Machine Learning (2022) 111:2839–2862
https://doi.org/10.1007/s10994-021-06084-6

Maintaining AUC and *H*-measure over time

Nikolaj Tatti[1]

Received: 12 May 2020 / Revised: 18 August 2021 / Accepted: 13 September 2021 /
Published online: 14 December 2021
© The Author(s) 2021

Abstract
Measuring the performance of a classifier is a vital task in machine learning. The running time of an algorithm that computes the measure plays a very small role in an offline setting, for example, when the classifier is being developed by a researcher. However, the running time becomes more crucial if our goal is to monitor the performance of a classifier over time. In this paper we study three algorithms for maintaining two measures. The first algorithm maintains area under the ROC curve (AUC) under addition and deletion of data points in $\mathcal{O}(\log n)$ time. This is done by maintaining the data points sorted in a self-balanced search tree. In addition, we augment the search tree that allows us to query the ROC coordinates of a data point in $\mathcal{O}(\log n)$ time. In doing so we are able to maintain AUC in $\mathcal{O}(\log n)$ time. Our next two algorithms involve in maintaining H-measure, an alternative measure based on the ROC curve. Computing the measure is a two-step process: first we need to compute a convex hull of the ROC curve, followed by a sum over the convex hull. We demonstrate that we can maintain the convex hull using a minor modification of the classic convex hull maintenance algorithm. We then show that under certain conditions, we can compute the H-measure exactly in $\mathcal{O}\left(\log^2 n\right)$ time, and if the conditions are not met, then we can estimate the H-measure in $\mathcal{O}\left((\log n + \epsilon^{-1}) \log n\right)$ time. We show empirically that our methods are significantly faster than the baselines.

Keywords AUC · H-measure · Online algorithm

1 Introduction

Measuring the performance of a classifier is a vital task in machine learning. The running time of an algorithm that computes the measure plays a very small role in an offline setting, for example, when the classifier is being developed by a researcher. However, the running time becomes more crucial if our goal is to monitor the performance of a classifier over time where the new data points may arrive at a significant speed.

Editor: Eyke Hüllermeier.

✉ Nikolaj Tatti
nikolaj.tatti@helsinki.fi

[1] HIIT, University of Helsinki, Helsinki, Finland

For example, consider a task of monitoring abnormal behaviour in IT systems based on event logs. Here, the main problem is the gargantuan volume of event logs making the manual monitoring impossible. One approach is to have a classifier to monitor for abnormal events and alert analysts for closer inspection. Here, monitoring should be done continuously to notice abnormalities rapidly. Moreover, the performance of the classifier should also be monitored continuously as the underlying distribution, and potentially the performance of the classifier, may change due to the changes in the IT system.

In order to detect recent changes in the performance, we are often interested in the performance over the last n data points. More generally, we are interested in maintaining the measure under addition or deletion of data points.

We study algorithms for maintaining two measures. The first measure is the area under the ROC curve (AUC), a classic technique of measuring the performance of a classifier based on its ROC curve. We also study H-measure, an alternative measure proposed by Hand (2009). Roughly speaking, the measure is based on the minimum weighted loss, averaged over the cost ratio. A practical advantage of the H-measure over AUC is that it allows a natural way of weighting classification errors.

Both measures can be computed in $\mathcal{O}(n \log n)$ time from scratch, or in $\mathcal{O}(n)$ time if the data points are already sorted. In this paper we present 3 algorithms that allow us to maintain the measures in polylogarithmic time.

The first algorithm maintains AUC under addition or deletion of data points. The approach is straightforward: we maintain the data points sorted in a self-balanced search tree. In order to update AUC we need to know the ROC coordinates of the data point that we are changing. Luckily, this can be done by modifying the search tree so that it maintains the cumulative counts of the labels in each subtree. Consequently, we can obtain the coordinates in $\mathcal{O}(\log n)$ time, which leads to a total of $\mathcal{O}(\log n)$ maintenance time.

Our next two algorithms involve maintaining the H-measure. Computing the H-measure involves finding the convex hull of the ROC curve, and enumerating over the hull. First we show that we can use a classic dynamic convex hull algorithm with some minor modifications to maintain the convex hull of the ROC curve. The modifications are required as we do not have the ROC coordinates of individual data points, but we can use the same trick as when computing AUC to obtain the needed coordinates.

Then we show that if we estimate the class priors from the test data, we can decompose the H-measure into a sum over the points in the convex hull such that the ith term depends only on the difference between the ith and the $(i-1)$st data points. This decomposition allows us to maintain the H-measure in $\mathcal{O}\left(\log^2 n\right)$ time.

