Efficient estimation of AUC in a sliding window

Nikolaj Tatti

F-Secure, Helsinki, Finland, nikolaj.tatti@gmail.com

Abstract. In many applications, monitoring area under the ROC curve (AUC) in a sliding window over a data stream is a natural way of detecting changes in the system. The drawback is that computing AUC in a sliding window is expensive, especially if the window size is large and the data flow is significant.
In this paper we propose a scheme for maintaining an approximate AUC in a sliding window of length k. More specifically, we propose an algorithm that, given ϵ, estimates AUC within $\epsilon/2$, and can maintain this estimate in $\mathcal{O}((\log k)/\epsilon)$ time, per update, as the window slides. This provides a speed-up over the exact computation of AUC, which requires $\mathcal{O}(k)$ time, per update. The speed-up becomes more significant as the size of the window increases. Our estimate is based on grouping the data points together, and using these groups to calculate AUC. The grouping is designed carefully such that (*i*) the groups are small enough, so that the error stays small, (*ii*) the number of groups is small, so that enumerating them is not expensive, and (*iii*) the definition is flexible enough so that we can maintain the groups efficiently.
Our experimental evaluation demonstrates that the average approximation error in practice is much smaller than the approximation guarantee $\epsilon/2$, and that we can achieve significant speed-ups with only a modest sacrifice in accuracy.

Keywords: AUC · approximation guarantee · sliding window

1 Introduction

Consider monitoring prediction performance in a stream of data points. That is, we first receive a data point d without the label, and we predict the missing label with a score of s, after the prediction we receive the true label ℓ. We are interested in monitoring how well s predicts ℓ as the stream evolves over time.

A good example of such a task is a monitoring system for corporate computers that detects abnormal behavior based on event logs. Here the positive label represents an abnormal event that requires a closer inspection, and such a label can be given, for example, by an expert or triggered automatically. The produced score can be used for decision making, and can be a specific feature or a simple statistic, or the result of some classifier, such as logistic regression. It is vital to monitor such a system continuously to notice breakdowns early. Possible causes may be changes in the underlying distribution or a system failure, due to the software update.

A natural choice to monitor the predictive power of a real-valued score is the area under the ROC curve (AUC) in a sliding window over the stream of events as proposed by Brzezinski and Stefanowski [5]. Unfortunately, maintaining the exact AUC requires $\mathcal{O}(k)$ time, per new event, where k is the size of the window. This may be too expensive if k is large and the rate of the events is significant.

In this paper we propose a technique for estimating AUC efficiently in a sliding window. Namely, we propose an approximation scheme that has $\epsilon/2$ approximation error guarantee while having $\mathcal{O}((\log k)/\epsilon)$ update time. That is, the scheme provides a trade-off between the accuracy and computational complexity.

Our approach is straightforward. Computing AUC exactly requires sorting data points and summing over all data points (see Eq. 1 for the exact formula). Maintaining points sorted can be done using binary search trees. However, estimating the sum requires additional tricks. We approach the problem by grouping neighboring data points together, that is, treating them as if the classifier given them the same score.

The key step is to design a grouping such that 3 properties hold at the same time: (i) the groups are small enough so that the relative error is small, more specifically, $\left|\widetilde{auc} - auc\right|/auc \leq \epsilon/2$, ($ii$) the number of groups is small enough, more specifically, it should be in $\mathcal{O}((\log k)/\epsilon)$, and ($iii$) the definition should be flexible enough so that we can do quick updates whenever points arrive or leave the sliding window.

Roughly speaking, in order to accommodate all 3 demands, we will maintain the groups with the two following properties: (i) the number of positive labels in a group is less than or equal to $(1 + \epsilon)$ than the total number of positive labels in all the previous groups, (ii) the number of positive labels in a group, *and the next group*, is larger than $(1 + \epsilon)$ than the total number of positive labels in all the previous groups. The first property will yield the approximation guarantee, while the second property guarantees that the number of groups remains small. Moreover, these properties are flexible enough so we can perform update procedures quickly.

The rest of the paper is organized as follows. We begin by reminding ourselves the definition of AUC in Section 2. Updating the groups of data points quickly requires several auxiliary structures, which we introduce in Section 3. We then proceed describing AUC estimation in Section 4. The related work is given in Section 5. In Section 6, we demonstrate that the relative error in practice is much smaller than the guaranteed bound, as well as, study the trade-off between the error and the computational cost. Finally, we conclude the paper with discussion in Section 7.

2 Preliminaries

We start with the definition of AUC, and provide a formula for computing it.

Assume that we are given a set of k pairs $W = (s_i, \ell_i)_i^k$, where ℓ_i is the true label of the ith instance, $\ell_i = 0, 1$, and s_i is score produced by the classification algorithm. The larger s_i, the more we believe that ℓ_i should be 0.[1]

In order to predict a label, we need a threshold σ, and predict that $\ell_i = 0$ if $s_i \geq \sigma$, and $\ell_i = 1$ otherwise. The ROC curve is obtained by varying σ and plotting true positive rate as a function of false positive rate. AUC is the area under the ROC curve. To compute AUC, we can use the following formula. Let

$$n(s) = |\{i \mid s_i = s, \ell_i = 0\}| \quad \text{and} \quad p(s) = |\{i \mid s_i = s, \ell_i = 1\}|$$

be the counts of labels with a score of s. Define also $hp(s) = \sum_{t<s} p(t)$. Then,

$$auc = \frac{1}{A} \sum_s (hp(s) + \frac{1}{2} p(s)) n(s)\,, \tag{1}$$

where $A = |\{i \mid \ell_i = 0\}||\{i \mid \ell_i = 1\}|$ is the normalization factor. Eq. 1 can be computed in $\mathcal{O}(k \log k + k)$ time by first sorting W, computing hp, and enumerating over the sum of Eq. 1.

