Temporally-Biased Sampling for Online Model Management

Brian Hentschel*
Harvard University
bhentschel@g.harvard.edu

Peter J. Haas*
University of Massachusetts
phaas@cs.umass.edu

Yuanyuan Tian
IBM Research – Almaden
ytian@us.ibm.com

ABSTRACT

To maintain the accuracy of supervised learning models in the presence of evolving data streams, we provide temporally-biased sampling schemes that weight recent data most heavily, with inclusion probabilities for a given data item decaying exponentially over time. We then periodically retrain the models on the current sample. This approach speeds up the training process relative to training on all of the data. Moreover, time-biasing lets the models adapt to recent changes in the data while—unlike in a sliding-window approach—still keeping some old data to ensure robustness in the face of temporary fluctuations and periodicities in the data values. In addition, the sampling-based approach allows existing analytic algorithms for static data to be applied to dynamic streaming data essentially without change. We provide and analyze both a simple sampling scheme (T-TBS) that probabilistically maintains a target sample size and a novel reservoir-based scheme (R-TBS) that is the first to provide both complete control over the decay rate and a guaranteed upper bound on the sample size, while maximizing both expected sample size and sample-size stability. The latter scheme rests on the notion of a "fractional sample" and, unlike T-TBS, allows for data arrival rates that are unknown and time varying. R-TBS and T-TBS are of independent interest, extending the known set of unequal-probability sampling schemes. We discuss distributed implementation strategies; experiments in Spark illuminate the performance and scalability of the algorithms, and show that our approach can increase machine learning robustness in the face of evolving data.

1 INTRODUCTION

A key challenge for machine learning (ML) is to keep ML models from becoming stale in the presence of evolving data. In the context of the emerging Internet of Things (IoT), for example, the data comprises dynamically changing sensor streams [26], and a failure to adapt to changing data can lead to a loss of predictive power.

One way to deal with this problem is to re-engineer existing static supervised learning algorithms to become adaptive. Some parametric algorithms such as SVM can indeed be re-engineered so that the parameters are time-varying, but for non-parametric algorithms such as kNN-based classification, it is not at all clear how re-engineering can be accomplished. We therefore consider alternative approaches in which we periodically retrain ML models, allowing static ML algorithms to be used in dynamic settings essentially as-is. There are several possible retraining approaches.

Retraining on cumulative data: Periodically retraining a model on all of the data that has arrived so far is clearly infeasible because of the huge volume of data involved. Moreover, recent data is swamped by the massive amount of past data, so the retrained model is not sufficiently adaptive.

Sliding windows: A simple sliding-window approach would be to, e.g., periodically retrain on the data from the last two hours. If the data arrival rate is high and there is no bound on memory, then one must deal with long retraining times caused by large amounts of data in the window. The simplest way to bound the window size is to retain the last n items. Alternatively, one could try to subsample within the time-based window [14]. The fundamental problem with all of these bounding approaches is that old data is completely forgotten; the problem is especially severe when the data arrival rate is high. This can undermine the robustness of an ML model in situations where old patterns can reassert themselves. For example, a singular event such as a holiday, stock market drop, or terrorist attack can temporarily disrupt normal data patterns, which will reestablish themselves once the effect of the event dies down. Periodic data patterns can lead to the same phenomenon. Another example, from [27], concerns influencers on Twitter: a prolific tweeter might temporarily stop tweeting due to travel, illness, or some other reason, and hence be completely forgotten in a sliding-window approach. Indeed, in real-world Twitter data, almost a quarter of top influencers were of this type, and were missed by a sliding window approach.

Temporally biased sampling: An appealing alternative is a temporally biased sampling-based approach, i.e., maintaining a sample that heavily emphasizes recent data but also contains a small amount of older data, and periodically retraining a model on the sample. By using a time-biased sample, the retraining costs can be held to an acceptable level while not sacrificing robustness in the presence of recurrent patterns. This approach was proposed in [27] in the setting of graph analysis algorithms, and has recently been adopted in the MacroBase system [3]. The orthogonal problem of choosing when to retrain a model is also an important question, and is related to, e.g., the literature on "concept drift" [13]; in this paper we focus on the problem of how to efficiently maintain a time-biased sample.

In more detail, our time-biased sampling algorithms ensure that the "appearance probability" for a given data item—i.e., the probability that the item appears in the current sample—decays over time at a controlled exponential rate. Specifically, we assume that items arrive in batches (see the next section for more details), and our goal is to ensure that (i) our sample is representative in that all items in a given batch are equally likely to be in the sample, and (ii) if items i and j belong to batches that have arrived at (wall clock) times t' and t'' with $t' \leq t''$, then for any time $t \geq t''$ our sample S_t is such that

$$\Pr[i \in S_t]/\Pr[j \in S_t] = e^{-\lambda(t''-t')}. \qquad (1)$$

Thus items with a given timestamp are sampled uniformly, and items with different timestamps are handled in a carefully controlled manner. The criterion in (1) is natural and appealing in applications and, importantly, is interpretable and understandable to users. As discussed in [27], the value of the decay rate λ can be chosen to meet application-specific criteria. For example, by setting $\lambda = 0.058$, around 10% of the data items from 40 batches

---

ago are included in the current analysis. As another example, suppose that, $k = 150$ batches ago, an entity such as a person or city was represented by $n = 1000$ data items and we want to ensure that, with probability $q = 0.01$, at least one of these data items remains in the current sample. Then we would set $\lambda = -k^{-1} \ln(1 - (1 - q)^{1/n}) \approx 0.077$. If training data is available, λ can also be chosen to maximize accuracy via cross validation.

The exponential form of the decay function has been adopted by the majority of time-biased-sampling applications in practice because otherwise one would typically need to track the arrival time of every data item—both in and outside of the sample—and decay each item individually at an update, which would make the sampling operation intolerably slow. (A "forward decay" approach that avoids this difficulty, but with its own costs, has been proposed in [9]; we plan to investigate forward decay in future work.) Exponential decay functions make update operations fast and simple.

For the case in which the item-arrival rate is high, the main issue is to keep the sample size from becoming too large. On the other hand, when the incoming batches become very small or widely spaced, the sample sizes for all of the time-biased algorithms that we discuss (as well as for sliding-window schemes based on wall-clock time) can become small. This is a natural consequence of treating recent items as more important, and is characteristic of any sampling scheme that satisfies (1). We emphasize that—as shown in our experiments—a smaller, but carefully time-biased sample typically yields greater prediction accuracy than a sample that is larger due to overloading with too much recent data or too much old data. I.e., more sample data is not always better. Indeed, with respect to model management, this decay property can be viewed as a feature in that, if the data stream dries up and the sample decays to a very small size, then this is a signal that there is not enough new data to reliably retrain the model, and that the current version should be kept for now.

It is surprisingly hard to both enforce (1) and to bound the sample size. As discussed in detail in Section 7, prior algorithms that bound the sample size either cannot consistently enforce (1) or cannot handle wall-clock time. Examples of the former include algorithms based on the A-Res scheme of Efraimidis and Spirakis [12], and Chao's algorithm [5]. A-Res enforces conditions on the *acceptance* probabilities of items; this leads to appearance probabilities which, unlike (1), are both hard to compute and not intuitive. A similar example is provided by Chao's algorithm [5]. In Appendix D of [16] we demonstrate how the algorithm can be specialized to the case of exponential decay and modified to handle batch arrivals. We then show that the resulting algorithm fails to enforce (1) either when initially filling up an empty sample or in the presence of data that arrives slowly relative to the decay rate, and hence fails if the data rate fluctuates too much. The second type of algorithm, due to Aggarwal [1] can only control appearance probabilities based on the indices of the data items. For example, after n items arrive, one could require that, with 95% probability, the $(n-k)$th item should still be in the sample for some specified $k < n$. If the data arrival rate is constant, then this might correspond to a constraint of the form "with 95% probability a data item that arrived 10 hours ago is still in the sample", which is often more natural in applications. For varying arrival rates, however, it is impossible to enforce the latter type of constraint, and a large batch of arriving data can prematurely flush out older data. Thus our new sampling schemes are interesting in their

own right, significantly expanding the set of unequal-probability sampling techniques.

T-TBS: We first provide and analyze Targeted-Size Time-Biased Sampling (T-TBS), a simple algorithm that generalizes the sampling scheme in [27]. T-TBS allows complete control over the decay rate (expressed in wall-clock time) and probabilistically maintains a target sample size. That is, the expected and average sample sizes converge to the target and the probability of large deviations from the target decreases exponentially or faster in both the target size and the deviation size. T-TBS is simple and highly scalable when applicable, but only works under the strong restriction that the mean data arrival rate is known and constant. There are scenarios where T-TBS might be a good choice (see Section 3), but many applications have non-constant, unknown mean arrival rates or cannot tolerate sample overflows.

R-TBS: We then provide a novel algorithm, Reservoir-Based Time-Biased Sampling (R-TBS), that is the first to simultaneously enforce (1) at all times, provide a guaranteed upper bound on the sample size, and allow unknown, varying data arrival rates. Guaranteed bounds are desirable because they avoid memory management issues associated with sample overflows, especially when large numbers of samples are being maintained—so that the probability of *some* sample overflowing is high—or when sampling is being performed in a limited memory setting such as at the "edge" of the IoT. Also, bounded samples reduce variability in retraining times and do not impose upper limits on the incoming data flow.

The idea behind R-TBS is to adapt the classic reservoir sampling algorithm, which bounds the sample size but does not allow time biasing. Our approach rests on the notion of a "fractional" sample whose nonnegative size is real-valued in an appropriate sense. We show that, over all sampling algorithms having exponential decay, R-TBS maximizes the expected sample size whenever the data arrival rate is low and also minimizes the sample-size variability.

Distributed implementation: Both T-TBS and R-TBS can be parallelized. Whereas T-TBS is relatively straightforward to implement, an efficient distributed implementation of R-TBS is nontrivial. We exploit various implementation strategies to reduce I/O relative to other approaches, avoid unnecessary concurrency control, and make decentralized decisions about which items to insert into, or delete from, the reservoir.

Organization: The rest of the paper is organized as follows. In Section 2 we formally describe our batch-arrival problem setting and discuss two prior simple sampling schemes: a simple Bernoulli scheme as in [27] and the classical reservoir sampling scheme, modified for batch arrivals. These methods either bound the sample size but do not control the decay rate, or control the decay rate but not the sample size. We next present and analyze the T-TBS and R-TBS algorithms in Section 3 and Section 4. We describe the distributed implementation in Section 5, and Section 6 contains experimental results. We review the related literature in Section 7 and conclude in Section 8.

