Online Model Management via Temporally Biased Sampling

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. 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 control over the decay rate and a guaranteed upper bound on the sample size. The R-TBS and T-TBS schemes are of independent interest, extending the known set of unequal-probability sampling schemes. We discuss distributed implementation strategies; experiments in Spark show that our approach can increase machine learning accuracy and 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 comprise dynamically changing sensor streams, 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 methods such as regression models can indeed be re-engineered so that the parameters are time-varying, but for many popular non-parametric algorithms such as k-nearest neighbors (kNN) classifiers, decision trees, random forests, gradient boosted machines, and so on, it is not at all clear how re-engineering can be accomplished. The 2017 Kaggle Data Science Survey [1] indicates that a substantial portion of the models that developers use in industry are non-parametric. 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 periodically retrains on the data from, e.g., 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 [10]. 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 [15], 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 [15] in the setting of graph analysis algorithms, and has recently been adopted in the MacroBase system [5]. 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" [9]; 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. We assume that items arrive in *batches* $\mathcal{B}_1, \mathcal{B}_2, \ldots$, at time points $t_1, t_2, \cdots$, where each batch contains 0 or more items. Our goal is to generate a sequence $\{S_k\}_{k\geq 1}$, where S_k is a sample of the items that have arrived at or prior to time t_k.

.

These samples should be biased towards recent items, in the following sense. For $1 \leq i \leq k$, denote by $\alpha_{i,k} = t_k - t_i$ the *age* at time t_k of an item belonging to batch $\mathcal{B}_i$. Then for arbitrary times $t_i \leq t_j \leq t_k$ and items $x \in \mathcal{B}_i$ and $y \in \mathcal{B}_j$,

$$\Pr[x \in S_k] / \Pr[y \in S_k] = f(\alpha_{i,k})/f(\alpha_{j,k}) = e^{-\lambda(t_j - t_i)}, \quad (1)$$

where $f(\alpha) = e^{-\lambda\alpha}$ is the exponential *decay function*. (We briefly discuss other decay functions in Section 7.) Thus items with a given timestamp are sampled uniformly, and items with different timestamps are handled in a carefully controlled manner, such that the appearance probability for an item of age α is proportional to $f(\alpha)$. The criterion in (1), which is expressed in terms of wall-clock time, is natural and appealing in applications and, importantly, is interpretable and understandable to users. As discussed in [7, 15], the decay function can be chosen to meet application-specific criteria. If training data is available, λ can also be chosen to maximize accuracy of a specified ML model via cross validation combined with grid search—in our experiments, we found empirically that accuracy tended to be a quasiconvex function of λ, which bodes well for automatic optimization methods such as stochastic gradient descent.

Prior work: It is surprisingly hard to both enforce (1) and to bound the sample size. As discussed in detail in an extended version of this paper [12], 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 [8] and on Chao's algorithm [6]. A-Res enforces conditions on the *acceptance* probabilities of items; this leads to appearance probabilities that, unlike (1), are both hard to compute and not intuitive. In [12], we show that Chao's 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. The second type of algorithm, due to Aggarwal [4], can only control appearance probabilities based on the indices of the data items and not wall-clock time; this can be suboptimal in the presence of time-varying arrival rates. 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 relatively simple algorithm that generalizes the Bernoulli sampling scheme in [15] (which we call B-TBS). T-TBS allows complete control over the decay rate (expressed in wall-clock time) and probabilistically maintains a target sample size. T-TBS is easy to implement and highly scalable when applicable, but only works under the strong restriction that the mean sizes of the arriving batches are constant over time and known a priori. There are scenarios where T-TBS might be a good choice (see Section 3), but many applications have non-constant, unknown mean batch sizes and/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.

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 minimize I/O, avoid unnecessary concurrency control, and make decentralized decisions about which items to insert into, or delete from, the reservoir. Our experiments (Section 6) demonstrate the efficiency and effectiveness of our techniques.

2. BACKGROUND

For the remainder of the paper, assume that batches arrive at regular time intervals, so that $t_i = i\Delta$ for some $\Delta > 0$. All items that arrive in an interval $\big((k-1)\Delta, k\Delta\big]$ are treated as if they arrived at time $k\Delta$, i.e., at the end of the interval, so that all items in batch $\mathcal{B}_i$ have time stamp $i\Delta$. It follows that the age at time t_k of an item that arrived at time $t_i \leq t_k$ is simply $\alpha_{i,k} = (k-i)\Delta$.

In this section, we briefly review two classical sampling schemes whose properties we will combine in the R-TBS algorithm. A detailed description of the two algorithms can be found in [12].

Bernoulli Time-Biased Sampling (B-TBS): One simple sampling scheme [15] processes each incoming batch by first downsampling the current sample and then accepting all items in the batch into the sample with probability 1. Downsampling is accomplished by flipping a coin independently for each item in the sample: an item is retained in the sample with probability $p = e^{-\lambda\Delta}$ and removed with probability $1 - p$. In [12], we prove that B-TBS enforces the relation in (1) as required. Unfortunately, the user cannot independently control the expected sample size, which is completely determined by λ and the sizes of the incoming batches.

Batched Reservoir Sampling (B-RS): The standard reservoir sampling algorithm with sample size n accepts the first n items into the sample with probability 1. For $k > n$, the kth item is accepted with probability n/k, overwriting a random victim; in [12], we show how to modify the algorithm to handle batch arrivals. Although B-RS guarantees an upper bound on the sample size, it does not support time biasing. R-TBS (Section 4) maintains a bounded reservoir as in B-RS while supporting time-biased sampling as in B-TBS.