In a streaming setting, W is a sliding window, and our goal is to compute AUC as W slides over a stream of predictions and labels.

3 Supporting data structures for estimating AUC

In this section we introduce supporting data structures that are needed to compute AUC in a streaming setting. Additional structures and the actual logic for computing AUC are given in the next section. We begin by describing the data structures, then follow with introducing the needed query operations, and finally finish with explaining the update procedures.

3.1 Data structures

Assume that we have a sequence of pairs $W = (s_i, \ell_i)_{i=1}^k$, where s_i is the score produced by the classifier, and $\ell_i \in \{0, 1\}$ is the true label.

We store W in a red-black tree T sorted by the scores s_i. Let $v \in T$ be a node in T. We will denote the corresponding score of v by $s(v)$. We store and maintain the following information:

- Counter $p(v) = |\{i \mid s_i = s(v)\,, \ell_i = 1\}|$, number of pairs in W with a score $s(v)$ and a positive label.
- Counter $n(v) = |\{i \mid s_i = s(v)\,, \ell_i = 0\}|$, number of pairs in W with a score $s(v)$ and a negative label.
- Counter $accpos(v)$, the total sum of $p(w)$, where w ranges over all descendant nodes of v in T, including v itself.
- Counter $accneg(v)$, the total sum of $n(w)$, where w ranges over all descendant nodes of v in T, including v itself.

[1] We chose this direction due to the notational convenience.

For simplicity, we will add two sentinel nodes to T. The first node will have a score of $-\infty$ and the second node has a score ∞. We will assume that the actual entries will never achieve these values. Both sentinel nodes have 0 positive labels and 0 negative labels.

Note that if the scores s_i are unique, then we have either $p(v) = 1$, $n(v) = 0$, or $p(v) = 0$, $n(v) = 1$. However, if there are duplicate scores, then we may have any integer combinations.

In addition to red-black trees, we need to maintain several linked lists, for which we will now introduce the notation. Assume that we are given a subset U of nodes in T. We would like to maintain U in a linked list L, sorted by the score. For that we will need two pointers for each node $u \in U$, namely, $\mathit{next}(u; L)$ indicating the next node in L, and $\mathit{prev}(u; L)$ indicating the previous node in L. Let $u \in U$ and assume that $v = \mathit{next}(u; L)$ exists. Let

$$B = \{w \in T \mid s(u) \leq s(w) < s(v)\}$$

be the set of nodes in T between u and v. We define

$$\mathit{gp}(u; L) = \sum_{w \in B} p(w) \quad \text{and} \quad \mathit{gn}(u; L) = \sum_{w \in B} n(w)$$

to be the total sums of the labels in the gap B. We will refer to L as *weighted linked list*. Note that deleting an element from L and maintaining the gap counters can be done in constant time. We will refer to the deletion algorithm by $\textsc{Remove}(L, v)$. Moreover, adding a new element, say v, to L after u can be also done in constant time, if we already know the total sums of labels, say p and n, between u and v. We will refer to the insertion algorithm by $\textsc{Add}(L, u, v, p, n)$.

We say that the node $v \in T$ is *positive*, if $p(v) > 0$. Similarly, we say that the node v is *negative*, if $n(v) > 0$. Note that v can be both negative and positive.

We maintain all positive nodes in a weighted linked list, which we will refer as P. Finally, we also store all positive nodes in its own dedicated red-black tree, denoted by TP. For simplicity, we also store the sentinel nodes of T in P and TP as the first and the last nodes.

3.2 Query procedures

The first query that we need is $\textsc{MaxPos}(s)$, returning the *positive* node v with the largest score such that $s(v) \leq s$. This can be done in $\mathcal{O}(\log k)$ time using TP, where k is the number of elements in the window.

Maintaining $\mathit{accpos}(v)$ and $\mathit{accneg}(v)$ allows us to query a cumulative sums of counts. Specifically, given a score s, we are interested in

$$\mathit{hp}(v) = \sum_{v \in T \mid s(v) < s} p(v) \quad \text{and} \quad \mathit{hn}(v) = \sum_{v \in T \mid s(v) < s} n(v) \quad . \tag{2}$$

We can compute both of these sums with $\textsc{HeadStats}(s)$, given in Algorithm 1.