2 SETTING AND PRIOR SCHEMES

After introducing our problem setting, we discuss two prior sampling schemes that provide context for our current work: simple Bernoulli time-biased sampling (B-TBS) with no sample-size control and the classical reservoir sampling algorithm (with no time biasing), modified for batch arrivals (B-RS).

Setting: Items arrive in *batches* $\mathcal{B}_1, \mathcal{B}_2, \ldots$, at time points $t = 1, 2, \ldots$, where each batch contains 0 or more items. This

simple integer batch sequence often arises from the discretization of time [24, 28]. Specifically, the continuous time domain is partitioned into intervals of length Δ, and the items are observed only at times $\{k\Delta : k = 0, 1, 2, \ldots\}$. All items that arrive in an interval $\big[k\Delta, (k + 1)\Delta\big)$ are treated as if they arrived at time $k\Delta$, i.e., at the start of the interval, so that all items in batch $\mathcal{B}_i$ have time stamp $i\Delta$, or simply time stamp i if time is measured in units of length Δ. As discussed below, our results can straightforwardly be extended to arbitrary real-valued batch-arrival times.

Our goal is to generate a sequence $\{S_t\}_{t \geq 0}$, where S_t is a sample of the items that have arrived at or prior to time t, i.e., a sample of the items in $U_t = S_0 \cup \big(\bigcup_{i=1}^{t} \mathcal{B}_i\big)$. Here we allow the initial sample S_0 to start out nonempty. These samples should be biased towards recent items so as to enforce (1) for $i \in \mathcal{B}_{t'}$ and $j \in \mathcal{B}_{t''}$ while keeping the sample size as close as possible to (and preferably never exceeding) a specified target n.

Our assumption that batches arrive at integer time points can easily be dropped. In all of our algorithms, inclusion probabilities—and, as discussed later, closely related item "weights"—are updated at a batch arrival time t' with respect to their values at the previous time $t = t' - 1$ via multiplication by $e^{-\lambda}$. To extend our algorithms to handle arbitrary successive batch arrival times t and t', we simply multiply instead by $e^{-\lambda(t'-t)}$. Thus our results can be applied to arbitrary sequences of real-valued batch arrival times, and hence to an arbitrary sequences of item arrivals (since batches can comprise single items).

Bernoulli Time-Biased Sampling (B-TBS): In the simplest sampling scheme, at each time t, we accept each incoming item $x \in \mathcal{B}_t$ into the sample with probability 1. At each subsequent time $t' > t$, we flip a coin independently for each item currently in the sample: an item is retained in the sample with probability $p = e^{-\lambda}$ and removed with probability $1 - p$. It is straightforward to adapt the algorithm to batch arrivals; see Appendix A of [16], where we show that $\Pr[x \in S_{t'}] = e^{-\lambda(t'-t)}$ for $x \in \mathcal{B}_t$, implying (1). This is essentially the algorithm used, e.g., in [27] to implement time-biased edge sampling in dynamic graphs. The user, however, cannot independently control the expected sample size, which is completely determined by λ and the sizes of the incoming batches. In particular, if the batch sizes systematically grow over time, then sample size will grow without bound. Arguments in [27] show that if $\sup_t |\mathcal{B}_t| < \infty$, then the sample size can be bounded, but only probabilistically. See Remark 1 below for extensions and refinements of these results.

Batched Reservoir Sampling (B-RS): The classic reservoir sampling algorithm can be modified to handle batch arrivals; see Appendix B of [16]. Although B-RS guarantees an upper bound on the sample size, it does not support time biasing. The R-TBS algorithm (Section 4) maintains a bounded reservoir as in B-RS while simultaneously allowing time-biased sampling.

3 TARGETED-SIZE TBS

As a first step towards time-biased sampling with a controlled sample size, we describe the simple T-TBS scheme, which improves upon the simple Bernoulli sampling scheme B-TBS by ensuring the inclusion property in (1) while providing probabilistic guarantees on the sample size. We require that the mean batch size equals a constant b that is both known in advance and "large enough" in that $b \geq n(1 - e^{-\lambda})$, where n is the target sample size and λ is the decay rate as before. The requirement on b ensures that, at the target sample size, items arrive on average at least as fast as they decay.

Algorithm 1: Targeted-size TBS (T-TBS)

1 λ: decay factor (≥ 0)
2 n: target sample size
3 b: assumed mean batch size such that $b \geq n(1 - e^{-\lambda})$
4 Initialize: $S \leftarrow S_0$; $p \leftarrow e^{-\lambda}$; $q \leftarrow n(1 - e^{-\lambda})/b$
5 **for** $t \leftarrow 1, 2, \ldots$ **do**
6     $m \leftarrow \textsc{Binomial}(|S|, p)$ //simulate $|S|$ trials
7     $S \leftarrow \textsc{Sample}(S, m)$ //retain m random elements
8     $k \leftarrow \textsc{Binomial}(|\mathcal{B}_t|, q)$
9     $B'_t \leftarrow \textsc{Sample}(\mathcal{B}_t, k)$ //down-sample new batch
10    $S \leftarrow S \cup B'_t$
11   output S

The pseudocode is given as Algorithm 1. T-TBS is similar to B-TBS in that we downsample by performing a coin flip for each item with retention probability p. Unlike B-TBS, we downsample the incoming batches at rate $q = n(1 - e^{-\lambda})/b$, which ensures that n becomes the "equilibrium" sample size. Specifically, when the sample size equals n, the expected number $n(1 - e^{-\lambda})$ of current items deleted at an update equals the expected number qb of inserted new items, which causes the sample size to drift towards n. Arguing similarly to Appendix A of [16], we have for $t' \geq t \geq 1$ and $x \in \mathcal{B}_t$ that $\Pr[x \in S_{t'}] = qe^{-\lambda(t'-t)}$, so that the key relative appearance property in (1) holds.

For efficiency, the algorithm exploits the fact that for k independent trials, each having success probability r, the total number of successes has a binomial distribution with parameters k and r. Thus, in lines 6 and 8, the algorithm simulates the coin tosses by directly generating the number of successes m or k—which can be done using standard algorithms [17]—and then retaining m or k randomly chosen items. So the function $\textsc{Binomial}(j, r)$ returns a random sample from the binomial distribution with j independent trials and success probability r per trial, and the function $\textsc{Sample}(A, m)$ returns a uniform random sample, without replacement, containing $\min(m, |A|)$ elements of the set A; note that the function call $\textsc{Sample}(A, 0)$ returns an empty sample for any empty or nonempty A.

Theorem 3.1 below precisely describes the behavior of the sample size; the proof—along with the proofs of most other results in the paper—is given in Appendix C of [16]. Denote by $B_t = |\mathcal{B}_t|$ the (possibly random) size of $\mathcal{B}_t$ for $t \geq 1$ and by $C_t = |S_t|$ the sample size at time t for $t \geq 0$; assume that C_0 is a finite deterministic constant. Define the *upper-support ratio* for a random batch size B as $r = b^*/b \geq 1$, where $b = \mathrm{E}[B]$ and b^* is the smallest positive number such that $P[B \leq b^*] = 1$; set $r = \infty$ if B can be arbitrarily large. For $r \in [1, \infty)$, set

$$v^+_{\epsilon, r} = (1 + \epsilon)\ln\big((1 + \epsilon)/r\big) - (1 + \epsilon - r).$$

for $\epsilon > 0$ and

$$v^-_{\epsilon, r} = (1 - \epsilon)\ln\big((1 - \epsilon)/r\big) - (1 - \epsilon - r)$$

for $\epsilon \in (0, 1)$. Note that $v^+_{\epsilon, r} > 0$ and is strictly increasing in ϵ for $\epsilon > r - 1$, and that $v^-_{\epsilon, r}$ increases from $r - 1 - \ln r$ to r as ϵ increases from 0 to 1. Write "i.o." to denote that an event occurs "infinitely often", i.e., for infinitely many values of t, and write "w.p.1" for "with probability 1".

THEOREM 3.1. *Suppose that the batch sizes $\{B_t\}_{t \geq 1}$ are i.i.d with common mean $b \geq n(1 - e^{-\lambda})$, finite variance, and upper support ratio r. Then, for any $p = e^{-\lambda} < 1$,*

(i) *for all $m \geq 0$, we have $\Pr[C_t = m \text{ i.o.}] = 1$;*
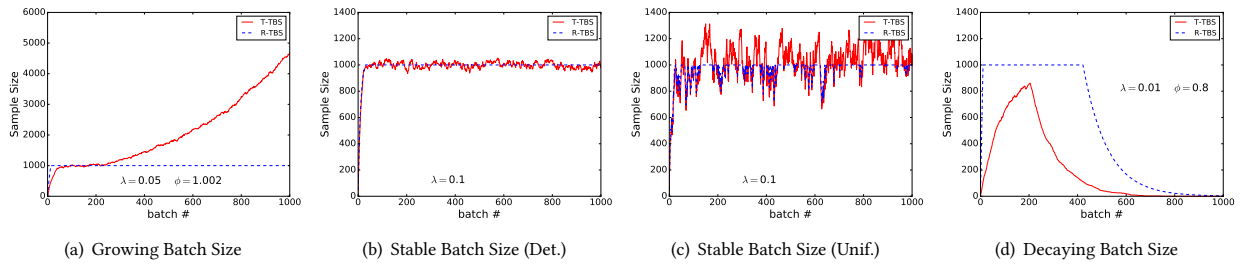(ii) *$\mathrm{E}[C_t] = n + p^t(C_0 - n)$ for $t > 0$;*

Figure 1: Targeted TBS: Sample Size Behavior, λ **= decay rate and** ϕ **= batch size multiplier.**

(iii) $\lim_{t \to \infty} (1/t) \sum_{i=0}^{t} C_i = n$ w.p.1;

(iv) if $C_0 = n$ and $r < \infty$, then

    (a) $\Pr[C_t \ge (1 + \epsilon)n] \le e^{-nv_{\epsilon,r}^+} \left(1 + O(n\epsilon p^t)\right)$ and

    (b) $\Pr[C_t \le (1 - \epsilon)n] \le e^{-nv_{\epsilon,r}^-} \left(1 + O(n(1-\epsilon)p^t)\right)$

    for (a) $\epsilon, t > 0$ and (b) $\epsilon \in (0, 1)$ and $t \ge \ln \epsilon / \ln p$.