3. TARGETED-SIZE TBS

We now describe the 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 write $B_k = |\mathcal{B}_k|$ for $k \geq 1$, and assume that the batch sizes $\{B_k\}_{k\geq 1}$ are independent and identically distributed (i.i.d.) as a random variable B, with common mean $b = \mathrm{E}[B] < \infty$.

The Algorithm: The idea behind the algorithm is to downsample to remove older items, as in B-TBS, but to also downsample the incoming batches at a rate q such that n becomes the "equilibrium" sample size, while also ensuring that (1) holds. Setting $p = e^{-\lambda\Delta}$ as before, a simple calculation shows that, for any $q \in (0, 1]$, $\Pr[x \in S_k] = qp^{k-i} = qe^{-\lambda(t_k - t_i)}$, and (1) follows immediately.

To choose q, suppose that $|S_{k-1}| = n$ and we are about to

process batch $\mathcal{B}_k$. The expected number of items that will be removed is $(1-p)n$ and the expected number of accepted items is qb. For n to be an equilibrium point, we equate these terms and solve for q to obtain $q = n(1-p)/b$. Note that, even if we always accept all items in an arriving batch (i.e., $q = 1$) but the resulting expected inflow b is less than the expected outflow $n(1-p)$, the sample will consistently fall below n, and so we require that $b \geq n(1-p)$.

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

```
1  p = e^{-λΔ}: decay factor
2  n: target sample size
3  b: assumed mean batch size such that b ≥ n(1 − p)
4  Initialize: S ← ∅; q ← n(1 − p)/b
5  for k ← 1, 2, ... do
6      m ← Binomial(|S|, p)              //simulate |S| trials
7      S ← Sample(S, m)                  //retain m random elements
8      l ← Binomial(|B_k|, q)            //simulate |B_k| trials
9      B'_k ← Sample(B_k, l)             //downsample new batch
10     S ← S ∪ {B'_k}                    //add new items to sample
11     output S
```

The resulting sampling scheme is given as Algorithm 1; it precisely controls inclusion probabilities in accordance with (1) while constantly pushing the sample size toward the target value n. Conceptually, at each time t_k, T-TBS first downsamples the current sample by independently flipping a coin for each item with retention probability p. T-TBS then downsamples the arriving batch $\mathcal{B}_k$ via independent coin flips; an item in $\mathcal{B}_k$ is inserted into the sample with probability q. For efficiency, the algorithm exploits the fact that for j independent trials, each having success probability r, the total number of successes has a binomial distribution with parameters j and r. Thus, in lines 6 and 8, the algorithm simulates the coin tosses by directly generating the number of successes m or l—which can be done using standard algorithms [13]—and then retaining m or l 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 $\textsc{Sample}(A, m)$ returns an empty sample if $A = \emptyset$, $m = 0$, or both.

Sample-Size Properties: Theorem 1 below precisely describes the sample size behavior of T-TBS, which directly impacts memory requirements, efficiency of memory usage, and ML model retraining time; see [12] for a statement and proof of this result in the setting of general decay functions. (The proof exploits the fact that $\{|S_k|\}_{k\geq 0}$ is a Markov chain.) Denote by $C_k = |S_k|$ the sample size at time t_k and by $\bar{b} \geq 1$ the maximum possible batch size, so that $\Pr[B \leq \bar{b}] = 1$. Also set $\sigma^2 = bq(1+p-q)/(1-p^2) \in (0, \infty)$ and write $p = e^{-\lambda\Delta}$ as before. Write "i.o." to denote that an event occurs "infinitely often", i.e., for infinitely many values of k, and write "w.p.1" for "with probability 1".

Assertions (i)–(iii) of Theorem 1 deal with the distribution of the sample size after a large number of batches have been processed. Specifically, by (i) and (ii), the expected sample size $\mathrm{E}[C_k]$ approximately equals the target size n and the variance of C_k approximately equals the finite constant σ^2, which depends on b, p, and q; note that the convergence of $\mathrm{E}[C_k]$ to n happens exponentially fast. By (iii), the probability that the sample size deviates from n by more than $100\epsilon\%$ is exponentially small when k or n is large, provided that the batch sizes are bounded.

Whereas Assertions (i)–(iii) describe average behavior over many sampling runs, Assertions (iv) and (v) concern the behavior of the successive sample sizes during an individual sampling run. By (iv), any sample size can be attained with positive probability, so one potential type of bad behavior might occur if, with positive probability, the sample size is unstable in that it drifts off to $+\infty$ over time. By (v), if the batch sizes are bounded, then such unstable behavior is ruled out: with probability 1, the sample-size process is stable in that every possible sample-size value occurs infinitely often, with finite expected time between visits. Moreover, the average sample size—averaged over times $t_1, t_2, \ldots, t_k$—converges to n with probability 1 as k becomes large. On the negative side, it follows that, for a given sampling run, the sample size will repeatedly—though infrequently, since the expected sample size at any time point is finite—become arbitrarily large, even if the average behavior is good. This result shows that, even in the most stable case, the sample-size control provided by T-TBS is incomplete, and thus motivates the more complex R-TBS algorithm given in the next section. This sample-size fragility is amplified when batch sizes fluctuate in a non-predicable way, as often happens in practice, and T-TBS can break down; see Section 6.