Algorithm 1: HeadStats(s), computes the cumulative counts of labels, $hp(v)$ and $hn(v)$. Assumes that a node in T with a score s exists.

```
1  $hp \leftarrow 0$; $hn \leftarrow 0$;
2  $v \leftarrow$ root of $T$;
3  while true do
4      if $s(v) < s$ then
5          $v \leftarrow left(v)$;
6      else
7          if $left(v)$ then
8              $hp \leftarrow hp + accpos(left(v))$;
9              $hn \leftarrow hn + accneg(left(v))$;
10         if $s(v) = s$ then
11             return $hp, hn$;
12         else
13             $hp \leftarrow hp + p(v)$;
14             $hn \leftarrow hn + n(v)$;
15             $v \leftarrow right(v)$;
```

The algorithm assumes that there is a node in T containing s, and proceeds to find it; during the search whenever we go the right branch we add the accumulative sums from the left branch. We omit the trivial proof of correctness. Since the tree is balanced, the running time of HeadStats(s) is $\mathcal{O}(\log k)$, where k is the number of entries in the window.

3.3 Update procedures

We now continue to the maintenance procedures as we slide the window. This comes down to two procedures: (i) removing an entry from the window and (ii) adding an entry to the window.

We will first describe removing an entry with a positive label and a score s. First we will find the node, say v, with the score s, and reduce the counter $p(v)$ by 1. We will need to update the *accpos* counters. However, we only need to do it for the ancestors of v, and there are only $\mathcal{O}(\log k)$ of them, where k is the number of entries in the window, since T is balanced. We also reduce $gp(v; P)$ by 1. In the process, v may become non-positive, and we need to delete it from TP as well as from P.

Finally, if $p(v) = n(v) = 0$, we need to delete the node from T. This may result in rebalancing of the tree, and during the balancing we need to make sure that the counters *accpos* and *accneg* are properly updated. Luckily, the red-black tree balancing is based on left and right rotations. During these rotations it is easy to maintain the counters without additional costs.

We will refer to this procedure as RemoveTreePos(s) and the pseudo-code is given in Algorithm 2. RemoveTreePos(s) runs in $\mathcal{O}(\log k)$ time.

Algorithm 2: RemoveTreePos(s, T, TP, P), removes an entry to T with a positive label and a score s.

```
1 $v \leftarrow$ node with score $s$ in $T$;
2 update $p(v)$, $gp(v; P)$, and $accpos$ counters of the ancestors of $v$;
3 if $p(v) = 0$ then remove $v$ from the linked list $P$ and the search tree $TP$;
4 if $p(v) = n(v) = 0$ then remove $v$ from $T$;
```

Deleting an entry with a negative label and a score s is simpler. First, we find the node, say v, with the score s, and reduce the $n(v)$ counter by 1. If needed, we delete v from T. Finally we use MaxPos(s) to find u, the largest positive node with $s(u) \leq u$, and reduce $gn(u; P)$ by 1. The procedure, referred as RemoveTreeNeg, runs in $\mathcal{O}(\log k)$ time.

Next, we will describe the addition of a positive entry with a score s. First, we will add the entry s to T, possibly creating a new node in the process. Let v be the node in T with the score s.

If v is a new node, then we need to add it to the weighted linked list P. First, we find the node, say $w =$ MaxPos(s), after which v is supposed to be added. We need to compute the new gap counter $gn(v; P)$. By definition, this value is equal to the total count of negative labels of nodes between w and v, including w. Thus, this new gap counter is equal to $hn(w) - hn(v)$. Both counters can be obtained using HeadStats in $\mathcal{O}(\log k)$ time.

We will refer to this procedure as AddTreePos(s), and the pseudo-code is given in Algorithm 3. AddTreePos(s) runs in $\mathcal{O}(\log k)$ time.

Algorithm 3: AddTreePos(s), adds an entry to T with a positive label and a score s.

```
1 $w \leftarrow$ MaxPos($s$);
2 add $s$ to $T$ (possibly creating new node), and update $accpos$ and $p$ counters;
3 $v \leftarrow$ node with score $s$ in $T$;
4 if $w \neq v$ then
5     add $v$ to $TP$;
6     $p_1, n_1 \leftarrow$ HeadStats($s(w)$);
7     $p_2, n_2 \leftarrow$ HeadStats($s(v)$);
8     Add($P, w, v, 1, n_2 - n_1$) ;
9 return $v$;
```

Adding an entry with negative label and a score s is simpler. First, we will add the entry s to T, possibly creating a new node in the process. Let v be the node in T with a score s. Then, we use MaxPos(s) to find u, the largest positive node with $s(u) \leq u$, and increase $gn(u)$ by 1. The procedure, referred as AddTreeNeg, runs in $\mathcal{O}(\log k)$ time.

4 Estimating AUC efficiently

In order to approximate AUC, we will use Eq. 1 as a basis. However, instead of enumerating over every node we will enumerate only over some selected nodes. The key is how to select the nodes such that we will obtain the approximation guarantee while keeping the number of nodes small.

We will maintain a weighted linked list C. Given $\alpha > 1$, we say that C is α-compressed, if for every two consecutive nodes in C, say v and w, it holds that

$$hp(w) \leq \alpha(hp(v) + p(v)), \tag{3}$$

and if $u = \mathit{next}(w; C)$ exists, then

$$hp(u) > \alpha(hp(v) + p(v)) \quad . \tag{4}$$

Eq. 3 will yield the approximation guarantee, while the Eq. 4 will guarantee the running time.