In Appendix C of [16], we actually prove a stronger version of the theorem in which the assumption in (iv) that $r < \infty$ is dropped.

Thus, from (ii), $\lim_{t \to \infty} \mathrm{E}[C_t] = n$ so that the expected sample size converges to the target size n as t becomes large; indeed, if $C_0 = n$ then the expected sample size equals n for all $t > 0$. By (iii), an even stronger property holds in that, w.p.1, the average sample size—averaged over the first t batch-arrival times—converges to n as t becomes large. For typical batch-size distributions, the assertions in (iv) imply that, at any given time t, the probability that the sample size deviates from n by more than $100\epsilon\%$ decreases exponentially with n and—in the case of a positive deviation as in (iv)(a)—super-exponentially in ϵ. However, the assertion in (i) implies that any sample size m, no matter how large, will be exceeded infinitely often w.p.1; indeed, it follows from the proof that the mean times between successive exceedances are not only finite, but are uniformly bounded over time. In summary, the sample size is generally stable and close to n on average, but is subject to infrequent, but unboundedly large spikes in the sample size, so that sample-size control is incomplete.

Indeed, when batch sizes fluctuate in a non-predicable way, as often happens in practice, T-TBS can break down; see Figure 1, in which we plot sample sizes for T-TBS and, for comparison, R-TBS. The problem is that the value of the mean batch size b must be specified in advance, so that the algorithm cannot handle dynamic changes in b without losing control of either the decay rate or the sample size.

In Figure 1(a), for example, the (deterministic) batch size is initially fixed and the algorithm is tuned to a target sample size of 1000, with a decay rate of $\lambda = 0.05$. At $t = 200$, the batch size starts to increase (with $B_{t+1} = \phi B_t$ where $\phi = 1.002$), leading to an overflowing sample, whereas R-TBS maintains a constant sample size.

Even in a stable batch-size regime with constant batch sizes (or, more generally, small variations in batch size), R-TBS can maintain a constant sample size whereas the sample size under T-TBS fluctuates in accordance with Theorem 3.1; see Figure 1(b) for the case of a constant batch size $B_t \equiv 100$ with $\lambda = 0.1$.

Large variations in the batch size lead to large fluctuations in the sample size for T-TBS; in this case the sample size for R-TBS is bounded above by design, but large drops in the batch size can cause drops in the sample size for both algorithms; see Figure 1(c) for the case of $\lambda = 0.1$ and i.i.d. uniformly distributed batch sizes on [0, 200] so that $\mathrm{E}[B_t] \equiv 100$. Similarly, as shown in Figure 1(d), systematically decreasing batch sizes will cause the

sample size to shrink for both T-TBS and R-TBS. Here, $\lambda = 0.01$ and, as with Figure 1(a), the batch size is initially fixed and then starts to change at time $t = 200$, with $\phi = 0.8$ in this case. This experiment—and others, not reported here, with varying values of λ and ϕ—indicate that R-TBS is more robust to sample underflows than T-TBS.

Overall, however, T-TBS is of interest because, when the mean batch size is known and constant over time, and when some sample overflows are tolerable, T-TBS is simple to implement and parallelize, and is very fast (see Section 6). For example, if the data comes from periodic polling of a set of robust sensors, the data arrival rate will be known a priori and will be relatively constant, except for the occasional sensor failure, and hence T-TBS might be appropriate. On the other hand, if data is coming from, e.g., a social network, then batch sizes may be hard to predict.

REMARK 1. When $q = 1$, Theorem 3.1 provides a description of sample-size behavior for B-TBS. Under the conditions of the theorem, the expected sample size converges to $n = b/(1 - e^{-\lambda})$, which illustrates that the sample size and decay rate cannot be controlled independently. The actual sample size fluctuates around this value, with large deviations above or below being exponentially or super-exponentially rare. Thus Theorem 3.1 both complements and refines the analysis in [27].

4 RESERVOIR-BASED TBS

Targeted time-biased sampling (T-TBS) controls the decay rate but only partially controls the sample size, whereas batched reservoir sampling (B-RS) bounds the sample size but does not allow time biasing. Our new reservoir-based time-biased sampling algorithm (R-TBS) combines the best features of both, controlling the decay rate while ensuring that the sample never overflows and has optimal sample size and stability properties. Importantly, unlike T-TBS, the R-TBS algorithm can handle any sequence of batch sizes.

4.1 The R-TBS Algorithm

To maintain a bounded sample, R-TBS combines the use of a reservoir with the notion of item *weights*. In R-TBS, the weight of an item initially equals 1 but then decays at rate λ, i.e., the weight of an item $i \in \mathcal{B}_t$ at time $t' \ge t$ is $w_{t'}(i) = e^{-\lambda(t'-t)}$. All items arriving at the same time have the same weight, so that the *total weight* of all items seen up through time t is $W_t = \sum_{j=1}^{t} B_j e^{-\lambda(t-j)}$, where, as before, $B_j = |\mathcal{B}_j|$ is the size of the jth batch.

R-TBS generates a sequence of latent "fractional samples" $\{L_t\}_{t \ge 0}$ such that (i) the "size" of each L_t equals the *sample weight* C_t, defined as $C_t = \min(n, W_t)$, and (ii) L_t contains $\lfloor C_t \rfloor$ "full" items and at most one "partial" item. For example, a latent sample of size $C_t = 3.6$ contains three "full" items that belong to the actual sample S_t with probability 1 and one partial item that

Algorithm 2: Reservoir-based TBS (R-TBS)

```
1  λ: decay factor (≥ 0)
2  n: maximum sample size
3  Initialize: A ← A₀; W ← C ← |A₀|; π ← ∅          //|A₀| ≤ n
4  for t ← 1, 2, . . . do
5  │  if W < n then                        //has been unsaturated
6  │  │  W ← e⁻λW                           //decay current items
7  │  │  if W > 0 then
8  │  │  └  (A, π, C) ← DSAMPLE((A, π, C), W)
9  │  │  A ← A ∪ Bₜ                        //accept all items in Bₜ
10 │  │  W ← W + |Bₜ|                        //update total weight
11 │  │  if W > n then                     //sample is now saturated
12 │  │  │  //adjust for overshoot
   │  │  └  (A, π, C) ← DSAMPLE((A, π, W), n)
13 │  else                                    //has been saturated
14 │  │  W ← e⁻λW + |Bₜ|                     //new total weight
15 │  │  if W ≥ n then                      //still saturated
16 │  │  │  m ← STOCHROUND(|Bₜ|n/W)
   │  │  │  //replace m A-items with m Bₜ-items
17 │  │  └  A ← A \ SAMPLE(A, m) ∪ SAMPLE(Bₜ, m)
18 │  │  else                                 //now unsaturated
   │  │  │  //adjust for undershoot
19 │  │  │  (A, π, C) ← DSAMPLE((A, π, n), W − |Bₜ|)
20 │  │  └  A ← A ∪ Bₜ                 //all batch items are full
21 │  S ← GETSAMPLE(A, π, C)
22 └  output S
```
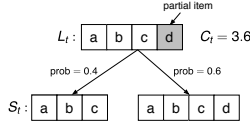

Figure 2: Latent sample L_t (sample weight $C_t = 3.6$) and possible realized samples.

belongs to S_t with probability 0.6. Thus S_t is obtained by including each full item and then including the partial item according to its associated probability, so that C_t represents the expected size of S_t. E.g., in our example, the sample S_t will contain either three or four items with respective probabilities 0.4 and 0.6, so that the expected sample size is 3.6; see Figure 2. Note that if $C_t = k$ for some $k \in \{0, 1, \ldots, n\}$, then with probability 1 the sample contains precisely k items, and C_t is the actual size of S_t, rather than just the expected size. Since each C_t by definition never exceeds n, no sample S_t ever contains more than n items.

More precisely, given a set U of items, a *latent sample* of U with sample weight C is a triple $L = (A, \pi, C)$, where $A \subseteq U$ is a set of $\lfloor C \rfloor$ *full* items and $\pi \subseteq U$ is a (possibly empty) set containing at most one *partial* item. At each time t, we randomly generate S_t from $L_t = (A_t, \pi_t, C_t)$ by sampling such that

$$S_t = \begin{cases} A_t \cup \pi & \text{with probability } \mathrm{frac}(C_t); \\ A_t & \text{with probability } 1 - \mathrm{frac}(C_t), \end{cases} \quad (2)$$

where $\mathrm{frac}(x) = x - \lfloor x \rfloor$. That is, each full item is included with probability 1 and the partial item is included with probability $\mathrm{frac}(C_t)$. Thus

$$\begin{aligned} \mathrm{E}[|S_t|] &= \lceil C_t \rceil \, \mathrm{frac}(C_t) + \lfloor C_t \rfloor \big(1 - \mathrm{frac}(C_t)\big) \\ &= (\lceil C_t \rceil - \lfloor C_t \rfloor) \, \mathrm{frac}(C_t) + \lfloor C_t \rfloor \quad (3) \\ &= \mathrm{frac}(C_t) + \lfloor C_t \rfloor = C_t \end{aligned}$$

as previously asserted. By allowing at most one partial item, we minimize the latent sample's footprint: $|A_t \cup \pi_t| \leq \lfloor C_t \rfloor + 1$.