Theorem 1. *If $\{B_k\}_{k\geq 1}$ are i.i.d. with mean $b < \infty$, then*

(i) $\mathrm{E}[C_k] = n(1-p^k) \uparrow n$ *as* $k \to \infty$;

(ii) $\mathrm{Var}[C_k] \to \sigma^2$ *as* $k \to \infty$;

(iii) if $\bar{b} < \infty$, *then (a)* $\Pr[C_k \geq (1+\epsilon)n] \leq e^{-O(kn^2\epsilon^2)}$ *for all* $\epsilon, k > 0$ *and (b)* $\Pr[C_k \leq (1-\epsilon)n] \leq e^{-O(kn^2)}$ *for any* $\epsilon \in (0,1)$ *and sufficiently large* k;

(iv) $\forall m \geq 0$, $\exists k \geq 0$ *such that* $\Pr[C_k \geq m] > 0$;

(v) if $\bar{b} < \infty$, *then (a)* $\Pr[C_k = m \text{ i.o.}] = 1$ *for all* $m \geq 0$, *(b) the expected times between successive visits to state* m *are uniformly bounded for any* $m \geq 0$, *and (c)* $\lim_{k\to\infty}(1/k)\sum_{i=0}^{k} C_i = n$ *w.p.1.*

Despite the fluctuations in sample size, 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 relatively 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.

4. RESERVOIR-BASED TBS

Our new reservoir-based time-biased sampling algorithm (R-TBS) combines the best features of T-TBS and B-RS, controlling the decay rate while ensuring that the sample never overflows. Importantly, unlike T-TBS, the R-TBS algorithm can handle any sequence of batch sizes. The proofs of all the theorems in this section can be found in [12].

4.1 Item Weights and Latent Samples

To precisely control the sample size in the presence of decay, we essentially need to handle samples having "fractional size". We do this via "item weights" and "latent samples".

Item weights: In R-TBS, the *weight* of an item of age α is given by $f(\alpha) = e^{-\lambda\alpha}$; note that a newly arrived item has a weight of $f(0) = 1$. As discussed later, R-TBS ensures that the probability that an item appears in the sample is proportional to its weight. All items arriving at the same time have the same weight, so that the *total weight* of all items seen up through time t_k is $W_k = \sum_{i=1}^{k} |\mathcal{B}_i| f(\alpha_{i,k})$.

ALGORITHM 2: Generating a sample from a latent sample

```
1 L = (A, π, C): latent sample
2 U ← UNIFORM()
3 if U ≤ frac(C) then S ← A ∪ π else S ← A
4 return S
```

Latent samples: The other key concept for R-TBS is the notion of a *latent sample*, which formalizes the idea of a sample of fractional size. Formally, 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; π is nonempty if and only if $C > \lfloor C \rfloor$.

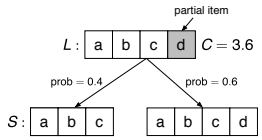


Figure 1: Latent sample L (sample weight $C = 3.6$) and possible realized samples

We randomly generate a sample S from L by sampling as described in Algorithm 2, where $\text{frac}(x) = x - \lfloor x \rfloor$; e.g., see Figure 1. In the pseudocode, the function UNIFORM() generates a random number uniformly distributed on $[0, 1]$. Each full item is included with probability 1 and the partial item is included with probability $\text{frac}(C)$, and it is easy to show that $\mathrm{E}[|S|] = C$. By allowing at most one partial item, we minimize the latent sample's footprint: $|A \cup \pi| \leq \lfloor C \rfloor + 1$. Importantly, if the weight C of a latent sample L is an integer, then L contains no partial item, and the sample S generated from L via Algorithm 2 is unique and contains exactly C items.

Downsampling: Besides extracting an actual sample from a latent sample, another key operation on latent samples is *downsampling* (Algorithm 3). For $\theta \in [0, 1]$, the goal of downsampling $L = (A, \pi, C)$ is to obtain an new latent sample $L' = (A', \pi', \theta C)$ such that, if we generate S and S' from C and C' via Algorithm 2, we have

$$\Pr[x \in S'] = \theta \Pr[x \in S] \tag{2}$$

for all $x \in S$. Thus all of the the appearance probabilities, as well as the sample weight (and hence expected sample size), are scaled down by a factor of θ. R-TBS uses downsampling to remove sample items that either decay or are overwritten by arriving items, and also to initially filter the items in an arriving batch.

In the pseudocode, the subroutine SWAP1(A, π) moves a randomly selected item from A to π and moves the current item in π (if any) to A. Similarly, MOVE1(A, π) moves a randomly selected item from A to π, replacing the current item in π (if any).

ALGORITHM 3: Downsampling

```
1  L = (A, π, C): input latent sample
2  θ: scaling factor with θ ∈ [0, 1]
3  if C = 0 then return L' = (∅, ∅, 0)                    //sample is empty
4  U ← UNIFORM(); C' = θC
5  if ⌊C'⌋ = 0 then                                       //no full items retained
6      if U > frac(C)/C then
7          (A', π') ← SWAP1(A, π)
8      A' ← ∅
9  else if 0 < ⌊C'⌋ = ⌊C⌋ then                            //no items deleted
10     if U > (1 − θ frac(C))/(1 − frac(C')) then
11         (A', π') ← SWAP1(A, π)
12 else                                                   //items deleted: 0 < ⌊C'⌋ < ⌊C⌋
13     if U ≤ θ frac(C) then
14         A' ← SAMPLE(A, ⌊C'⌋)
15         (A', π') ← SWAP1(A', π)
16     else
17         A' ← SAMPLE(A, ⌊C'⌋ + 1)
18         (A', π') ← MOVE1(A', π)
19 if C' = ⌊C'⌋ then                                      //no fractional item
20     π' ← ∅
21 return L' = (A', π', C')
```