4.1 Computing approximate AUC

Our next step is to show how we can approximate AUC using a compressed list L in $\mathcal{O}(L)$ time. The idea is as follows. Let B be the set of nodes between two consecutive nodes v and w in L. Normally, we would have to go over each individual node in B when computing AUC. Instead, we will group B to a *single* node. We will use the total number of positive labels in B, that is, $gp(v; L) - p(v)$, for the number of positive labels for this node. Similarly, we will use $gn(v; L) - n(v)$ for the negative labels. The pseudo-code for the algorithm is given in Algorithm 4.

Algorithm 4: ApproxAUC(L) computes approximate AUC using a weighted linked list.

```
1  $hp \leftarrow 0$; $a \leftarrow 0$;
2  while $v \in L$ do
3      $p \leftarrow p(v)$; $n \leftarrow n(v)$;
4      $a \leftarrow a + (hp + p/2)n$;
5      $hp \leftarrow hp + p$;
6      $p \leftarrow gp(v; L) - p(v)$; $n \leftarrow gn(v; L) - n(v)$;
7      $a \leftarrow a + (hp + p/2)n$;
8      $hp \leftarrow hp + p$;
9  $A \leftarrow$ (total number of positive labels) $\times$ (total number of negative labels);
10 return $a/A$;
```

Let us first establish that ApproxAUC produces an accurate estimate.

Proposition 1. *Let L be $(1 + \epsilon)$-compressed list constructed from the search tree T. Let $\widetilde{auc}$= ApproxAUC(L) be an approximate AUC, and let auc be the correct AUC. Then $|\widetilde{auc} - auc| \leq \epsilon auc/2$.*

Proof. Let A be as defined in ApproxAUC. Let $v \in T$ be a node, and let u be the node in L with the largest score such that $s(u) < s(v)$. Let $w = \mathit{next}(u; L)$ be the next node. Define

$$c_v = \frac{1}{2}(hp(u) + p(u) + hp(w)) \quad .$$

Then, ApproxAUC returns

$$\widetilde{auc} = \frac{1}{A} \sum_{v \in L} (hp(v) + \frac{1}{2}p(v))n(v) + \sum_{v \in T \setminus L} c_v n(v) \quad . \tag{5}$$

We will argue the approximation guarantee by comparing the terms in Eq. 1 and Eq. 5. Let v be a node in L. Then the corresponding term can be found in sums of both equations.

Let $v \in T \setminus L$, and write $b = hp(v) + \frac{1}{2}p(v)$. Let u be the node in L with the largest score such that $s(u) \le s(v)$. Let $w = \mathit{next}(u; L)$ be the next node. By definition, we have $hp(u) + p(u) \le b \le hp(w)$. Since c_v is the average of the lower bound and the upper bound, we have

$$|b - c_v| \le \frac{1}{2}(hp(w) - hp(u) - p(u)) \le \frac{\epsilon}{2}(hp(u) + p(u)) \le \frac{\epsilon b}{2},$$

where the second inequality follows since L is $(1+\epsilon)$-compressed.

We have shown that the approximation holds for individual terms. Consequently, it holds for the summands $\widetilde{auc}$ and auc, completing the proof. □

Two remarks are in order. First, since AUC is always smaller than 1, Proposition 1 implies that the approximation is also absolute, $|\widetilde{auc} - auc| \le \epsilon/2$. The relative approximation is more accurate if AUC is small. However, if AUC is close to 1, it may make sense to reverse the approximation guarantee, that is, modify the algorithm such that we have a guarantee of $|\widetilde{auc} - auc| \le (1 - auc)\epsilon/2$. This can be done by flipping the labels, and using $1 -$ ApproxAUC(C) as the estimate.

ApproxAUC runs in $\mathcal{O}(|L|)$ time. Next we establish that $|L|$ is small.

Proposition 2. *Let L be $(1+\epsilon)$-compressed list. Then $|L| \in \mathcal{O}\left(\frac{\log k}{\epsilon}\right)$, where k is the number of entries in the sliding window.*

Proof. Write $L = u_0, \ldots, u_m$. Since L is $(1+\epsilon)$-compressed, $hp(u_2) \ge 1$ and $hp(u_{i+2}) > (1+\epsilon)hp(u_i)$. Since $hp(u_m) \le k$, we have $(1+\epsilon)^{\lfloor m/2 \rfloor - 1} \le k$. Solving for m leads to $m \in \mathcal{O}\left(\frac{\log k}{\log 1+\epsilon}\right) \subseteq \mathcal{O}\left(\frac{\log k}{\epsilon}\right)$. □

4.2 Updating the data structures

Our final step is to describe procedures for maintaining C as the data window slides. In the previous section, we already described how to update the search trees T and TP as well as the weighed linked list P. Our next step is to make sure that the weighted linked list C stays α-compressed.

We will need two utility routines. The first routine, AddNext, given in Algorithm 5, takes as input a node included in both P and C, and adds to C the next node in P. This procedure will be used extensively to add extra nodes to C so that Eq. 3 is satisfied.

Algorithm 5: AddNext(v, L, P), adds the following node of v in P to L. Here P is the weighted linked list of all positive labels, and v is a node in P and L.

```
1 $w \leftarrow next(v, P)$;
2 $p \leftarrow gp(v, P)$; $n \leftarrow gn(v, P)$;
3 if $w \notin L$ then Add$(L, v, w, p, n)$;
```

Next, we demonstrate how AddNext enforces Eq. 3.