The key goal of R-TBS is to maintain the invariant

$$\Pr[i \in S_t] = (C_t / W_t) w_t(i) \quad (4)$$

for each $t \geq 0$ and each item $i \in U_t$, where, as before, U_t denotes the set of all items that arrive up through time t, so that the appearance probability for an item i at time t is proportional to its weight $w_t(i)$. This immediately implies the desired relative-inclusion property (1). Since $w_t(i) = 1$ for an arriving item $i \in B_t$, the equality in (4) implies that the initial acceptance probability for this item is

$$\Pr[i \in S_t] = C_t / W_t. \quad (5)$$

The pseudocode for R-TBS is given as Algorithm 2. Suppose the sample is *unsaturated* at time $t - 1$ in that $W_{t-1} < n$ and hence $C_{t-1} = W_{t-1}$ (line 5). The decay process first reduces the total weight (and hence the sample weight) to $W'_{t-1} = C'_{t-1} = e^{-\lambda} W_{t-1}$ (line 6). R-TBS then *downsamples* L_{t-1} (line 8) to reflect this decay and maintain a minimal sample footprint; the downsampling method, described in Section 4.2, is designed to maintain the invariant in (4). If the weight of the arriving batch does not cause the sample to overflow, i.e., $C'_{t-1} + |B_t| < n$, then $C_t = C'_{t-1} + |B_t| = W'_{t-1} + |B_t| = W_t$. The relation in (5) then implies that all newly arrived items are accepted into the sample with probability 1 (line 9); see Figure 3(a) for an example of this scenario. The situation is more complicated if the weight of the arriving batch would cause the sample to overflow. It turns out that the simplest way to deal with this scenario is to initially accept all incoming items as in line 9, and then run an additional round of downsampling to reduce the sample weight to n (line 12), so that the sample is now saturated; see Figure 3(b). Note that these two steps can be executed without ever causing the sample footprint to exceed n.

Now suppose that the sample is *saturated* at time $t - 1$, so that $W_{t-1} \geq n$ and hence $C_{t-1} = |S_{t-1}| = n$. The new total weight is $W_t = W'_{t-1} + |B_t|$ as before (line 14). If $W_t \geq n$, then the weight of the arriving batch exceeds the weight loss due to decay, and the sample remains saturated. Then (5) implies that each item in B_t is accepted into the sample with probability $p = n/W_t$. Letting $I_j = 1$ if item $j \in B$ is accepted and $I_j = 0$ otherwise, we see that the expected number of accepted items is

$$m = \mathrm{E}\Big[\sum_{j \in B_t} I_j\Big] = \sum_{j \in B_t} \mathrm{E}[I_j] = \sum_{j \in B_t} \Pr[I_j = 1] = B_t n / W_t.$$

There are a number of possible ways to carry out this acceptance operation, e.g., via independent coin flips. To minimize the variability of the sample size (and hence the likelihood of severely small samples), R-TBS uses *stochastic rounding* in line 16 and accepts a random number of items M such that $M = \lfloor m \rfloor$ with probability $\lceil m \rceil - m$ and $M = \lceil m \rceil$ with probability $m - \lfloor m \rfloor$, so that $\mathrm{E}[M] = m$ by an argument essentially the same as in (3). To maintain the bound on the sample size, the M accepted items replace M randomly selected "victims" in the current sample (line 17). If $W_t < n$, then the sample weight decays to W'_{t-1} and the weight of the arriving batch is not enough to fill the sample back up. Moreover, (5) implies that all arriving items are accepted with probability 1. Thus we downsample to the decayed weight of $W'_{t-1} = W_t - |B_t|$ in line 19 and then insert the arriving items in line 20.

4.2 Downsampling

Before describing Algorithm 3, the downsampling algorithm, we intuitively motivate a key property that any such procedure must have. For any item $i \in L$, the relation in (4) implies that we must have $\Pr[i \in S] = (C/W) w_i$ and $\Pr[i \in S'] = (C'/W') w'_i$, where W and w_i represent the total and item weight before decay and
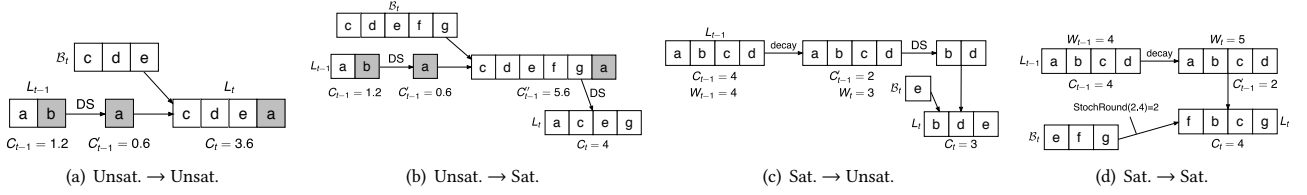
Figure 3: R-TBS scenarios for $n = 4$ and $e^{-\lambda} = 0.5$. For simplicity, we take $W_{t-1} = C_{t-1}$. "DS" denotes downsampling.

Algorithm 3: Downsampling

1   $L = (A, \pi, C)$: input latent sample
2   C': input target weight with $0 < C' < C$
3   $L' = (A', \pi', C')$: output latent sample

4   $U \leftarrow \textsc{Uniform}()$
5   **if** $\lfloor C' \rfloor = 0$ **then**   //no full items retained
6   **if** $U > \text{frac}(C)/C$ **then**
7       $(A', \pi') \leftarrow \textsc{Swap1}(A, \pi)$
8     $A' \leftarrow \emptyset$
9   **else if** $0 < \lfloor C' \rfloor = \lfloor C \rfloor$ **then**   //no items deleted
10   **if** $U > \left(1 - (C'/C)\,\text{frac}(C)\right)/\left(1 - \text{frac}(C')\right)$ **then**
11       $(A', \pi') \leftarrow \textsc{Swap1}(A, \pi)$
12   **else**   //items deleted: $0 < \lfloor C' \rfloor < \lfloor C \rfloor$
13   **if** $U \le (C'/C)\,\text{frac}(C)$ **then**
14       $A' \leftarrow \textsc{Sample}(A, \lfloor C' \rfloor)$
15       $(A', \pi') \leftarrow \textsc{Swap1}(A', \pi)$
16   **else**
17       $A' \leftarrow \textsc{Sample}(A, \lfloor C' \rfloor + 1)$
18       $(A', \pi') \leftarrow \textsc{Move1}(A', \pi)$
19   **if** $C' = \lfloor C' \rfloor$ **then**   //no fractional item
20     $\pi' \leftarrow \emptyset$

downsampling, and W' and w'_i represent the weights afterwards. Since decay affects all items equally, we have $w/W = w'/W'$, and it follows that

$$\Pr[i \in S'] = (C'/C)\Pr[i \in S]. \qquad (6)$$

That is, the inclusion probabilities for all items must be scaled down by the same fraction, namely C'/C. Theorem 4.1 (later in this section) asserts that Algorithm 3 satisfies this property.

In the pseudocode for Algorithm 3, the function $\textsc{Uniform}()$ generates a random number uniformly distributed on $[0, 1]$. The subroutine $\textsc{Swap1}(A, \pi)$ moves a randomly selected item from A to π and moves the current item in π (if any) to A. Similarly, $\textsc{Move1}(A, \pi)$ moves a randomly selected item from A to π, replacing the current item in π (if any). More precisely, $\textsc{Swap1}(A, \pi)$ executes the operations $I \leftarrow \textsc{Sample}(A, 1)$, $A \leftarrow (A \backslash I) \cup \pi$, and $\pi \leftarrow I$, and $\textsc{Move1}(A, \pi)$ executes the operations $I \leftarrow \textsc{Sample}(A, 1)$, $A \leftarrow A \setminus I$, and $\pi \leftarrow I$.

To gain some intuition for why the algorithm works, consider a simple special case, where the goal is to form a fractional sample $L' = (A', \pi', C')$ from a fractional sample $L = (A, \pi, C)$ of integral size $C > C'$; that is, L comprises exactly C full items. Assume that C' is non-integral, so that L' contains a partial item. In this case, we simply select an item at random (from A) to be the partial item in L' and then select $\lfloor C' \rfloor$ of the remaining $C - 1$ items at random to be the full items in L'; see Figure 4(a). By symmetry, each item $i \in L$ is equally likely to be included in S', so that the inclusion probabilities for the items in L are all scaled down by the same fraction, as required for (6). For example, taking $t = 0$ in Figure 4(a), item a appears in S_t with probability 1 since it is a full item. In S'_t, where the weights have been reduced by 50%, item a (either as a full or partial item, depending on the random outcome) appears with probability $2 \cdot (1/6) + 2 \cdot (1/6) \cdot 0.5 = 0.5$, as expected. This scenario corresponds to lines 17 and 18 in the

algorithm, where we carry out the above selections by randomly sampling $\lfloor C' \rfloor + 1$ items from A to form A' and then choosing a random item in A' as the partial item by moving it to π.

In the case where L contains a partial item i^* that appears in S with probability $\text{frac}(C)$, it follows from (6) that i^* should appear in S' with probability $p = (C'/C)P[i^* \in S] = (C'/C)\,\text{frac}(C)$. Thus, with probability p, lines 13–15 retain i^* and convert it to a full item so that it appears in S'. Otherwise, in lines 17 and 18, i^* is removed from the sample when it is overwritten by a random item from A'; see Figure 4(b). Again, a new partial item is chosen from A in a random manner to uniformly scale down the inclusion probabilities. For instance, in Figure 4(b), item d appears in S_t with probability 0.2 (because it is a partial item) and in S'_t, appears with probability $3 \cdot (0.1/3) = 0.1$. Similarly, item a appears in S_t with probability 1 and in S'_t with probability $(1.8)/6 + 0.6 \cdot (1.8/6) + 0.6 \cdot (0.1/3) = 0.5$.

The if-statement in line 5 corresponds to the corner case in which L' does not contain a full item. The partial item $i^* \in L$ either becomes full or is swapped into A' and then immediately ejected; see Figure 4(c).

The if-statement in line 9 corresponds to the case in which no items are deleted from the latent sample, e.g., when $C = 4.7$ and $C' = 4.2$. In this case, i^* either becomes full by being swapped into A' or remains as the partial item for L'. Denoting by ρ the probability of *not* swapping, we have $P[i^* \in S'] = \rho \cdot \text{frac}(C') + (1 - \rho) \cdot 1$. On the other hand, (6) implies that $P[i^* \in S'] = (C'/C)\,\text{frac}(C)$. Equating these expression shows that ρ must equal the expression on the right side of the inequality on line 10; see Figure 4(d).

Formally, we have the following result.

Theorem 4.1. *For $0 < C' < C$, let $L' = (A', \pi', C')$ be the latent sample produced from a latent sample $L = (A, \pi, C)$ via Algorithm 3, and let S' and S be samples produced from L' and L via* (2). *Then $\Pr[i \in S'] = (C'/C)\Pr[i \in S]$ for all $i \in L$.*

4.3 Properties of R-TBS

Theorem 4.2 below asserts that R-TBS satisfies (4) and hence (1), thereby maintaining the correct inclusion probabilities; see Appendix C of [16] for the proof. Theorems 4.3 and 4.4 assert that, among all sampling algorithms with exponential time biasing, R-TBS both maximizes the expected sample size in unsaturated scenarios and minimizes sample-size variability. Thus R-TBS tends to yield more accurate results (from more training data) and greater stability in both result quality and retraining costs.