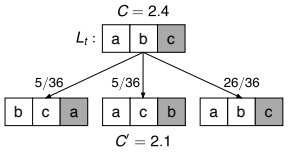


Figure 2: Downsampling example: $C = 2.4 \to C' = 2.1$

To gain some intuition for why the algorithm works, consider a special case (the if-statement in line 9): the goal is to form a latent sample $L' = (A', \pi', \theta C)$ from a latent sample $L = (A, \pi, C)$, where L and L' have the same number of full items and each has one partial item; e.g., $C = 2.4$ and $C' = 2.1$. In this case, the partial item $x^* \in \pi$ either becomes full by being swapped into A' or remains as the partial item for L'. The symmetric treatment of the items in A ensures that appearance probabilities are scaled down uniformly. Denoting by β the probability of *not* swapping, we have $P[x^* \in S'] = \beta \cdot \text{frac}(C') + (1 - \rho) \cdot 1$. On the other hand, (2) implies that $P[x^* \in S'] = \theta\,\text{frac}(C)$. Equating these expression shows that β must equal the expression on the right side of the inequality on line 10; see Figure 2. The other possible downsampling scenarios are described in [12].

Theorem 2. *For $\theta \in [0, 1]$, let $L' = (A', \pi', \theta C)$ 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 Algorithm 2. Then $\Pr[x \in S'] = \theta \Pr[x \in S]$ for all $x \in A \cup \pi$.*

The union operator: We also need to take the union of disjoint latent samples while preserving the inclusion probabilities for each. Two latent samples $L_1 = (A_1, \pi_1, C_1)$ and $L_2 = (A_2, \pi_2, C_2)$ are *disjoint* if $(A_1 \cup \pi_1) \cap (A_2 \cup \pi_2) = \emptyset$. The pseudocode for the union operation is given as Algorithm 4. The idea is to add all full items to the combined latent sample. If there are partials items in L_1 and L_2, then we transform them to either a single partial item, a full item, or a full plus partial item, depending on the values of $\text{frac}(C_1)$ and $\text{frac}(C_2)$. Such transformations are done in a manner that preserves the appearance probabilities. We obtain the

union of multiple latent samples by iterating Algorithm 4; for latent samples $L_1, \ldots, L_k$, we denote the resulting latent sample by $\bigcup_{j=1}^{k} L_j$.

ALGORITHM 4: Union

```
1  L_1 = (A_1, π_1, C_1): fractional sample of size C_1
2  L_2 = (A_2, π_2, C_2): fractional sample of size C_2
3  C ← C_1 + C_2
4  U ← UNIFORM()
5  if frac(C_1) + frac(C_2) < 1 then
6  |  A ← A_1 ∪ A_2
7  |  if U ≤ frac(C_1)/(frac(C_1) + frac(C_2)) then π ← π_1 else
   |    π ← π_2
8  else if frac(C_1) + frac(C_2) = 1 then
9  |  π ← ∅
10 |  if U ≤ frac(C_1) then A ← A_1 ∪ A_2 ∪ π_1 else
   |    A ← A_1 ∪ A_2 ∪ π_2
11 else //frac(C_1) + frac(C_2) > 1
12 |  if U ≤ (1 − frac(C_1)) / [(1 − frac(C_1)) + (1 − frac(C_2))] then
13 |  |  π = π_1
14 |  |  A ← A_1 ∪ A_2 ∪ π_2
15 |  else
16 |  |  π = π_2
17 |  |  A ← A_1 ∪ A_2 ∪ π_1
18 return L=(A,π, C)
```

Theorem 3. *Let $L_1 = (A_1, \pi_1, C_1)$ and $L_2 = (A_2, \pi_2, C_2)$, be disjoint latent samples, and let $L = (A, \pi, C)$ be the latent sample produced from L_1 and L_2 by Algorithm 4. Let S_1, S_2, and S be random samples generated from L_1, and L_2, and L via Algorithm 2. Then (i) $C = C_1 + C_2 = \mathrm{E}[S]$, (ii) $\forall x \in L_1$, $\Pr[x \in S] = \Pr[x \in S_1]$, and (iii) $\forall x \in L_2$, $\Pr[x \in S] = \Pr[x \in S_2]$.*

4.2 The R-TBS Algorithm

The algorithm: R-TBS is given as Algorithm 5. The algorithm generates a sequence of latent samples $\{L_k\}_{k\geq 1}$ and from these generates a sequence of actual samples $\{S_k\}_{k\geq 1}$ that are returned to the user. In the algorithm, the functions GETSAMPLE, DOWNSAMPLE, and UNION execute the operations described in Algorithms 2, 3, and 4.

The goal of the algorithm is to ensure that

$$\Pr[x \in S_k] = \rho_k f(\alpha_{i,k}) \tag{3}$$

for all $k \geq 1$, $i \leq k$, and $x \in \mathcal{B}_i$, where $f(\alpha) = e^{-\lambda\alpha}$ and $\{\rho_k\}_{k\geq 1}$ are the successive values of the variable ρ during a run of the algorithm. Clearly, (3) immediately implies (1). We choose ρ_k to make the sample size as large as possible without exceeding n. Indeed, we show in Theorem 4 below that $C_k = \rho_k W_k$ for all k, and therefore set $\rho_k = \min(1, n/W_k)$, so that $C_k = \min(W_k, n)$. Thus if $W_k < n$, then the sample weight is at its maximum possible value W_k, leading to the maximum possible sample size of $\lfloor W_k \rfloor$ or $\lceil W_k \rceil$. If $W_k \geq n$, then the sample weight, and hence the sample size, is capped at n.