Lemma 1. *Assume that a linked list L satisfies Eq. 3 for consecutive positive nodes v and w. Add or remove a single positive entry with a score s, and assume that v and w are still positive. Let u be the next positive node from v in P, and let L' be the list obtained from L by adding a* positive *node u. Then Eq. 3 holds for L' for the nodes v and u as well as for the nodes u and w.*

Proof. Let us write $c_x = hp(x)$ before modifyng T, and $c'_x = hp(x)$ after the modification. Similarly, write $b_x = p(x)$ before the modification, and $b'_x = p(x)$ after the modification.

Since u is the next positive node of v, we have $c'_u = c'_v + b'_v \leq \alpha(c'_v + b'_v)$, proving the case of v and u.

If $s \geq s(w)$, then $c'_w = c_w \leq \alpha(c_v + b_v) = \alpha c_u = \alpha c'_u \leq \alpha(c'_u + b'_u)$.

If we are adding s and $s < s(w)$, then

$$c'_w = c_w + 1 \leq \alpha(c_v + b_v + 1) \leq \alpha(c'_v + b'_v + 1) = \alpha(c'_u + 1) \leq \alpha(c'_u + b'_u),$$

where the last inequality holds since u is a positive node.

If we are removing s and $s < s(w)$, then $c_v + b_v - 1 \leq c'_v + b'_v$, and so

$$c'_w \leq c_w \leq \alpha(c_v + b_v) \leq \alpha(c'_v + b'_v + 1) = \alpha(c'_u + 1) \leq \alpha(c'_u + b'_u).$$

This proves the case for u and w, and completes the proof. □

Note that the execution of AddNext is done in constant time, the key step for this being able to obtain $gp(v, P) = p(v)$ and $gn(v, P)$ in constant time. This is the main reason why we maintain P.

While the first utility algorithm adds new entries to C, our second utility algorithm, Compress, given in Algorithm 6 tries to delete as many entries as possible. It assumes that the input list C already satisfies Eq. 3, and searches for violations of Eq. 4. Whenever such violation is found, the algorithm proceeds deleting the middle node. Note that deleting this node will not violate Eq. 3. Consequently, upon termination, the resulted linked list will be α-compressed. The computational complexity of Compress(C, α) is $\mathcal{O}(|C|)$.

Algorithm 6: Compress(L, α), forces a weighted linked list L that satisfies Equation 3 to also satisfy Equation 4, making L α-compressed.

```
1 $v \leftarrow$ first element in $L$;
2 $c \leftarrow 0$;
3 while $next(next(v; L)\,; L)$ exists do
4     $w \leftarrow next(v; L)$;
5     if $c + gp(v; L) + gp(w; L) \leq \alpha(c + p(v))$ then
6         delete $w$ from $L$;
7     else
8         $c \leftarrow c + gp(v; L)$;
9         $v \leftarrow w$;
```

Next, we describe the update steps. We will start with the easier ones:

Adding negative entry: Given a negative entry with a score s, we first invoke AddTreeNeg. Then we search $u \in C$ with the largest score such that $s(u) \leq s$. Once this entry is found, we increase $gn(u; C)$ by 1.

Removing negative entry: Given a negative entry with a score s, we first invoke RemoveTreeNeg. Then we search $u \in C$ with the largest score such that $s(u) \leq s$. Once this entry is found, we decrease $gn(u; C)$ by 1.

Since the positive labels are not modified, C remains α-compressed, so there is no need for modifying C. The running time for both routines is $\mathcal{O}\left(\log k + \frac{\log k}{\epsilon}\right)$.

Let us now consider more complex cases:

Adding positive entry: Given a positive entry with a score s, we first invoke AddTreePos. Then we search $u \in C$ with the largest score such that $s(u) \leq s$. Once this entry is found, we increase $gp(u; C)$ by 1. By doing so, we may have violated Eq. 3 for u. Lemma 1 states that we can correct the problem by adding the next positive node for each violation. However, a closer inspection of the proof shows that there can be only one violation, namely u. Consequently, we check if Eq. 3 holds for u, and if it fails, we add the next positive node by invoking AddNext(u, C, P). Finally, we call Compress(C, α) to force Eq. 4; ensuring that C is α-compressed. The pseudo-code for AddPos is given in Algorithm 7.

Removing positive entry: Assume that we are given a positive entry with a score s. First we search $u \in C$ with the largest score such that $s(u) \leq s$. Once this entry is found, we decrease $gp(u; C)$ by 1. If u is no longer positive, we add the next

Algorithm 7: AddPos($s, \alpha; T, TP, P, C$), adds an entry with a positive label and a score s, updates the tree structures T and TP and the weighted linked lists P and C.

```
1 $v \leftarrow$ AddTreePos($s, T, TP, P$);
2 $u \leftarrow \arg\max \{s(w) \mid w \in C, s(w) \leq s\}$ ;
3 $gp(u; C) \leftarrow gp(u; C) + 1$;
4 $c \leftarrow \sum_{w \in C \mid s(w) < s(u)} gp(w; C)$;                    $\{c = hp(u)\}$
5 if $c + gp(u; C) > \alpha(c + p(v))$ then AddNext($u, C, P$) ;
6 Compress($C, \alpha$);
```

positive entry to C and delete u from C. The reason for this is explained later. We proceed by deleting the entry from the search trees with RemoveTreePos.