Theorem 4.2. *The relation $\Pr[i \in S_t] = (C_t/W_t)w_t(i)$ holds for all $t \ge 1$ and $i \in U_t$.*

Theorem 4.3. *Let H be any sampling algorithm that satisfies* (1) *and denote by S_t and S_t^H the samples produced at time t by R-TBS and H. If the total weight at some time $t \ge 1$ satisfies $W_t < n$, then $\text{E}[|S_t^H|] \le \text{E}[|S_t|]$.*

Proof. Since H satisfies (1), it follows that, for each time $j \le t$ and $i \in \mathcal{B}_j$, the inclusion probability $\Pr[i \in S_t^H]$ must be of the
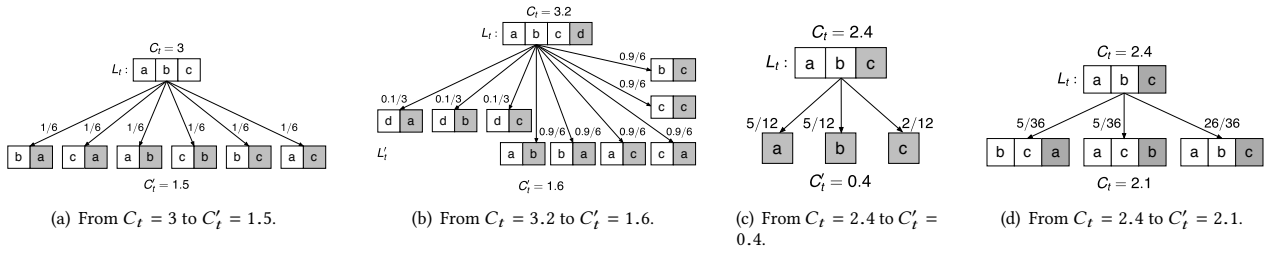
(a) From $C_t = 3$ to $C'_t = 1.5$. (b) From $C_t = 3.2$ to $C'_t = 1.6$. (c) From $C_t = 2.4$ to $C'_t = 0.4$. (d) From $C_t = 2.4$ to $C'_t = 2.1$.

Figure 4: Downsampling examples ($t = 0$).

form $r_t e^{-\lambda(t-j)}$ for some function r_t independent of j. Taking $j = t$, we see that $r_t \leq 1$. For R-TBS in an unsaturated state, (4) implies that $r_t = C_t/W_t = 1$, so that $\Pr[i \in S_t^H] \leq \Pr[i \in S_t]$, and the desired result follows directly. □

THEOREM 4.4. *Let H be any sampling algorithm that satisfies* (1) *and has maximal expected sample size C_t and denote by S_t and S_t^H the samples produced at time t by R-TBS and H. Then* $\mathrm{Var}[|S_t^H|] \geq \mathrm{Var}[|S_t|]$ *for any time $t \geq 1$.*

PROOF. Considering all possible distributions over the sample size having a mean value equal to C_t, it is straightforward to show that variance is minimized by concentrating all of the probability mass onto $\lfloor C_t \rfloor$ and $\lceil C_t \rceil$. There is precisely one such distribution, namely the stochastic-rounding distribution, and this is precisely the sample-size distribution attained by R-TBS. □

5 DISTRIBUTED TBS ALGORITHMS

In this section, we describe how to implement distributed versions of T-TBS and R-TBS to handle large volumes of data.

5.1 Overview of Distributed Algorithms

The distributed T-TBS and R-TBS algorithms, denoted as D-T-TBS and D-R-TBS respectively, need to distribute large data sets across the cluster and parallelize the computation on them.

Overview of D-T-TBS: The implementation of the D-T-TBS algorithm is very similar to the simple distributed Bernoulli time-biased sampling algorithm in [27]. It is embarrassingly parallel, requiring no coordination. At each time point t, each worker in the cluster subsamples its partition of the sample with probability p, subsamples its partition of $\mathcal{B}_t$ with probability q, and then takes a union of the resulting data sets.

Overview of D-R-TBS: This algorithm, unlike D-T-TBS, maintains a bounded sample, and hence cannot be embarrassingly parallel. D-R-TBS first needs to aggregate local batch sizes to compute the incoming batch size $|\mathcal{B}_t|$ to maintain the total weight W. Then, based on $|\mathcal{B}_t|$ and the previous total weight W, D-R-TBS determines whether the reservoir was previously saturated and whether it will be saturated after processing $\mathcal{B}_t$. For each possible situation, D-R-TBS chooses the items in the reservoir to delete through downsampling and the items in $\mathcal{B}_t$ to insert into the reservoir. This process requires the master to coordinate among the workers. In Section 5.3, we introduce two alternative approaches to determine the deleted and inserted items. Finally, the algorithm applies the deletes and inserts to form the new reservoir, and computes the new total weight W.

Both D-T-TBS and D-R-TBS periodically checkpoint the sample as well as other system state variables to ensure fault tolerance. The implementation details for D-T-TBS are mostly subsumed by those for D-R-TBS, so we focus on the latter.
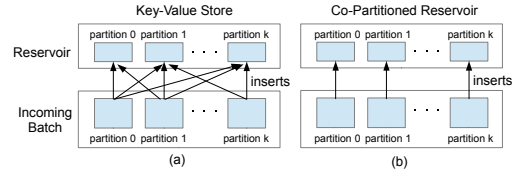

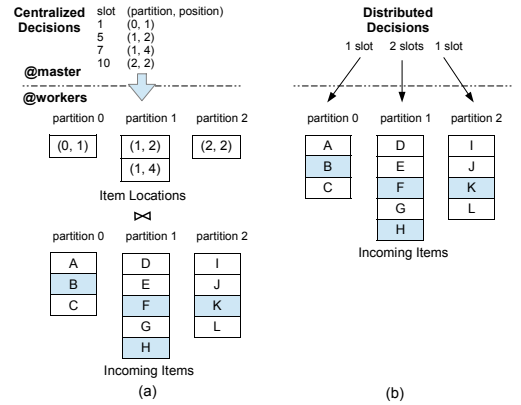**Figure 5: Design choices for implementing the reservoir**

Figure 6: Retrieving insert items

5.2 Distributed Data Structures

There are two important data structures in the D-R-TBS algorithm: the incoming batch and the reservoir. Conceptually, we view an incoming batch $\mathcal{B}_t$ as an array of slots numbered from 1 through $|\mathcal{B}_t|$, and the reservoir as an array of slots numbered from 1 through $\lfloor C \rfloor$ containing full items plus a special slot for the partial item. For both data structures, data items need to be distributed into partitions due to the large data volumes. Therefore, the slot number of an item maps to a specific partition ID and a position inside the partition.

The incoming batch usually comes from a distributed streaming system, such as Spark Streaming; the actual data structure is specific to the streaming system (e.g. an incoming batch is stored as an RDD in Spark Streaming). As a result, the partitioning strategy of the incoming batch is opaque to the D-R-TBS algorithm. Unlike the incoming batch, which is read-only and discarded at the end of each time period, the reservoir data structure must be continually updated. An effective strategy for storing and operating on the reservoir is thus crucial for good performance. We now explore alternative approaches to implementing the reservoir.

Distributed in-memory key-value store: One quite natural approach implements the reservoir using an off-the-shelf distributed in-memory key-value store, such as Redis [25] or Memcached [23]. In this scheme, each item in the reservoir is stored as a key-value pair, with the slot number as the key and the item as the value. Inserts and deletes to the reservoir naturally translate into put and delete operations to the key-value store.

There are two major limitations to this approach. Firstly, the hash-based or range-based data-partitioning scheme used by a distributed key-value store yields reservoir partitions that do not correlate with the partitions of incoming batch. As illustrated in Figure 5(a), when items from a given partition of an incoming batch are inserted into the reservoir, the inserts touch many (if not all) partitions of the reservoir, incurring heavy network I/O. Secondly, key-value stores incur needless concurrency-control overhead. For each batch, D-R-TBS already carefully coordinates the deletes and inserts so that no two delete or insert operations access the same slots in the reservoir and there is no danger of write-write or read-write conflicts.

Co-partitioned reservoir: In the alternative approach, we implement a distributed in-memory data structure for the reservoir so as to ensure that the reservoir partitions coincide with the partitions from incoming batches, as shown in Figure 5(b). This can be achieved in spite of the unknown partitioning scheme of the streaming system. Specifically, the reservoir is initially empty, and all items in the reservoir are from the incoming batches. Therefore, if an item from a given partition of an incoming batch is always inserted into the corresponding "local" reservoir partition and deletes are also handled locally, then the co-partitioning and co-location of the reservoir and incoming batch partitions is automatic. For our experiments, we implemented the co-partitioned reservoir in Spark using the in-place updating technique for RDDs in [27]; see Appendix E of [16].

Note that, at any point in time, a given slot number in the reservoir maps to a specific partition ID and a position inside the partition. Thus the slot number for a given full item may change over time due to reservoir insertions and deletions. This does not cause any statistical issues, because the functioning of the set-based R-TBS algorithm is oblivious to specific slot numbers.

5.3 Choosing Items to Delete and Insert

In order to bound the reservoir size, D-R-TBS requires careful coordination when choosing the set of items to delete from, and insert into, the reservoir. At the same time, D-R-TBS must ensure the statistical correctness of random number generation and random permutation operations in the distributed environment. We consider two possible approaches.