If the class priors are *not* estimated from the test data, then we propose an estimation algorithm. Here the idea is to group points that are close in the convex hull together. Or in other words, if there are points in the convex hull that are close to each other, then we only use one data point from such group. The grouping is done in a way that we maintain ϵ-approximation in $\mathcal{O}\left((\log n + \epsilon^{-1}) \log n\right)$ time.

Structure The rest of the paper is organized as follows. We present preliminary definitions in Sect. 2. In Sect. 4 we demonstrate how to maintain AUC, and in Sects. 5 and 6 we demonstrate how to maintain the H-measure. We present the experimental evaluation in Sect. 7, and conclude the paper with a discussion in Sect. 8.

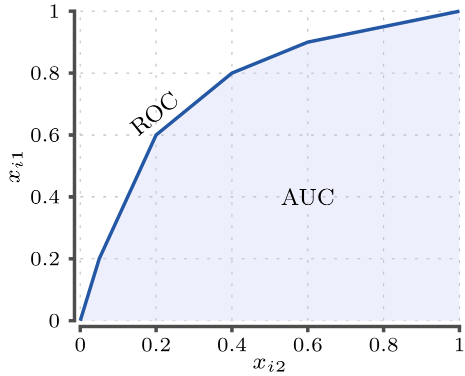


Fig. 1 Example of a ROC curve and AUC. If we consider label 1 as a true label and label 2 as a false label, then the vertical axis is the true positive rate (TPR) while the horizonal axis is the false positive rate (FPR)

2 Preliminaries

Assume that we are given a multiset of n data points Z. Each data point $z = (s, \ell)$ consists of a score $s \in R$ and a true label $\ell \in \{1, 2\}$. The score is typically obtained by applying a classifier with high values implying that z should be classified as class 2. To simplify the notation greatly, given $z = (s, \ell)$ we define $d(z) = (1, 0)$ if $\ell = 1$, and $d(z) = (0, 1)$ if $\ell = 2$. We can now write

$$(n_1, n_2) = \sum_{z \in Z} d(z),$$

that is, n_j is the number of points having the label equal to j. Here we used a convention that the sum of two tuples, say (a, b) and (c, d), is $(a + c, b + d)$. Note that $n = n_1 + n_2$.

Let $S = \left(s_1, \ldots, s_n\right)$ be the list of all scores, ordered from the smallest to the largest. Let us write

$$r_i = \sum_{z \in Z, s(z) \leq s_i} d(z), \tag{1}$$

that is, r_i are the label counts of points having a score less than or equal to s_i.

We obtain the *ROC curve* by normalizing r_i in Eq. 1, that is, the ROC curve is a list of $n + 1$ points $X = (x_0, x_1, \ldots, x_n)$, where

$$x_i = (r_{i1}/n_1, r_{i2}/n_2)$$

and $x_0 = (0, 0)$. Note that not all points in X are necessarily unique. The points in X are confined in the unit rectangle of $(0, 1) \times (0, 1)$. See Fig. 1 for illustration.[1]

The area under the curve, $auc(Z)$ is the area below the ROC curve. If there is a threshold σ such that all data points with a score smaller than σ belong to class 1 and all data points with a score larger than σ belong to class 2, then $auc(Z) = 1$. If the scores are independent of the true labels, then the expected value of $auc(Z)$ is 1/2.

[1] For notational convenience, we treat the first coordinate as the vertical and the second coordinate as the horizontal.

Instead of defining $auc(Z)$ using the ROC curve, we can also define it directly with Mann–Whitney U statistic (Mann and Whitney 1947). Assume that we are given a multiset of points Z. Let $S_1 = \{s \mid (s, \ell) \in Z, \ell = 1\}$ be a multiset of scores with the corresponding labels being equal to 1, and define S_2 similarly. The Mann–Whitney U statistic is equal to

$$U = \sum_{s \in S_1} \sum_{t \in S_2} f(s,t), \quad \text{where} \quad f(s,t) = \begin{cases} 1 & \text{if } s < t, \\ 0.5 & \text{if } s = t, \\ 0 & \text{if } s > t. \end{cases} \tag{2}$$

We obtain $auc(Z)$ by normalizing U, that is, $auc(Z) = \frac{1}{|S_1||S_2|}U$.

AUC can be computed naively using U statistic in $\mathcal{O}(n^2)$ time. However, we can easily speed up the computation to $\mathcal{O}(n \log n)$ time using Algorithm 1. To see the correctness, note that in Eq. 2 each $t \in S_2$ contributes to U with

$$\sum_{s \in S_1} f(s,t) = \left|\{s \in S_1 \mid s < t\}\right| + \frac{1}{2}\left|\{s \in S_1 \mid s = t\}\right|.$$

Algorithm 1 achieves its running time by maintaining the first term (in a variable h) as it loops over sorted scores. Note that if Z is already sorted, then the running time reduces to linear.