Next we make sure that Eq. 3 holds for every consecutive nodes v and w. There are two possible cases: (*i*) v and w were consecutive nodes in C before the deletion, or (*ii*) u was deleted from C, and w was the next positive node before the deletion. In the first case, Lemma 1 guarantees that using AddNext forces Eq. 3. In the second case, note that $hp(w)$ *after* the deletion is equal to $hp(u)$ *before* the deletion of u. This implies that since Eq. 3 held for v and u before the deletion, Eq. 3 holds for v and w after the deletion. Finally, we enforce Eq. 4 with Compress. The pseudo-code for RemovePos is given in Algorithm 8.

Algorithm 8: RemovePos($s, \alpha; T, TP, P, C$), removes an entry with a positive label and a score s, updates the tree structures T and TP and the weighted linked lists P and C.

```
1 $u \leftarrow \arg\max \{s(w) \mid w \in C, s(w) \leq s\}$;
2 $gp(u) \leftarrow gp(u) - 1$;
3 if $u \in C$ and $p(u) = 1$ then
4     AddNext($u, C, P$);
5     Remove($C, u$);
6 RemoveTreePos($s, T, TP, P$);
7 $v \leftarrow$ first element in $C$;
8 $c \leftarrow 0$;
9 while $next(v; C)$ exists do
10    $w \leftarrow next(v; C)$;
11    $x \leftarrow gp(v; C)$;
12    if $c + x > \alpha(c + p(v))$ then AddNext($v, C, P$);
13    $c \leftarrow c + x$;
14    $v \leftarrow w$;
15 Compress($C, \alpha$);
```

In both routines, modifying the search trees is done in $\mathcal{O}(\log k)$ time, while modifying C is done in $\mathcal{O}(|C|) \subseteq \mathcal{O}\left(\frac{\log k}{\epsilon}\right)$ time.

5 Related work

The closest related work is a study by Bouckaert [3], where the author divided the ROC curve area into bins, allowing only to maintain the counters for individual bins. However, the number of the bins as well as the bins were static, and no direct approximation guarantees were provided.

Using AUC in a streaming setting was proposed in a paper by Brzezinski and Stefanowski [5]. Here the authors use red-black tree, similar to T, to maintain the order of the data points in a sliding window, but they recompute the AUC from scratch every time, leading to a update time of $\mathcal{O}(k + \log k)$. In fact, our approach is essentially equivalent to their approach if we set $\epsilon = 0$.

Note that using AUC is useful if we do not have a threshold to binarize the score. If we do have such a threshold, then we can easily maintain a confusion matrix, which allows us to compute many metrics, such as, accuracy, recall, $F1$-measure [8, 9], and Kappa-statistic [2, 13]. However, determining such a threshold may be extremely difficult since it depends on the misclassification costs. Selecting such costs may come down to a(n educated) guess.

We based our AUC calculation on a sliding window, that is, we abruptly forget the data points after certain period of time. The other option is to gradually forget the data points, for example using an exponential decay (see a survey by Gama et al. [10] for such examples). There are currently no methodology for efficiently estimating AUC under exponential decay, and this is a promising future line of work.

In a related line of work, training a classifier by optimizing AUC in a static setting has been proposed by Ataman et al. [1], Brefeld and Scheffer [4], Ferri et al. [7], Herschtal and Raskutti [12]. Here, AUC is used as an optimization criterion, and needs to be recomputed from scratch in $\mathcal{O}(|D| \log |D|)$ time. Naturally, this may be too expensive for large databases. Calders and Jaroszewicz [6] estimated AUC as a continuous function. This allowed to view AUC as a smooth function, and optimize the parameters of the underlying classifier efficiently using gradient descent techniques. While the underlying problem is the same as ours, that is, computing AUC from scratch is expensive, the maintenance procedures make problems orthogonal: in our settings we are required to do updates when a single data point leaves or enters to our window, whereas here AUC needs to be recomputed since the scores (and the order) for all existing data points have changed. However, it may be possible and fruitful to use similar tricks in order to speed-up the AUC calculation when optimizing classifiers. We leave this as a future line of work.

Hand [11] proposed a fascinating alternative for AUC. Namely, the author views AUC as the optimal classification loss averaged (with weights) over misclassification cost ratio. He then argues that AUC evaluates incoherently, namely the cost ratio weights depend on the ROC curve, and then he proposes a different coherent alternative. The computation of proposed metric, though more complex, shares some similarity with AUC, and it may be possible to use similar techniques as in this paper to approximate this measure efficiently in a stream.

6 Experimental evaluation

In this section we present our experimental evaluation. We have two goals: to demonstrate the relative error in practice as a function of the guaranteed error, and to demonstrate the trade-off between the computational cost and the error.

We implemented calculation of AUC using C++, and conducted the experiments using Macbook Air (1.6 GHz Intel Core i5 / 8 GB Memory).[2] As a classifier we used Python's scikit implementation of logistic regression. Computing AUC was done in a separate job from training the classifier as well as scoring new data points; the reported running times measure only the computation of AUC over the whole test data.