Centralized decisions: In the most straightforward approach, the master makes centralized decisions about which items to delete and insert. For deletes, the driver generates slot numbers of the items in the reservoir to be deleted, which are then mapped to the actual data locations in a manner that depends on the representation of the reservoir (key-value store or co-partitioned reservoir). For inserts, the driver generates the slot numbers of the incoming items $\mathcal{B}_t$ at time t that need to be inserted into the reservoir. Suppose that $\mathcal{B}_t$ comprises $k \geq 1$ partitions. Each generated slot number $i \in \{1, 2, \ldots, |\mathcal{B}_t|\}$ is mapped to a partition p_i of the $\mathcal{B}_t$ (where $0 \leq p_i \leq k - 1$) and a position r_i inside partition p_i. Denote by Q the set of "item locations", i.e., the set of (p_i, r_i) pairs. In order to perform the inserts, we need to first retrieve the actual items based on the item locations. This can be achieved with a join-like operation between Q and $\mathcal{B}_t$, with the (p_i, r_i) pair matching the actual location of an item inside $\mathcal{B}_t$. To optimize this operation, we make Q a distributed data structure and use a customized partitioner to ensure that all pairs (p_i, r_i) with $p_i = j$ are co-located with partition j of $\mathcal{B}_t$ for $j = 0, 1, \ldots, k - 1$. Then a co-partitioned and co-located join can be carried out between Q and $\mathcal{B}_t$, as illustrated in Figure 6(a)

for $k = 3$. The resulting set of retrieved insert items, denoted as $\mathcal{S}$, is also co-partitioned with $\mathcal{B}_t$ as a by-product. After that, the actual deletes and inserts are then carried out depending on how reservoir is stored, as discussed below.

When the reservoir is implemented as a key-value store, the deletes can be directly applied based on the slot numbers. For inserts, the master takes each generated slot number of an item in $\mathcal{B}_t$ and chooses a companion destination slot number in the reservoir into which the $\mathcal{B}_t$ item will be inserted. This destination reservoir slot might currently be empty due to an earlier deletion, or might contain an item that will now be replaced by the newly inserted batch item. After the actual items to insert are retrieved as described previously, the destination slot numbers are used to put the items into the right locations in the key-value store.

When the co-partitioned reservoir is used, the delete slot numbers in the reservoir are mapped to (p_i, r_i) pairs of partitions of the reservoir and positions inside the partitions. As with inserts, we again use a customized partitioner for the set of pairs $\mathcal{R}$ such that deletes are co-located with the corresponding reservoir partitions. Then a join-like operation on $\mathcal{R}$ and the reservoir performs the actual delete operations on the reservoir. For inserts, we simply use another join-like operation on the set of retrieved insert items $\mathcal{S}$ and the reservoir to add the corresponding insert items to the co-located partition of the reservoir. In this approach, we don't need the master to generate destination reservoir slot numbers for these insert items, because we view the reservoir as a set when using co-partitioned reservoir data structure.

Distributed decisions: The above approach requires generating a large number of slot numbers inside the master, so we now explore an alternative approach that offloads the slot number generation to the workers while still ensuring the statistical correctness of the computation. This approach has the master choose only the number of deletes and inserts per worker according to appropriate multivariate hypergeometric distributions. For deletes, each worker chooses random victims from its local partition of the reservoir based on the number of deletes given by the master. For inserts, the worker randomly and uniformly selects items from its local partition of the incoming batch $\mathcal{B}_t$ given the number of inserts. Figure 6(b) depicts how the insert items are retrieved under this decentralized approach. We use the technique in [15] for parallel pseudo-random number generation.

Note that this distributed decision making approach works only when the co-partitioned reservoir data structure is used. This is because the key-value store representation of the reservoir requires a target reservoir slot number for each insert item from the incoming batch, and the target slot numbers have to be generated in such a way as to ensure that, after the deletes and inserts, all of the slot numbers are still unique and contiguous in the new reservoir. This requires a lot of coordination among the workers, which inhibits truly distributed decision making.

6 EXPERIMENTS

In this section, we study the empirical performance of D-R-TBS and D-T-TBS, and demonstrate the potential benefit of using them for model retraining in online model management. We implemented D-R-TBS and D-T-TBS on Spark (refer to Appendix E of [16] for implementation details).

Experimental Setup: All performance experiments were conducted on a cluster of 13 IBM System x iDataPlex dx340 servers. Each has two quad-core Intel Xeon E5540 2.8GHz processors and 32GB of RAM. Servers are interconnected using a 1Gbit Ethernet and each server runs Ubuntu Linux, Java 1.7 and Spark 1.6.

One server is dedicated to run the Spark coordinator and, each of the remaining 12 servers runs Spark workers. There is one worker per processor on each machine, and each worker is given all 4 cores to use, along with 8 GB of dedicated memory. All other Spark parameters are set to their default values. We used Memcached 1.4.33 as the key-value store in our experiments.

For all experiments, data was streamed in from HDFS using Spark Streaming's microbatches. We report run time per round as the average over 100 rounds, discarding the first round from this average because of Spark startup costs. Unless otherwise stated, each batch contains 10 million items, the target reservoir size is 20 million elements, and the decay parameter is $\lambda = 0.07$.
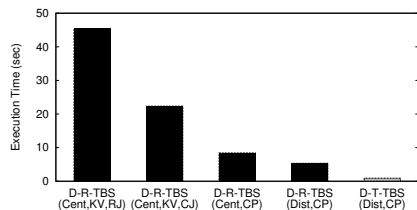

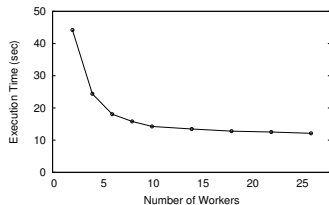**Figure 7: Per-batch distributed runtime comparison**
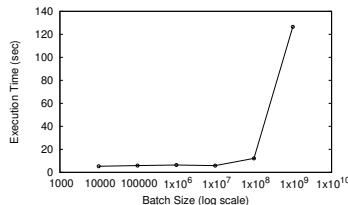

Figure 8: Scale out of D-R-TBS

Figure 9: Scale up of D-R-TBS

6.1 Runtime Performance

Comparison of TBS Implementations: Figure 7 shows the average runtime per batch for five different implementations of distributed TBS algorithms. The first four (colored black) are D-R-TBS implementations with different design choices: whether to use centralized or distributed decisions (abbreviated as "Cent" and "Dist", respectively) for choosing items to delete and insert, and whether to use key-value store for storing reservoir or co-partitioned reservoir (abbreviated as "KV" and "CP", respectively). The first two implementations both use the key-value store representation for reservoir together with the centralized decision strategy for determining inserts and deletes. They only differ in how the insert items are actually retrieved when subsampling the incoming batch. The first uses the standard repartition join (abbreviated as "RJ"), whereas the second uses the customized partitioner and co-located join (abbreviated as "CJ") as described in Section 5.3 and depicted in Figure 6(a). This optimization effectively cuts the network cost in half, but the KV representation of reservoir still requires the insert items to be written across the network to their corresponding reservoir location. The third implementation employs the co-partitioned reservoir instead, resulting in an significant speedup of over 2.6x. The fourth implementation further employs the distributed decision for choosing items to delete and insert. This yields a further 1.6x speedup. We use this D-R-TBS implementation in the remaining experiments.

The fifth implementation (colored grey) in Figure 7 is D-T-TBS using co-partitioned reservoir and the distributed strategy for choosing delete and insert items. Since, D-T-TBS is embarrassingly parallelizable, it's much faster than the best D-R-TBS implementation. But, as we discussed in Section 3, T-TBS only works under a very strong restriction on the data arrival rate,

and can suffer from occasional memory overflows; see Figure 1. In contrast, D-R-TBS is much more robust and works in realistic scenarios where it is hard to predict the data arrival rate.

Scalability of D-R-TBS: Figure 8 shows how D-R-TBS scales with the number of workers. We increased the batch size to 100 million items for this experiment. Initially, D-R-TBS scales out very nicely with the increasing number of workers. However, beyond 10 workers, the marginal benefit from additional workers is small, because the coordination and communication overheads, as well as the inherent Spark overhead, become prominent. For the same reasons, in the scale-up experiment in Figure 9, the runtime stays roughly constant until the batch size reaches 10 million items and increases sharply at 100 million items. This is because processing the streaming input and maintaining the sample start to dominate the coordination and communication overhead. With 10 workers, R-TBS can handle a data flow comprising 100 million items arriving approximately every 14 seconds.

6.2 Application: Classification using kNN

We now demonstrate the potential benefits of the R-TBS sampling scheme for periodically retraining representative ML models in the presence of evolving data. For each model and data set, we compare the quality of models retrained on the samples generated by R-TBS, a simple sliding window (SW), and uniform reservoir sampling (Unif). Due to limited space, we do not give quality results for T-TBS; we found that whenever it applies—i.e. when the mean batch size is known and constant—the quality is very similar to R-TBS, since they both use time-biased sampling.

Our first model is a kNN classifier, where a class is predicted for each item in an incoming batch by taking a majority vote of the classes of the k nearest neighbors in the current sample, based on Euclidean distance; the sample is then updated using the batch. To generate training data, we first generate 100 class centroids uniformly in a $[0, 80] \times [0, 80]$ rectangle. Each data item is then generated from a Gaussian mixture model and falls into one of the 100 classes. Over time, the data generation process operates in one of two "modes". In the "normal" mode, the frequency of items from any of the first 50 classes is five times higher than that of items in any of the second 50 classes. In the "abnormal" mode, the frequencies are five times lower. Thus the frequent and infrequent classes switch roles at a mode change. We generate each data point by randomly choosing a ground-truth class c_i with centroid (x_i, y_i) according to relative frequencies that depend upon the current mode, and then generating the data point's (x, y) coordinates independently as samples from $N(x_i, 1)$ and $N(y_i, 1)$. Here $N(\mu, \sigma)$ denotes the normal distribution with mean μ and standard deviation σ.

In this experiment, the batch sizes are deterministic with $b = 100$ items, and $k = 7$ neighbors for the kNN classifier. The reservoir size for both R-TBS and Unif is 1000, and SW contains the last 1000 items; thus all methods use the same amount of data for retraining. (We choose this value because it achieves near maximal classification accuracies for all techniques. In general, we choose sampling and ML parameters to achieve good learning performance while ensuring fair comparisons.) In each run, the sample is warmed up by processing 100 normal-mode batches before the classification task begins. Our experiments focus on two types of temporal patterns in the data, as described below.

