of k items, either with replacement (by performing k independent repetitions of the above algorithm) or without replacement (by picking k items and replacing one with the ith with probability $1/i$).

A drawback of this approach is it does not easily generalize when we have many, distributed streams, and wish to sample uniformly from their union. For example, consider trying to sample from a stream formed by network traffic crossing the Atlantic and Pacific oceans. It is not feasible to operate jointly on both streams. Instead we use an alternative sampling algorithm, which we refer to as "min-wise sampling" (by analogy with an alternate technique known as min-wise hashing [3]). For each item in the stream we pick a random label as a real number in the range 0 to 1. We retain the item with the smallest random label seen so far. It is straightforward to observe that each item has an equal chance of getting the smallest tag, due to the symmetry of the procedure, and therefore it picks uniformly from the stream. Moreover, we can run the same algorithm across distributed streams and merge the results to get an item picked uniformly from the (disjoint) union of the streams, by picking the retained item with the smallest label.

Application: Estimating Entropy. The empirical entropy of a sequence of characters is computed by finding the number of occurrences, f_i, for each character i and computing $H = \sum_{i=1}^{m} \frac{f_i}{n} \log_2 \frac{n}{f_i}$. This entropy is often used in network monitoring applications to detect anomalies. When the number of possible

items m is very large, we need a different approach to approximate the entropy. We build an estimator for entropy as follows based on sampling a position j in the stream (using the above min-wise sampling), and counting the number of subsequent occurrences in the stream of the character at position j as r. We can build an unbiased estimate of H as $r \log \frac{m}{r} - (r-1) \log \frac{m}{r-1}$. This estimate is not very reliable; it can be improved by taking the average of many repetitions using different random samples. This can be shown to give an $(\frac{\epsilon}{H^2}, \frac{1}{4})$ estimator; by taking the median of $O(\log 1/\delta)$ repetitions we form an $(\frac{\epsilon}{H^2}, \delta)$ estimator. This works well when H is large, but H can be very small, which results in a less reliable estimator. Further modifications of this technique can be used to generate an (ϵ, δ) estimator; see [4] for details.

2.2 Sketches for Estimation

Many data stream problems cannot be solved with just a sample. Instead, we can make use of data structures which, in effect, include a contribution from the entire input, rather than just the items picked in the sample. For example, consider trying to count the number of distinct objects in a stream. It is easy to see that unless almost all items are included in the sample, then we cannot tell whether they are the same or distinct. Since a streaming algorithm gets to see each item in turn, it can do better, as we shall see later. We refer to a "sketch" as a compact data structure which summarizes the stream for certain types of query. Typically it is a linear transformation of the stream: we can imagine the stream as defining a vector, and the algorithm computes the product of a matrix with this vector (to be effective, the matrix must have a very small representation, e.g. being defined implicitly by hash functions). We highlight three popular sketch algorithms:

Count-Min Sketch. The count-min sketch [6] is an array of counters of size $\frac{2}{\epsilon} \times \log \frac{1}{\delta}$, and $\log \frac{1}{\delta}$ hash functions. Each update is mapped to $\log \frac{1}{\delta}$ counters, one in each row, which are incremented to reflect the update. From this data structure, one can estimate the frequency f_i of any item, with error at most ϵn with probability at least $1 - \delta$, in space $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$.

Flajolet-Martin Sketch. The Flajolet-Martin sketch [10] is a bitmap of length approximately $\log m$. Each item is mapped by a hash function into an entry of the bitmap: with probability $\frac{1}{2}$ it maps into entry 1, $\frac{1}{4}$ to entry 2, $\frac{1}{8}$ to entry 3 and so on. For each item in the stream, we map to its bit under the hash function, and set the bit to 1. The position of the least significant 0 in the bitmap indicates the logarithm of the number of distinct items seen, D; taking repetitions with randomly chosen hash functions improves the accuracy. Space $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ is sufficient for an (ϵ, δ) approximation of D.