Algorithm 1: Algorithm for computing $auc(Z)$

```
1 S ← unique scores of Z, sorted;
2 (n1, n2) ← label counts;
3 U ← 0; h ← 0;
4 foreach s ∈ S do
5 |   (w1, w2) ← Σ_{s(z)=s} d(z);
6 |   U ← U + w2(h + w1/2);
7 |_  h ← h + w1;
8 return U/(n1 n2);
```

Our first goal is to show that we can maintain AUC in $\mathcal{O}(\log n)$ time under addition or removal of data points.

Our second contribution is a procedure for maintaining H-measure.

H-measure is an alternative method proposed by Hand (2009). The main idea is as follows: consider minimizing weighted loss,

$$\begin{aligned} Q(c, \sigma) &= cp(s(z) > \sigma, \ell(z) = 1) + (1-c)p(s(z) \le \sigma, \ell(z) = 2) \\ &= c\pi_1 p(s(z) > \sigma \mid \ell(z) = 1) + (1-c)\pi_2 p(s(z) \le \sigma \mid \ell(z) = 2), \end{aligned}$$

where c is a cost ratio, σ is a threshold, z is a random data point, and $\pi_k = p(\ell(z) = k)$ are class priors. Let us write $\sigma(c)$ to be the threshold minimizing $Q(c, \sigma)$ for a given c. Increasing c will decrease $\sigma(c)$, or in other words by varying c we will vary the threshold. As pointed out by Flach et al. (2011) the curve $Q(c, \sigma(c))$ is a variant of a *cost curve* [see Drummond and Holte (2006)],

$$cp(s(z) > \sigma \mid \ell(z) = 1) + (1-c)p(s(z) \le \sigma \mid \ell(z) = 2).$$

Here the difference is that $Q(c, \sigma(c))$ uses class priors π_k whereas the cost curve omits them.

Since not all values of c may be sensible, we assume that we are given a weight function $u(c)$. We are interested in measuring the weighted minimum loss as we vary c,

$$L = \int Q(c, \sigma(c)) u(c) dc. \tag{3}$$

Here small values of L indicate strong signal between the labels and the score.

The H-measure is a normalized version of L,

$$H = 1 - L/L_{max}.$$

Here, L_{max} is the largest possible value of L over all possible ROC curves. The negation is done so that the values of H are consistent with the AUC scores: values close to 1 represent good performance.

We will see that the convenient choice for u will be a beta distribution, as suggested by Hand (2009), since it allows us to express the integrals in a closed form.

Computing the empirical H-measure in practice starts with an ROC curve X. The following computations assume that the ROC curve is convex. If not, then the first step is to compute the convex hull of X, which we will denote by $Y = (y_0, \ldots, y_m)$. Taking a convex hull will inflate the performance of the underlying classifier, however it is possible to modify the underlying classifier [see Hand (2009) for more details] so that its ROC curve is convex.

We then define

$$c_i = \frac{\pi_2 (y_{i2} - y_{(i-1)2})}{\pi_2 (y_{i2} - y_{(i-1)2)}) + \pi_1 (y_{i1} - y_{(i-1)1})}, \tag{4}$$

where, recall that, $\pi_k = p(\ell(z) = k)$ are the class probabilities and $(y_0, \ldots, y_m)$ is the convex hull. The probabilities π_k can be either estimated from Z or by some other means. If former, then we show that we can maintain the H-measure exactly, if latter, then we need to estimate the measure in order to achieve a sublinear maintenance time.

We also set $c_0 = 0$ and $c_m = 1$. Note that c_i is a monotonically decreasing function of the slope of the convex hull. This guarantees that $c_i \leq c_{i+1}$. We can show that [see Hand (2009)] if $c_i < c < c_{i+1}$, then the minimum loss is equal to

$$Q(c, \sigma(c)) = c\pi_1(1 - y_{i1}) + (1 - c)\pi_2 y_{i2}.$$

We can now write Eq. 3 as

$$L = \sum_{i=0}^{m} \pi_1 (1 - y_{i1}) \int_{c_i}^{c_{i+1}} c u(c) dc + \pi_2 y_{i2} \int_{c_i}^{c_{i+1}} (1 - c) u(c) dc, \tag{5}$$

and if we use beta distribution with parameters (α, β) as $u(c)$, we have

$$\begin{aligned} L = \frac{1}{B(1, \alpha, \beta)} \sum_{i=0}^{m} & \pi_1 (1 - y_{i1}) \big(B(c_{i+1}; \alpha + 1, \beta) - B(c_i; \alpha + 1, \beta) \big) \\ & + \pi_2 y_{i2} \big(B(c_{i+1}; \alpha, \beta + 1) - B(c_i; \alpha, \beta + 1) \big), \end{aligned} \tag{6}$$

where $B(\cdot, \alpha, \beta)$ is an incomplete beta function.