Single change: Here we model the occurrence of a singular event. The data is generated in normal mode up to $t = 10$ (time is measured here in units after warm-up), then switches to abnormal

mode, and finally at $t = 20$ switches back to normal (Figure 10(a)). As can be seen, the misclassification rate (percentage of incorrect classifications) with R-TBS, SW and Unif all increase from around 18% to roughly 50% when the distribution becomes abnormal. Both R-TBS and SW adapt to the change, recovering to around 16% misclassification rate after $t = 16$, with SW adapting slightly better. In comparison, Unif does not adapt at all. But, when the distribution snaps back to normal, the error rate of SW rises sharply to 40% before gradually recovering, whereas R-TBS error rate stays low around 15% throughout. These results prove that R-TBS is indeed more robust: although slightly more sluggish than SW in adapting to changes, R-TBS avoids wild fluctuations in classification error as with SW.
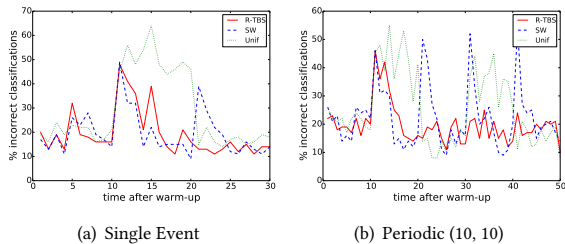

(a) Single Event   (b) Periodic (10, 10)

Figure 10: Misclassification rate (percent) for kNN

Periodic change: For this temporal pattern, the changes from normal to abnormal mode are periodic, with δ normal batches alternating with η abnormal batches, denoted as Periodic(δ, η), or $P(\delta, \eta)$ for short. Figures 10(b) shows the misclassification rate for Periodic(10, 10). Experiments on other periodic patterns (in Appendix F of [16]) demonstrate similar results. The robust behavior of R-TBS described above manifests itself even more clearly in the periodic setting. Note, for example, how R-TBS reacts significantly better to the renewed appearances of the abnormal mode. Observe that the first 30 batches of Periodic(10, 10) display the same behavior as in the single event experiment in Figure 10(a). We therefore focus primarily on the Periodic(10, 10) temporal pattern for the remaining experiments.

Robustness and Effect of Decay Parameter: In the context of online model management, we need a sampling scheme that delivers high overall prediction accuracy and, perhaps even more importantly, robust prediction performance over time. Large fluctuations in the accuracy can pose significant risks in applications, e.g., in critical IoT applications in the medical domain such as monitoring glucose levels for predicting hyperglycemia events. To assess the robustness of the performance results across different sampling schemes, we use a standard risk measure called *expected shortfall (ES)* [22, p. 70]. ES measures downside risk, focusing on worst-case scenarios. Specifically, the z% ES is the average value of the worst z% of cases.

For each of 30 runs and for each sampling scheme, we compute the 10% ES of the misclassification rate (expressed as a percentage) starting from $t = 20$, since all three sampling schemes perform poorly (as would be expected) during the first mode change, which finishes at $t = 20$. Table 1 lists both the *accuracy*, measured in terms of the average misclassification rate, and the *robustness*, measured as the average 10% ES, of the kNN classifier over 30 runs across different temporal patterns. To demonstrate the effect of the decay parameter λ on model performance, we also include numbers for different λ values in Table 1.

In terms of accuracy, Unif is always the worst by a large margin. R-TBS and SW have similar accuracies, with R-TBS having a

slight edge in most cases. On the other hand, for robustness, SW is almost always the worst, with ES ranging from 1.4x to 2.7x the maximum ES (over different λ values) of R-TBS. Mostly, Unif is also significantly worse than R-TBS, with ES ratios ranging from 1.4x to 1.7x. The only exception is the single-event pattern: since the data remains in normal mode after the abnormal period, time biasing becomes unimportant and Unif performs well. In general, R-TBS provides both better accuracy and robustness in almost all cases. The relative performance of the sampling schemes in terms of accuracy and robustness tend to be consistent across temporal patterns. Table 1 also shows that different λ values affect the accuracy and robustness, however, R-TBS provides superior results over a fairly wide range of λ values.

Varying batch size: We now examine model quality when the batch sizes are no longer constant. Overall, the results look similar to those for constant batch size. For example, Figure 11(a) shows results for a Uniform(0,200) batch-size distribution, and Figure 11(b) shows results for a deterministic batch size that grows at a rate of 2% after warm-up. In both experiments, $\lambda = 0.07$ and the data pattern is Periodic(10, 10). These figures demonstrate the robust performance of R-TBS in the presence of varying data arrival rates. Similarly, the average accuracy and robustness over 30 runs resembles the results in Table 1. For example, pick $\lambda = 0.07$ and a Periodic(10, 10) pattern. Then, the misclassification rate under uniform/growing batch sizes is 1.16x/1.14x that of R-TBS for SW, and 1.47x/1.40x for Unif. In addition, the ES is 1.82x/1.98x that of R-TBS for SW, and 1.76x/1.78x for Unif.

Table 1: Accuracy and robustness of kNN performance

λ	Single Event		P(10,10)		P(20,10)		P(30,10)	
	Miss%	ES	Miss%	ES	Miss%	ES	Miss%	ES
0.05	19.8	**17.7**	18.2	24.2	17.9	28.2	15.5	31.6
0.07	19.1	18.7	17.4	**23.2**	17.2	**28.1**	**14.9**	**31.0**
0.10	**18.0**	20.0	**16.6**	24.1	**16.6**	29.9	15.1	31.0
SW	19.2	53.3	19.0	49.8	18.8	47.3	16.5	44.5
Unif	25.6	19.3	25.4	42.3	25.0	43.2	21.0	47.6

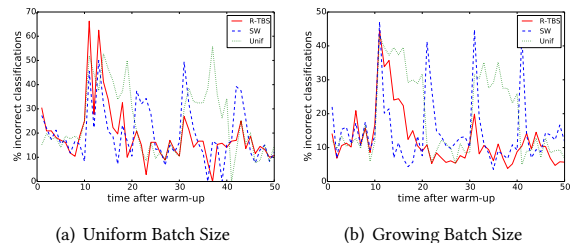(a) Uniform Batch Size   (b) Growing Batch Size

Figure 11: Varying batch sizes for kNN classifier

6.3 Application: Linear Regression

We now assess the effectiveness of R-TBS for retraining regression models. The experimental setup is similar to kNN, with data generated in "normal" and "abnormal" modes. In both modes, data items are generated from the standard linear regression model $y = b_1 x_1 + b_2 x_2 + \epsilon$, with the noise term ϵ distributed according to a $N(0, 1)$ distribution. In normal mode, $(b_1, b_2) = (4.2, -0.4)$ and in abnormal mode, $(b_1, b_2) = (-3.6, 3.8)$. In both modes, x_1 and x_2 are generated according to Uniform$(0, 1)$ distribution. As before, the experiment starts with a warm-up of 100 "normal" mode batches and each batch contains 100 items.

Saturated samples: Figure 12(a) shows the performance of R-TBS, SW, and Unif for the Periodic(10, 10) pattern with a maximum sample size of 1000 for each technique. We note that, for this sample size and temporal pattern, the R-TBS sample is always saturated. (This is also true for all of the prior experiments.) The results echo that of the previous section, with R-TBS exhibiting

slightly better prediction accuracy on average, and significantly better robustness, than the other methods. The mean square errors (MSEs) across all data points for R-TBS, Unif, and SW are 3.51, 4.43, 4.02 respectively, and their 10% ES of the MSEs are 6.04, 10.05, 10.94 respectively.

Unsaturated Samples: We now investigate the case of unsaturated samples for R-TBS. We increase the target sample size to $n = 1600$. With a constant batch size of 100, and a decay rate $\lambda = 0.07$, the reservoir of R-TBS is never full, stabilizing at 1479 items, whereas Unif and SW both have a full sample of 1600 items.

For the Periodic$(10, 10)$ pattern, shown in Figure 12(b), SW has a window size large enough to keep some data from older time periods (up to 16 batches ago), making SW's robustness comparable to R-TBS (ES of 5.86 for SW and 5.97 for R-TBS). However, this amalgamation of old data also hurts its overall accuracy, with MSE rising to 4.17, as opposed to 3.50 for R-TBS. In comparison, the shape of R-TBS remains almost unchanged from Figure 12(a), and Unif behaves as poorly as before. When the pattern changes to Periodic$(16, 16)$ as shown in Figure 12(c), SW doesn't contain enough old data, making its prediction performance suffer from huge fluctuations again, and the superiority of R-TBS is more prominent. In both cases, R-TBS provides the best overall performance, despite having a smaller sample size. This backs up our earlier claim that more data is not always better. A smaller but more balanced sample with good ratios of old and new data can provide better prediction performance than a large but unbalanced sample.

6.4 Application: Naive Bayes

In our final experiment, we evaluate the performance of R-TBS for retraining Naive Bayes models with the Usenet2 dataset (mlkd. csd.auth.gr/concept_drift.html), which was used in [18] to study classifiers coping with recurring contexts in data streams. This dataset contains a stream of 1500 messages on different topics from the 20 News Groups Collections [21]. They are sequentially presented to a simulated user who marks whether a message is interesting or not. The user's interest changes after every 300 messages. More details of the dataset can be found in [18].

Following [18], we use Naive Bayes with a bag of words model, and set the optimal parameters for SW with maximum sample size of 300 and batch size of 50. Since this dataset is rather small and contexts change frequently, we use the optimal value of 0.3 for λ. We find through experiments that R-TBS displays higher prediction accuracy for all λ in the range of $[0.1, 0.5]$, so precise tuning of λ is not critical. In addition, there is not enough data to warm up the models on different sampling schemes, so we report the model performance on all the 30 batches. Similarly, we report 20% ES for this dataset, due to the limited number of batches.

The results are shown in Figure 13. The misprediction rate for R-TBS, SW, and Unif are 26.5%, 30.0%, and 29.5%; and the 20% ES values are 43.3%, 52.7%, and 42.7%. Importantly, for this dataset the changes in the underlying data patterns are less pronounced than in the previous two experiments. Despite this, SW fluctuates wildly, yielding inferior accuracy and robustness. In contrast, Unif barely reacts to the context changes. As a result, Unif is very slightly better than R-TBS with respect to robustness, but at the price of lower overall accuracy. Thus, R-TBS is generally more accurate under mild fluctuations in data patterns, and its superior robustness properties manifest themselves as the changes become more pronounced.

7 RELATED WORK

Time-decay and sampling: Work on sampling with unequal probabilities goes back to at least Lahiri's 1951 paper [20]. A growing interest in streaming scenarios with weighted and decaying items began in the mid-2000's, with most of that work focused on computing specific aggregates from such streams, such as heavy-hitters, subset sums, and quantiles; see, e.g., [2, 7, 8]. The first papers on time-biased reservoir sampling with exponential decay are due to Aggarwal [1] and Efraimidis and Spirakis [12]; batch arrivals are not considered in these works. As discussed in Section 1, the sampling schemes in [1] are tied to item sequence numbers rather than the wall clock times on which we focus; the latter are more natural when dealing with time-varying data arrival rates.