AMS Sketches. The Alon-Matias-Szegedy sketch [2] can be described in terms of the Count-Min sketch. Now, when we go to update a counter, we multiply the value of the update by a hash function g on the item being updated: half the items are mapped to $+1$ by this hash function, and half to -1. Taking the sum of the squares of all counters in each row gives a high-quality estimate for $F_2 = \sum_{i=1}^{m} f_i^2$, the sum of the squares of the frequency counts. This computation, or variations thereof, is at the heart of many data stream analyses. An (ϵ, δ) approximation for F_2 can be formed in space $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$.

A common feature of these sketch algorithms is that they rely on hash functions on item identifiers, which are relatively easy to implement and fast to compute. Indeed, many practical streaming data management systems implement such sketches, such as Sprint's CMON system [14] and AT&T's Gigascope [8], both of which operate on network data streams at gigabit speeds. Implementations of sketches can be found on the web, including `http://www.cs.rutgers.edu/~muthu/massdal-code-index.html`.

3 Stream Data Mining Algorithms

Building on ideas of sampling and sketching, we can design algorithms for specific analysis and data mining tasks. We discuss three popular problems: association rule mining, change detection, and clustering.

3.1 Association Rule Mining

A classic problem in data mining is Association Rule Mining [1]. Given a large collection of transactions t_i, each of which is a subset of possible items, for example sets of items bought from a supermarket, the

goal is for find rules of the form $X \rightarrow y$. The support of the rule is the fraction of the input which contains all members of the rule, i.e. $|\{i|X \cup \{y\} \subseteq t_i\}/|\{t_i\}|$. The confidence of the rule is the number of input transactions which contains all members of the rule divided by the number containing the conditions (left side), i.e. $|\{i|X \cup \{y\} \subseteq t_i\}/|\{t_i|X \subseteq t_i\}$. In general, one seeks to find all rules with support and confidence both exceeding specific thresholds. There are exponentially many possible rules, and so careful strategies are designed to search through them efficiently. Typically, the problem is reduced to one of finding all *frequent itemsets*: subsets of items with high support (above some threshold ϕ). From these itemsets, the association rules can be determined.

Clearly this problem is especially challenging when the input transactions are observed in a streaming fashion, and limited resources are available to process them. Indeed, even the question of finding the frequent 1-itemsets (sets of size 1) — a necessary precursor to solving the general problem — is a challenge when the set of possible items is large, and has attracted significant interest. Sketching techniques as outlined above can be applied, but here we describe deterministic (non-randomized) approaches. The SpaceSaving algorithm [12] shows that the problem can be $(\epsilon, 0)$ approximated using space $O(\frac{1}{\epsilon})$. It tracks a set of $k = 1/\epsilon$ items and associated counts. For each item, if it has an associated counter then the counter is incremented; else, the item replaces the item with the smallest count, and that count is incremented. It can be shown that this simple algorithm gives the desired accuracy, and can be implemented efficiently.

Given ways to find frequent items, they can be extended to frequent itemsets. The method outlined by Manku and Motwani [11] attempts to use the available space as fully as possible. For each new transaction, it generates all the subsets, and stores them in a compact trie-based structure. When the space is full, it uses a pruning algorithm based on frequent items algorithms to delete the least frequent itemsets, and track the error in the estimated counts of each item. This gives an efficient and somewhat scalable solution, although in general there is no convenient non-trivial worst case bound on the space required for a given accuracy. Many variations of the problem have been studied, based on finding itemsets which correspond to ordered subsequences, or sequential patterns (substrings) of the input transactions.

3.2 Change Detection

As we are monitoring a stream of values, a fundamental question is "has the distribution of values changed recently?". We want to know if things have changed so that we can detect anomalies — some deviation from what is expected — and trigger an alert if it has. It can tell us if there has been some problem with a data feed which has caused the distribution to shift. If we have built some data mining algorithm based on a particular model, a change indicated that the model may no longer be valid and we need to rebuild. But what is a "change"? It can be the change in behaviour (frequency) of some subset of items, or a change in other patterns. Here, we take a definition where the underlying distribution (of frequencies) changes. We aim to do this non-parametrically: that is, without explicitly fixing a model that we expect the data to fit.