R-TBS functions similarly to classic reservoir sampling. When a new batch of items arrives, all of the items are accepted if the cumulative set of (weighted) items plus the batch items fit in the reservoir of size n ($\rho = 1$ in line 9). If the total item weight exceeds n just before the batch arrives, then a random subset of old items is removed from the sample via downsampling ($\rho/\rho' < 1$ in line 8, over and above decay θ) and a random subset of the arriving items, also filtered via downsampling ($\rho < 1$ in line 9), take their place

ALGORITHM 5: Reservoir-based TBS (R-TBS)

```
1  θ = e^{−λΔ}: decay factor
2  n: maximum sample size
3  Initialize: W ← 0; A ← ∅; π ← ∅; C ← 0; ρ ← 1
4  for k ← 1, 2, ... do
5  |  W ← θW + |B_k|                          //update total weight
6  |  ρ' ← ρ
7  |  ρ ← min(1, n/W)                          //update ρ
8  |  (A, π, C) ← DOWNSAMPLE((A, π, C), (ρ/ρ')θ)  //decay old items
9  |  L_0 ← DOWNSAMPLE((B_k, ∅, |B_k|), ρ)       //take in new items
10 |  L ← UNION(L_0, (A, π, C))                 //combine old and new items
11 |  S ← GETSAMPLE(L)
12 |  output S
```

(line 10). The algorithm also correctly handles the intermediate case where all cumulative items fit, but inserting all arriving items would cause the reservoir to overflow; in this case, only some of the new items overwrite sample items.

Algorithm properties: Theorem 4(i) below asserts that R-TBS satisfies (3) and hence (1), thereby maintaining the correct inclusion probabilities. Theorem 4(ii) implies that the sample size and stability are maximized, as formalized in Theorem 5 below. Finally, the assertion in Theorem 4(iii) ensures that the inclusion probabilities for a given item are nonincreasing over time. This is crucial, since otherwise we might have to recover an item that was previously deleted from the sample, which is impossible.

Theorem 4. *Let $\{L_k = (A_k, \pi_k, C_k)\}_{k\geq 1}$ and $\{S_k\}_{k\geq 1}$ be a sequence of latent samples and samples, respectively, produced by Algorithm 5 and define $\rho_k = \min(1, n/W_k)$. Then (i) $\Pr[x \in S_k] = \rho_k f(\alpha_{i,k})$ for all $1 \leq i \leq k$ and $x \in \mathcal{B}_i$, (ii) $C_k = \rho_k W_k$ for all k, and (iii) $\rho_k f(\alpha_{i,k}) \leq \rho_{k-1} f(\alpha_{i,k-1})$ for all $1 \leq i < k$.*

A sample S_k is *unsaturated* if $C_k < n$ and *saturated* if $C_k = |S_k| = n$; note that $W_k < n$ if and only if S_k is unsaturated. Theorem 5 asserts that, among all sampling schemes 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 ML results (via more training data) and greater stability in both result quality and retraining costs.

Theorem 5. *Let H be any sampling algorithm for exponential decay that satisfies* (1) *and denote by S_k and S_k^H the samples produced at (arbitrary) time t_k by R-TBS and H. Then (i) if $W_k < n$, then $\mathrm{E}[|S_k^H|] \leq \mathrm{E}[|S_k|]$, and (ii) if $\mathrm{E}[|S_k^H|] = \mathrm{E}[|S_k|]$, then $\mathrm{Var}[|S_k^H|] \geq \mathrm{Var}[|S_k|]$.*

Indeed, (1) implies that, for any $t_i \leq t_k$ and $x \in \mathcal{B}_i$, the inclusion probability $\Pr[x \in S_k^H]$ must be of the form $r_k^H f(\alpha_{i,k})$ for some function r_k^H independent of i. Taking $i = k$, we see that $r_k^H \leq 1$. For R-TBS with $C_k < n$, Theorem 4 implies that $r_k^H = \rho_k = C_k/W_k = 1$, so that $\Pr[x \in S_k^H] \leq \Pr[x \in S_k]$, proving (i). To prove (ii), observe that, over all possible sample-size distributions having mean value equal to $C_k = \mathrm{E}[|S_k|]$, the variance is minimized by concentrating all of the probability mass onto $\lfloor C_k \rfloor$ and $\lceil C_k \rceil$, and this is precisely the sample-size distribution attained by R-TBS.

5. DISTRIBUTED TBS ALGORITHMS

We now describe the distributed implementation of T-TBS and R-TBS, denoted as D-T-TBS and D-R-TBS.