We used 3 UCI datasets[3] for our experiments, see Table 1: (*i*) *Hepmass*, a dataset containing features from simulated particle collisions, split in training and test datasets. We used the *Hepmass*-1000 variant. Due to the memory restrictions of Python, we only used a sample of 500 000 data points from training data. We used the whole test dataset. (*ii*) *Miniboone*: a data used to distinguish electron neutrinos from muon neutrinos. Since the original data has data points ordered by label, we permuted the dataset and split it to training and test data. (*iii*) *Tvads*: a data containing features for identifying commercials from TV news channels. We used BBC and CNN channels as training data, and the remaining channels as test data.

Dataset	size of training dataset	size of test dataset
Hepmass	500 000	3 500 000
Miniboone	30 064	100 000
Tvads	40 265	89 420

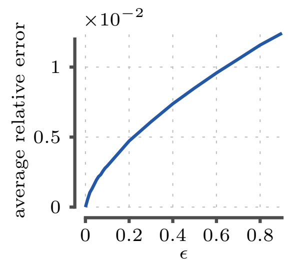
Table 1: Basic characteristics of the benchmark datasets.

Actual error vs. guarantee: Proposition 1 states that the error cannot be more than $\epsilon/2$. First, we test the actual relative error, that is, $|\widetilde{auc} - auc|/auc$ as a function of ϵ. Here we set the sliding window size to be 1000.

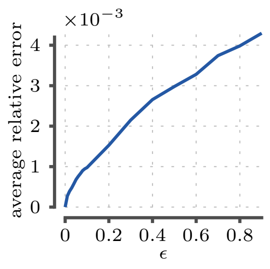
The top row of Figure 1 shows the relative error, averaged over all sliding windows, and the bottom row of Figure 1 shows the relative error, maximized over all sliding windows. From the results we see that both maximum and average error are smaller than the guaranteed. Especially, the average error is typically smaller of several orders than the theoretical guarantee. As expected, both errors tend to increase as ϵ increases.

Computational cost vs. error: Next, we test the trade-off between the computational cost and the relative error. The top row of Figure 2 shows the running time as a function of the average error, while the bottom row of Figure 2

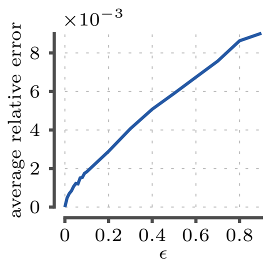
[2] See `https://bitbucket.org/orlyanalytics/streamauc` for the implementation.
[3] `https://archive.ics.uci.edu/`



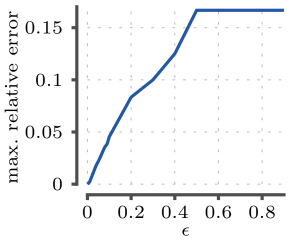
(a) *Hepmass*



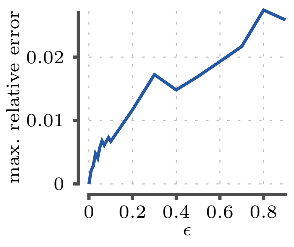
(b) *Tvads*



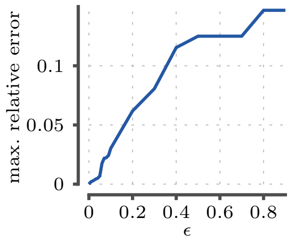
(c) *Miniboone*



(d) *Hepmass*



(e) *Tvads*



(f) *Miniboone*

Fig. 1: Actual relative error as a function of ϵ. Top row: average error, bottom row: maximum error. Proposition 1 states that error cannot be larger than $\epsilon/2$.

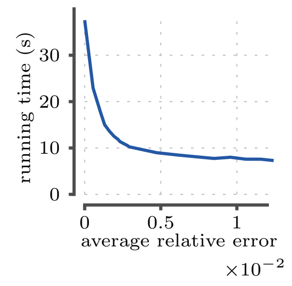
shows the size of $(1+\epsilon)$-compressed list as a function of the average error. Here, we used a window size of 1000.

From the results, we see the trade-off between the error and the running time: as the error increases, the running time drops. This is mainly due to the fewer elements in the compressed list as demonstrated in the bottom row. The running stabilizes for larger errors; this is due to the operations that do not depend on ϵ, such as maintaining binary tree T.

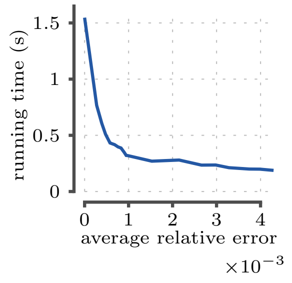
Computational cost vs. window size: Computing exact AUC requires $\mathcal{O}(k)$ time while estimating AUC is $\mathcal{O}(\log k/\epsilon)$. Consequently, the speed-up should increase as the size of the sliding window increases. We demonstrate this effect in Figure 3 using the *Miniboone* dataset. We see that the speed-up increases as a function of window size: computing estimates using $\epsilon = 0.1$ is 17 times faster for a window size of 10 000.