Cormode et al. [9] propose a time biased reservoir sampling algorithm based on the A-Res weighted sampling scheme proposed in [12]. Rather than enforcing (1), the algorithm enforces the (different) A-Res biasing scheme. In more detail, if s_i denotes the element at slot i in the reservoir, then the algorithm in [12] implements a scheme where an item x is chosen to be at slot $i + 1$ in the reservoir with probability $w_x / (\sum_{j=1}^{x} w_j - \sum_{j=1}^{i} w_{s_j})$. From the form of this equation, it becomes clear that resulting sampling algorithm violates (1). Indeed, Efraimidis [11] gives some numerical examples illustrating this point (in his comparison of the A-Res and A-Chao algorithms). Again, we would argue that the constraint on appearance probabilities in (1) is easier to understand in the setting of model management than the foregoing constraint on initial acceptance probabilities.

The closest solution to ours adapts the weighted sampling algorithm of Chao [5] to batches and time decay; we call the resulting algorithm B-Chao and describe it in Appendix D of [16]. Unfortunately, as discussed, the relation in (1) is violated both during the initial fill-up phase and whenever the data arrival rate becomes slow relative to the decay rate, so that the sample contains "overweight" items. Including overweight items causes over-representation of older items, thus potentially degrading predictive accuracy. The root of the issue is that the sample size is nondecreasing over time. The R-TBS algorithm is the first algorithm to correctly (and optimally) deal with "underflows" by allowing the sample to shrink—thus handling data streams whose flow rates vary unrestrictedly over continuous time. The current paper also explicitly handles batch arrivals and explores parallel implementation issues. The VarOpt sampling algorithm of Cohen et al. [6]—which was developed to solve the specific problem of estimating "subset sums"—can also be modified to our setting. The resulting algorithm is more efficient than Chao, but as stated in [6], it has the same statistical properties, and hence does not satisfy (1).

Model management: A key goal of our work is to support model management; see [13] for a survey on methods for detecting changing data—also called "concept drift" in the setting of online learning—and for adapting models to deal with drift. As mentioned previously, one possibility is to re-engineer the learning algorithm. This has been done, for example, with support-vector machines (SVMs) by developing incremental versions of the basic SVM algorithm [4] and by adjusting the training data in an SVM-specific manner, such as by adjusting example weights as in Klinkenberg [19]. Klinkenberg also considers using curated data selection to learn over concept drift, finding that weighted data selection also improves the performance of learners. Our approach of model retraining using time-biased samples follows this
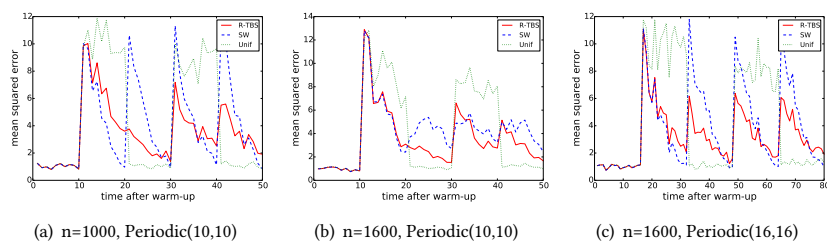
| (a) n=1000, Periodic(10,10) | (b) n=1600, Periodic(10,10) | (c) n=1600, Periodic(16,16) |

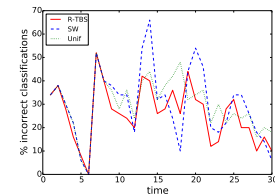**Figure 12: Mean square error for linear regression**

Figure 13: Misclassification rate (percent) for Naive Bayes

latter approach, and is appealing in that it is simple and applies to a large class of machine-learning models. The recently proposed Velox system for model management [10] ties together online learning and statistical techniques for detecting concept drift. After detecting drift through poor model performance, Velox kicks off batch learning algorithms to retrain the model. Our approach to model management is complementary to the work in [10] and could potentially be used in a system like Velox to help deployed models recover from poor performance more quickly. The developers of the recent MacroBase system [3] have incorporated a time-biased sampling approach to model retraining, for identifying and explaining outliers in fast data streams. MacroBase essentially uses Chao's algorithm, and so could potentially benefit from the R-TBS algorithm to enforce the inclusion criterion (1) in the presence of highly variable data arrival rates.

8 CONCLUSION

Our experiments with classification and regression algorithms, together with the prior work on graph analytics in [27], indicate the potential usefulness of periodic retraining over time-biased samples to help ML algorithms deal with evolving data streams without requiring algorithmic re-engineering. To this end we have developed and analyzed several time-biased sampling algorithms that are of independent interest. In particular, the R-TBS algorithm allows simultaneous control of both the item-inclusion probabilities and the sample size, even when the data arrival rate is unknown and can vary arbitrarily. R-TBS also maximizes the expected sample size and minimizes sample-size variability over all possible bounded-size algorithms with exponential decay. Using techniques from [9], we intend to generalize these properties of R-TBS to hold under arbitrary forms of temporal decay.

We have also provided techniques for distributed implementation of R-TBS and T-TBS, and have shown that use of time-biased sampling together with periodic model retraining can improve model robustness in the face of abnormal events and periodic behavior in the data. In settings where (i) the mean data arrival rate is known and (roughly) constant, as with a fixed set of sensors, and (ii) occasional sample overflows can be easily dealt with by allocating extra memory, we recommend use of T-TBS to precisely control item-inclusion probabilities. In many applications, however, we expect that either (i) or (ii) will violated, in which case we recommend the use of R-TBS. Our experiments showed that R-TBS is superior to sliding windows over a range of λ values, and hence does not require highly precise parameter tuning; this may be because time-biased sampling avoids the all-or-nothing item inclusion mechanism inherent in sliding windows.

REFERENCES

[1] Charu C. Aggarwal. 2006. On biased reservoir sampling in the presence of stream evolution. In *VLDB*. VLDB Endowment, 607–618.
[2] Noga Alon, Nick Duffield, Carsten Lund, and Mikkel Thorup. 2005. Estimating arbitrary subset sums with few probes. In *PODS*. ACM, 317–325.
[3] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. 2017. MacroBase: Prioritizing Attention in Fast Data. In *SIGMOD*. 541–556.
[4] Gert Cauwenberghs and Tomaso Poggio. 2000. Incremental and Decremental Support Vector Machine Learning. In *NIPS*. 388–394.
[5] M. T. Chao. 1982. A general purpose unequal probability sampling plan. *Biometrika* (1982), 653–656.
[6] Edith Cohen, Nick G. Duffield, Haim Kaplan, Carsten Lund, and Mikkel Thorup. 2011. Efficient Stream Sampling for Variance-Optimal Estimation of Subset Sums. *SIAM J. Comput.* 40, 5 (2011), 1402–1431.
[7] Edith Cohen and Martin J Strauss. 2006. Maintaining time-decaying stream aggregates. *J. Algo.* 59, 1 (2006), 19–36.
[8] Graham Cormode, Flip Korn, and Srikanta Tirthapura. 2008. Exponentially decayed aggregates on data streams. In *ICDE*. IEEE, 1379–1381.
[9] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. 2009. Forward decay: A practical time decay model for streaming systems. In *ICDE*. IEEE, 138–149.
[10] Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. 2015. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. In *CIDR*.
[11] Pavlos S. Efraimidis. 2015. Weighted Random Sampling over Data Streams. In *Algorithms, Probability, Networks, and Games*, Christos D. Zaroliagis, Grammati E. Pantziou, and Spyros C. Kontogiannis (Eds.). Springer, 183–195.
[12] Pavlos S Efraimidis and Paul G Spirakis. 2006. Weighted random sampling with a reservoir. *Inf. Process. Lett.* 97, 5 (2006), 181–185.
[13] João Gama, Indre Zliobaite, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM Comput. Surv.* 46, 4 (2014), 44.
[14] Rainer Gemulla and Wolfgang Lehner. 2008. Sampling time-based sliding windows in bounded space. In *SIGMOD*. 379–392.
[15] Hiroshi Haramoto, Makoto Matsumoto, Takuji Nishimura, François Panneton, and Pierre L'Ecuyer. 2008. Efficient Jump Ahead for 2-Linear Random Number Generators. *INFORMS Journal on Computing* 20(3) (2008), 385–390.
[16] Brian Hentschel, Peter J. Haas, and Yuanyuan Tian. 2018. Temporally-Biased Sampling for Online Model Management. *CoRR* abs/1801.09709 (2018). https://arxiv.org/abs/1801.09709
[17] Voratas Kachitvichyanukul and Bruce W. Schmeiser. 1988. Binomial Random Variate Generation. *Commun. ACM* 31, 2 (1988), 216–222.
[18] Ioannis Katakis, Grigorios Tsoumakas, and I Vlahavas. 2008. An Ensemble of Classifiers for coping with Recurring Contexts in Data Streams. (01 2008), 763-764 pages.
[19] Ralf Klinkenberg. 2004. Learning drifting concepts: Example selection vs. example weighting. *Intell. Data Anal.* 8, 3 (2004), 281–300.
[20] D. B. Lahiri. 1951. A method of sample selection providing unbiased ratio estimates. *Bull. Intl .Statist. Inst.* 33 (1951), 133–140.
[21] M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml
[22] Alexander J. McNeil, Rüdiger Frey, and Paul Embrechts. 2015. *Quantitative Risk Management: Concepts, Techniques and Tools* (second ed.).
[23] Memcached. 2017. (2017). Retrieved 2017-07-13 from https://memcached.org
[24] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable stream computation in the cloud. In *EuroSys*.
[25] Redis. 2017. (2017). Retrieved 2017-07-13 from https://redis.io
[26] Andrew Whitmore, Anurag Agarwal, and Li Da Xu. 2015. The Internet of Things – A survey of topics and trends. *Information Systems Frontiers* 17, 2 (2015), 261–274.
[27] Wenlei Xie, Yuanyuan Tian, Yannis Sismanis, Andrey Balmin, and Peter J. Haas. 2015. Dynamic interaction graphs with probabilistic edge decay. In *ICDE*. 1143–1154.
[28] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*.