Overview of D-T-TBS: The D-T-TBS implementation is very similar to the simple distributed Bernoulli time-biased sampling algorithm in [15]. It is embarrassingly parallel, requiring no coordination. At each time point t_k, each worker in the cluster subsamples its partition of the sample with probability p, subsamples its partition of $\mathcal{B}_k$ 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 is not embarrassingly parallel. D-R-TBS first needs to aggregate the local partition sizes for the incoming batch $\mathcal{B}_k$ to compute the total batch size $|\mathcal{B}_k|$ and calculate the new total weight W_k. Then, based on $|\mathcal{B}_k|$, W_k, and the current sample weight C_k, D-R-TBS computes the downsample rate for the items in the reservoir, as well as the downsample rate for the items in $\mathcal{B}_k$. After that, D-R-TBS chooses the items in the reservoir to delete through a DOWNSAMPLE operation, selects items in $\mathcal{B}_k$ (also via DOWNSAMPLE), inserts the selected items into the reservoir (via UNION), and finally generates the sample (via GETSAMPLE). The expensive operations DOWNSAMPLE, UNION, and GETSAMPLE are all performed in a distributed manner. They each require the master to coordinate among the workers. GETSAMPLE and UNION operations are relatively straightforward. The most challenging part of D-R-TBS lies in choosing items to delete from the reservoir and selecting new items to insert; we introduce two alternative approaches in Section 5.2. The implementation details for D-T-TBS are mostly subsumed by those for D-R-TBS, so we focus on the latter.

5.1 Distributed Data Structures

There are two important data structures in D-R-TBS: the incoming batch and the reservoir. Conceptually, we view an incoming batch $\mathcal{B}_k$ as an array of slots numbered from 1 through $|\mathcal{B}_k|$, and the reservoir L as an array of slots numbered from 1 through $\lfloor C_k \rfloor$ 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, s, maps to a pair (p_s, r_s), where p_s is the partition ID and r_s is the position inside the partition.

Incoming batches usually come 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 D-R-TBS. 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 natural approach implements the reservoir using an off-the-shelf distributed in-memory key-value (KV) store, such as Redis [3] or Memcached [2]. Each item in the reservoir is stored as a KV pair, with the slot number as the key and the item as the value. The partial item has a special slot number such as -1. Inserts and deletes to the reservoir naturally translate into put and delete operations to the KV store.

There are three major limitations to this approach. First, the hash-based or range-based data-partitioning scheme used by a distributed KV store yields reservoir partitions that do not correlate with the partitions of incoming batch. 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. Second, KV stores incur unnecessary 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. Finally, the KV store approach requires an explicit slot number for each item. As a result, D-R-TBS needs to take extra care to make sure that after deletes and inserts of reservoir items, the slot numbers are still unique and contiguous, e.g. by recycling the slot numbers of deleted items for new inserts. The burden of keeping track of delete and insert slot numbers falls on the master node.

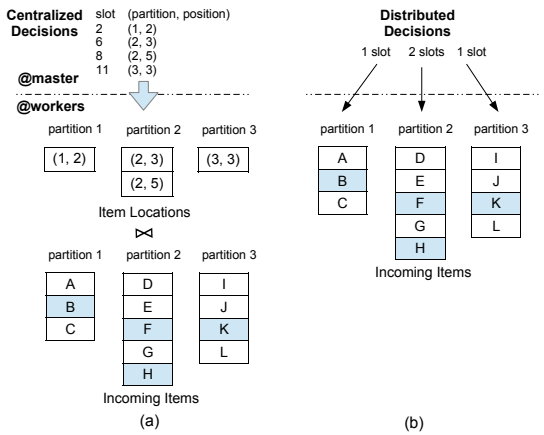


Figure 3: Retrieving insert items

Co-partitioned reservoir: An alternative approach implements a distributed in-memory data structure for the reservoir so as to ensure that the reservoir partitions coincide with the partitions from incoming batches. 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-partition 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 [15]; see [12].

Note that, with co-partitioned reservoir, the mapping between a specific full item and its current slot number may change over time due to insertions and deletions. This does not cause any statistical issues, because the set-based R-TBS algorithm is oblivious to specific slot numbers.

5.2 Choosing Items to Delete and Insert

To bound the reservoir size, D-R-TBS carefully coordinates among workers in choosing the items to delete from, and insert into, the reservoir. At the same time, it must ensure the statistical correctness of random number generation and random permutation operations in the distributed environment. We consider two possible approaches. Our description focuses on the co-partitioned reservoir; see [12] for the KV store implementation.

Centralized decisions: In the most straightforward ap-

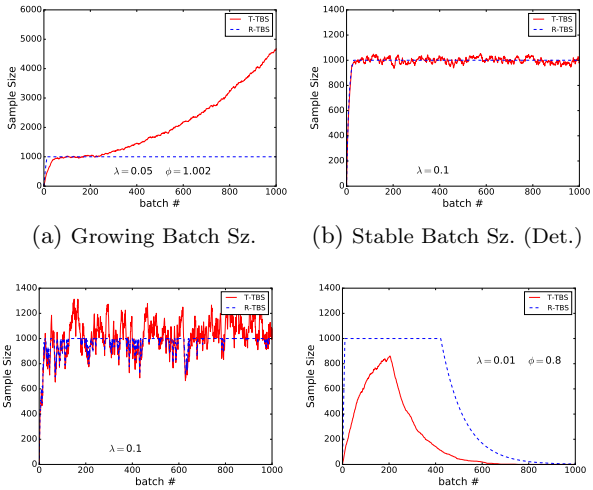
proach, the master makes centralized decisions. For inserts, the master generates the slot numbers of the incoming items $\mathcal{B}_k$ at time t_k that need to be inserted into the reservoir. Suppose that $\mathcal{B}_k$ comprises $m \geq 1$ partitions. Each generated slot number $s \in \{1, 2, \ldots, |\mathcal{B}_k|\}$ is mapped to an item location indicated by (p_s, r_s). Denote by $\mathcal{Q}$ the set of item locations, i.e., the set of (p_s, r_s) pairs. In order to perform the inserts, D-R-TBS needs to first retrieve the actual items based on the item locations. This can be achieved with a join-like operation between $\mathcal{Q}$ and $\mathcal{B}_k$, with the (p_s, r_s) pair matching the actual location of an item inside $\mathcal{B}_k$. To optimize this operation, we make $\mathcal{Q}$ a distributed data structure and use a customized partitioner to ensure that all pairs (p_s, r_s) with $p_s = j$ are co-located with partition j of $\mathcal{B}_k$ for $j = 1, 2, \ldots, m$. Then a co-partitioned and co-located join can be carried out between $\mathcal{Q}$ and $\mathcal{B}_k$, as illustrated in Figure 3(a) for $m = 3$. The resulting set of retrieved insert items, denoted as $\mathcal{S}$, is also co-partitioned with $\mathcal{B}_k$ as a by-product. After that, the actual inserts are carried out depending on the reservoir representation (KV store or co-partitioned reservoir). For the co-partitioned reservoir, we simply use a join-like operation on $\mathcal{S}$ and the reservoir L to add the corresponding insert items to the co-located partition of L. Similarly, for deletes, the master generates slot numbers of the reservoir items to be deleted, then deletes are executed based on the reservoir representation. For the co-partitioned reservoir, we again use a customized partitioner for the set of (p_s, r_s) pairs that represent the slot numbers, denoted as $\mathcal{R}$, such that deletes are co-located with the corresponding L partitions. Then a join-like operation on $\mathcal{R}$ and L performs the actual delete operations on the reservoir.