Dasu *et al.* propose a technique based on statistical bootstrapping to identify when a change has occurred [9]. They consider the case when the input consists of a series of points from a high dimensional space (value-based or categorical). Because we do not expect to see the exact same points many times, instead we use a space-partitioning algorithm over a "reference window" to define regions, and compute the relative frequencies within each region: a set of empirical probabilities $p(i)$ for the reference window and $q(i)$ for the sliding window. This is applied both to a fixed reference window, and a sliding window, both of size n points. To test for change, they compute the Kullback-Leibler distance (KL) as $D(p||q) = \sum_i p(i) \log_2 p(i)/q(i)$.

In order to test whether this distance is significance, they use a bootstrapping idea: compute distances based on randomly assigning points from the two windows to two sets, and computing the distance. A high quantile (e.g. the 99th percentile) of the distances is used as a boundary: if the measured KL distance exceeds this for several steps, we declare that a change has occurred. The whole procedure can be implemented efficiently in a streaming fashion by keeping appropriate data structures, and observing that as the sliding window advances, we do not have to recompute the KL distance from scratch, but rather can compute it incrementally from the previous value with only a few operations. This technique turns out to be quite efficient in practice, requiring only tens of microseconds per update. Many extensions and variations are possible, based on variant formulations, and the use of other change tests, kernel based methods etc.

3.3 Clustering

The notion of a cluster is a familiar one: we often talk of "cancer clusters", or "crime clusters", indicating a high local density of events. Formally, given a set of items, a good clustering places those items that are similar together in clusters, and ensures that the items in different clusters are different. It is natural to try to extend clustering to a stream, but what does it mean when the stream is so large we cannot store for each point which cluster it is allocated to? Typically, we seek a number of clusters, k, which is much smaller than the number of points, n to be clustered. After seeing the stream, the output is just the k clusters, from which the mapping of points to clusters is implicit (e.g. each point is mapped to its closest cluster).

We give a simple example of clustering the stream based on optimizing the k-center objective: attempting to minimize the diameter (the maximum distance between any two points in the same cluster). The algorithm arises by guessing the diameter of the clustering is some value d. The first point is allocated a cluster of its own. For each subsequent point in the stream, if it is far from any existing cluster, a new cluster containing the new point is created, else it is allocated to an existing cluster. If the guess of d was good, then no more than k clusters will be created. Moreover, if d was reasonably close to the true diameter, then the diameter of the stream clustering will be within a factor of 2 of the best possible cluster radius. By trying different guesses of d in parallel, and discarding any that generate more than k clusters, we can build a $(1 + \epsilon, 0)$ (i.e. deterministic) clustering algorithm [5, 7].

Other stream clustering algorithms get more complex. Some are based on the notion of "core-sets": a small subset of the input such that solving the problem on the subset gives a good approximation to the solution on the full input. Yet more use a hierarchical approach: solving the problem exactly on a small subset of data that fits in memory, then merging such solutions to get an approximate solution to the full problem. Different techniques are needed to guarantee good results for other clustering objective functions, such as k-median, k-means and so on.

References

1. R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*,1993.
2. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, pages 20–29, 1996.
3. A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *STOC*, 1998.
4. A. Chakrabarti, G. Cormode, and A. McGregor. A near-optimal algorithm for computing the entropy of a stream. In *SODA*, 2007.
5. M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. In *STOC*, 1997.
6. G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
7. G. Cormode, S. Muthukrishnan, and W. Zhuang. Conquering the divide: Continuous clustering of distributed data streams. In *ICDE*, 2007.
8. C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD* , 2003.
9. T. Dasu, S. Krishnan, S. Venkatasubramanian, and K. Yi. An information theoretic approach to detecting changes in multi-dimensional data streams. In *Interface*, 2006.
10. P. Flajolet and G. N. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.
11. G.S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.
12. A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.
13. S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2005.
14. K. To, T. Ye, and S. Bhattacharyya. CMON: A general purpose continuous IP backbone traffic analysis platform. Research Report RR04-ATL-110309, Sprint ATL, 2004.
15. J. S. Vitter. Random sampling with a reservoir. *ACM Trans. on Mathematical Software*, 11(1):37–57, March 1985.