7 Concluding remarks

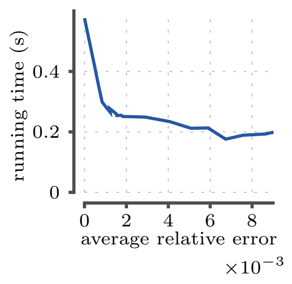
In this paper we introduced an approximation scheme that allows to maintain an estimate AUC in a sliding window within the guaranteed relative error of $\epsilon/2$ in $\mathcal{O}((\log k)/\epsilon)$ time. The key idea behind the estimator is to group the data points. The grouping has to be done cleverly so that the error stays small, the number of groups stay small, and the list can be updated quickly. We achieve this by maintaining groups, where the number of positive labels can only increase



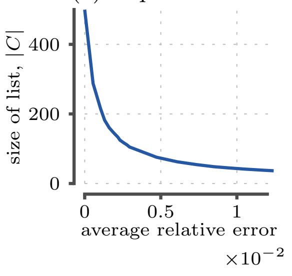
(a) *Hepmass*



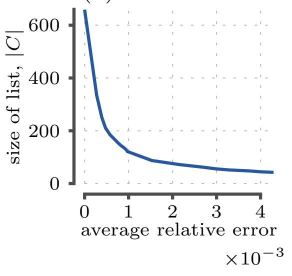
(b) *Tvads*



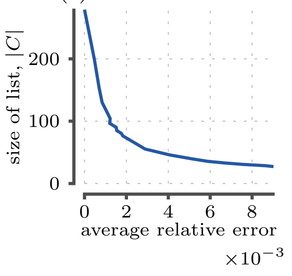
(c) *Miniboone*



(d) *Hepmass*



(e) *Tvads*



(f) *Miniboone*

Fig. 2: Top row: running time as a function of average relative error. Bottom row: size of the compressed list $|C|$ as a function of average relative error.

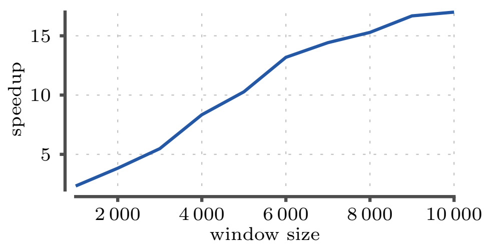


Fig. 3: A speed-up of estimating AUC with $\epsilon = 0.1$ against computing AUC exactly, as a function of sliding window size. The dataset is *Miniboone*.

relatively by $(1 + \epsilon)$ within one group, and must increase by at least $(1 + \epsilon)$ within two groups. Our experimental evaluation suggests that the average error in practice is much smaller than the guaranteed approximation, and that we can achieve significant speed-up, especially as the window size grows.

Our algorithm relies on the fact that the data points have no weights, specifically, Lemma 1 relies on the fact that the update may change the counters only by 1. If the data points are weighted, a different approach is required: It is possible to construct $(1+\epsilon)$-list from a scratch. The key idea here is a new query, where, given a threshold σ, we look for a node v that has the largest $hp(v)$ such that $hp(v) \leq \sigma$. This query can be done using the same trick as in HeadStats, and it requires $\mathcal{O}(\log k)$ time. The list can be then constructed by calling this query with exponentially increasing thresholds $\mathcal{O}((\log k)/\epsilon)$ times. This leads to a running time of $\mathcal{O}\big((\log^2 k)/\epsilon\big)$. An interesting direction for future work is to improve this complexity to, say, $\mathcal{O}((\log k)/\epsilon)$.

References

1. Ataman, K., Streetr, W., Zhang, Y.: Learning to rank by maximizing auc with linear programming. In: Neural Networks, 2006. IJCNN'06. International Joint Conference on. pp. 123–129. IEEE (2006)
2. Bifet, A., Frank, E.: Sentiment knowledge discovery in twitter streaming data. In: Discovery Science. pp. 1–15. Springer (2010)
3. Bouckaert, R.R.: Efficient AUC learning curve calculation. In: Australasian Joint Conference on Artificial Intelligence. pp. 181–191 (2006)
4. Brefeld, U., Scheffer, T.: Auc maximizing support vector learning. In: Proceedings of the ICML 2005 workshop on ROC Analysis in Machine Learning (2005)
5. Brzezinski, D., Stefanowski, J.: Prequential AUC: properties of the area under the ROC curve for data streams with concept drift. KAIS 52(2), 531–562 (2017)
6. Calders, T., Jaroszewicz, S.: Efficient AUC optimization for classification. In: PKDD. pp. 42–53 (2007)
7. Ferri, C., Flach, P., Hernández-Orallo, J.: Learning decision trees using the area under the roc curve. In: ICML. vol. 2, pp. 139–146 (2002)
8. Gama, J.: Knowledge discovery from data streams. CRC Press (2010)
9. Gama, J., Sebastião, R., Rodrigues, P.P.: On evaluating stream learning algorithms. Machine learning 90(3), 317–346 (2013)
10. Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., Bouchachia, A.: A survey on concept drift adaptation. ACM computing surveys 46(4), 44 (2014)
11. Hand, D.J.: Measuring classifier performance: a coherent alternative to the area under the ROC curve. Machine Learning 77(1), 103–123 (2009)
12. Herschtal, A., Raskutti, B.: Optimising area under the roc curve using gradient descent. In: Proceedings of the twenty-first international conference on Machine learning. p. 49. ACM (2004)
13. Žliobaitė, I., Bifet, A., Read, J., Pfahringer, B., Holmes, G.: Evaluation methods and decision theory for classification of streaming data with temporal dependence. Machine Learning 98(3), 455–482 (2015)