Distributed decisions: The above approach requires the master to generate a large number of slot numbers, 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 an appropriate multivariate hypergeometric distribution. 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 $\mathcal{B}_k$ given the number of inserts. Figure 3(b) depicts how the insert items are retrieved under this decentralized approach. We use the technique in [11] for parallel pseudo-random number generation.

The foregoing distributed decision making approach works only when the co-partitioned reservoir is used. This is because the KV store approach 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

We briefly highlight some of our experimental results; see [12] for details and additional experiments. We implemented R-TBS and T-TBS on Spark. Data was streamed in from HDFS using Spark Streaming's microbatches. All performance experiments were conducted on a cluster of 9 ProLiant DL160 G6 servers interconnected by 1 Gbit Ethernet. Decay occurs according to a time scale such that the batch-arrival interval is $\Delta = 1$ in the decay formulas.



(a) Growing Batch Sz. (b) Stable Batch Sz. (Det.)

(c) Stable Batch Sz. (Unif.) (d) Decaying Batch Sz.

Figure 4: Sample size behavior of T-TBS and R-TBS

Sample size Behavior: Figures 4 shows sample size behavior of T-TBS and R-TBS under a variety of batch-size regimes. In Figure 4(a), 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 $k = 200$, the batch size starts to increase: $B_{k+1} = \phi B_k$, where $\phi = 1.002$. This leads to an overflowing sample for T-TBS, whereas R-TBS maintains a constant sample size. Even in a stable batch-size regime with batch sizes either constant (Figure 4(b); $B_k \equiv 100$ with $\lambda = 0.1$) or fluctuating (Figure 4(c); B_k uniform on $[0, 200]$), R-TBS can maintain a bounded sample size, whereas the sample size under T-TBS fluctuates per Theorem 1; as in Theorem 5, the R-TBS unsaturated sample size is always larger than than for T-TBS. For $\phi = 0.8$, so that the batch sizes start to shrink at $k = 200$, Figure 4(d) shows that R-TBS is more robust to sample underflows.

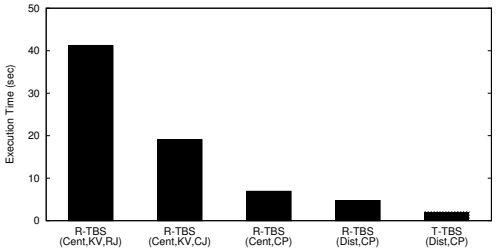


Figure 5: Per-batch distributed runtime comparison

Runtime performance: Figure 5 shows the average runtime per batch for five different implementations of distributed TBS algorithms. Each batch contains 10 million items. The first four are D-R-TBS implementations with different design choices: whether to use centralized or distributed decisions in choosing items to insert and delete (abbreviated as "Cent" and "Dist", respectively), whether to implement the reservoir using a key-value store or a co-partitioned reservoir scheme (abbreviated as "KV" and "CP"), and whether to subsample the incoming batch using the standard repartition join or using a copartitioned join (abbreviated as "RJ" and "CJ") under the centralized decision scheme. As can be seen, the best implementation is almost

an order of magnitude faster than the worst. Since D-T-TBS is embarrassingly parallelizable, it is much faster than the best D-R-TBS implementation (see rightmost bar). But, as discussed in Section 3, T-TBS only works under very strong restrictions on the data arrival process, and can suffer from occasional memory overflows.

We have also conducted scalability experiments and evaluated the impact of the decay factor as well as batch-size skew on the runtime performance; see [12]. With 8 workers, our implementation of R-TBS can handle 100 million items arriving approximately every 16 seconds.

Application to ML models: We first compare the performance of R-TBS, simple sliding windows (SW), and uniform sampling (Unif) when applied to a kNN classifier that predicts a class for each item in an incoming batch and then updates the sample. We use 100 classes, and 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. The sample 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.

Figure 6 shows the misclassification rates for the three sampling methods under a periodic pattern of 10 normal batches alternating with 10 abnormal batches, denoted as P(10, 10). When the data distribution first becomes abnormal at $t = 10$, the misclassification rates under all sampling schemes increase sharply. R-TBS and SW adapt to the new mode, with SW adapting slightly faster. (Unif never adapts at all.) After the first mode change, however, R-TBS "remembers" both normal and abnormal values, and thereby becomes much more robust to subsequent mode changes, whereas SW continues to overreact with wild fluctuations.

Table 1 displays both the accuracy and robustness of Unif, SW, and R-TBS (using several values of λ) over 30 runs. Accuracy is measured in terms of the average misclassification rate, and robustness is measured as the average 10% *expected shortfall (ES)* , i.e., the average value of the worst 10% of cases [14, p. 70]. Results are shown for a set of temporal patterns that include several periodic patterns and a "single event" comprising one normal-abnormal-normal cycle. As can be seen, R-TBS and SW have similar accuracies, and Unif is always the worst by a large margin. R-TBS is always best in terms of robustness and SW is always the worst, with ES values 1.5 to 2.5 times higher than for R-TBS. Unif also does poorly in terms of robustness, except for the single event, since the data remains in normal mode after the abnormal period and time biasing becomes unimportant. Overall, R-TBS provides superior accuracy and robustness, and this performance edge is fairly stable across a wide range of λ values.

Table 1: Accuracy and robustness of kNN performance

	Single Event		P(10,10)		P(20,10)		P(30,10)	
λ	Miss%	ES	Miss%	ES	Miss%	ES	Miss%	ES
0.05	17.1	**16.8**	16.1	22.1	15.3	24.4	15.1	25.9
0.07	16.5	17.3	15.3	**21.3**	14.9	**24.0**	**14.4**	**25.2**
0.10	**15.7**	18.5	**15.1**	22.1	**14.7**	24.9	14.7	26.9
SW	19.2	42.1	17.1	41.7	16.1	39.8	15.9	38.3
Unif	21.3	18.3	25.4	34.8	19.6	35.7	19.0	35.8

Figure 7 shows similar results for an experiment involving a regression model. Interestingly, the parameters were such that the R-TBS sample was never full, whereas SW and Unif were always full. This shows that a smaller sample with good ratios of old and new data can provide better prediction performance than a larger but temporally unbalanced sample.

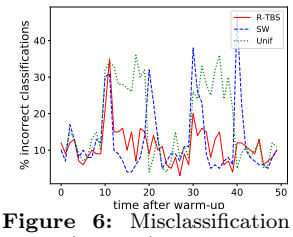


Figure 6: Misclassification rate (percent) for kNN: n=1000, P(10, 10)

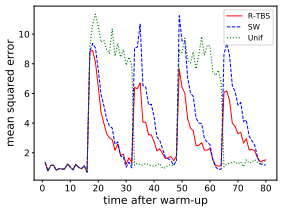


Figure 7: Mean square error for linear regression: n=1600, P(16, 16)

7. CONCLUSION & FUTURE WORK

Our experiments with ML models and graph analytics [15], indicate the usefulness of periodic retraining over time-biased samples to help ML algorithms robustly deal with evolving data streams without requiring algorithmic re-engineering.

In ongoing work [12], we have extended our R-TBS and T-TBS sampling schemes to arbitrary decay functions. Theory and algorithms are more complex in this setting because, unlike the exponential case, decay factors now vary by age, so item ages must be tracked. R-TBS then satisfies (1) only approximately, with an error that can be made arbitrarily small by increasing the sample footprint. There is also a well defined trade-off between sample footprint and sample-size stability and saturation. Interesting future directions are to apply our ideas to other types of streaming analytics, and to develop end-to-end solutions via drift-detection techniques.

8. REFERENCES

[1] An interactive deep dive into the Kaggle data science survey. https://www.kaggle.com/sudalairajkumar/an-interactive-deep-dive-into-survey-results.

[2] Memcached. https://memcached.org.

[3] Redis. https://redis.io.

[4] C. C. Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *VLDB*, pages 607–618. VLDB Endowment, 2006.

[5] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. MacroBase: Prioritizing attention in fast data. In *SIGMOD*, pages 541–556, 2017.

[6] M. T. Chao. A general purpose unequal probability sampling plan. *Biometrika*, pages 653–656, 1982.

[7] E. Cohen and M. J. Strauss. Maintaining time-decaying stream aggregates. *J. Algo.*, 59(1):19–36, 2006.

[8] P. S. Efraimidis and P. G. Spirakis. Weighted random sampling with a reservoir. *Inf. Process. Lett.*, 97(5):181–185, 2006.

[9] J. Gama, I. Zliobaite, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44, 2014.

[10] R. Gemulla and W. Lehner. Sampling time-based sliding windows in bounded space. In *SIGMOD*, pages 379–392, 2008.

[11] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. Efficient Jump Ahead for 2-Linear Random Number Generators. *INFORMS Journal on Computing*, 20(3):385–390, 2008.

[12] B. Hentschel, P. J. Haas, and Y. Tian. Online Model Management via Temporally Biased Sampling. *CoRR*, abs/1906.05677, 2019.

[13] V. Kachitvichyanukul and B. W. Schmeiser. Binomial random variate generation. *Commun. ACM*, 31(2):216–222, 1988.

[14] A. J. McNeil, R. Frey, and P. Embrechts. *Quantitative Risk Management: Concepts, Techniques and Tools*. Second edition, 2015.

[15] W. Xie, Y. Tian, Y. Sismanis, A. Balmin, and P. J. Haas. Dynamic interaction graphs with probabilistic edge decay. In *ICDE*, pages 1143–1154, 2015.