Finally, we can show that the normalization constant is equal to

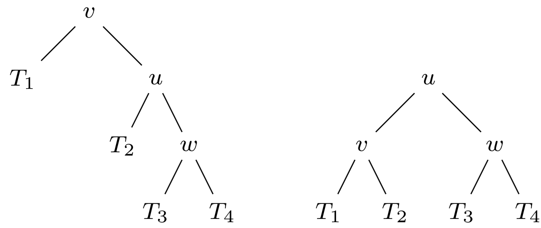


Fig. 2 An example of left rotation in a search tree. Left figure: before rotation, right figure: after rotation. Note that only u and v have different children after the rotation

$$L_{max} = \frac{\pi_1 B(\pi_1;\alpha+1,\beta) + \pi_2 B(1;\alpha,\beta+1) - \pi_2 B(1;\alpha,\beta+1)}{B(1,\alpha,\beta)}.$$

Given an ROC curve X, computing the convex hull Y, and subsequent steps, can be done in $\mathcal{O}(n)$ time. We will show in Sect. 5 that we can maintain the H-measure in $\mathcal{O}\left(\log^2 n\right)$ time if π_k are estimated from Z. Otherwise we will show in Sect. 6 that we can approximate the H-measure in $\mathcal{O}\left((\epsilon^{-1} + \log n)\log n\right)$ time.

As pointed earlier, $Q(c, \sigma(c))$ can be viewed as a variant of a cost curve. If we were to replace Q with the cost curve and use uniform distribution for u, then, as pointed by Flach et al. (2011), L is equivalent to the area under the cost curve. Interestingly enough, we cannot use the algorithm given in Sect. 5 to compute the area of under the cost curve as the precense of the priors is needed to decompose the measure. However, we can use the algorithm in Sect. 6 to estimate the area under the cost curve.

Interestingly enough, $Q(c, \sigma)$ can be linked to AUC. If, instead of using the optimal threshold $\sigma(c)$, we average Q over carefully selected distribution for σ and also use uniform distribution for c, then the resulting integral is a linear transformation of AUC (Flach et al. 2011).

Self-balancing search trees In this paper we make a significant use of self-balancing search trees such as AVL-trees of red-black trees. Such trees are binary trees where each node, say u, has a key, say k. The left subtree of u contains nodes with keys smaller than k and the right subtree of u contains nodes with keys larger than k. Maintaining this invariant allows for efficient queries as long as the height of the tree is kept in check. Self-balancing trees such as AVL-trees or red-black trees keep the height of the tree in $\mathcal{O}(\log n)$. The balancing is done with $\mathcal{O}(\log n)$ number of left rotations or right rotations whenever the tree is modified (see Fig. 2). Searching for nodes with specific keys, inserting new nodes, and deleting existing nodes can be done in $\mathcal{O}(\log n)$ time. Moreover, splitting the search tree into two search tree or combining two trees into one can also be done in $\mathcal{O}(\log n)$ time.

We assume that we can compare and manipulate integers of size $\mathcal{O}(n)$ and real numbers in constant time. We do this because it is reasonable to assume that the current bit-length of integers in modern computer acrhitecture is sufficient for any practical applications, and we do need to resort to any custom big integer implementations. If needed, however, the running times need to be multiplied by an additional $\mathcal{O}(\log n)$ factor.

3 Related work

Several works have studied maintaining AUC in a sliding window. Brzezinski and Stefanowski (2017) maintained the order of n data points using a red-black tree but computed AUC from scratch, resulting in a running time of $\mathcal{O}(n + \log n)$, per update. Tatti (2018) proposed algorithm yielding ϵ-approximation of AUC in $\mathcal{O}\left((1 + \epsilon^{-1}) \log n)\right)$ time, per

update. Here the approach bins the ROC space into a small number of bins. The bins are selected so that the AUC estimate is accurate enough. Bouckaert (2006) proposed estimating AUC by binning and only maintaining counters for individual bins. On the other hand, in this work we do not need to resort to binning, instead we can maintain the exact AUC by maintaining a search search tree structure in $\mathcal{O}(\log n)$ time, per update.

We should point out that AUC and the H-measure are defined over the whole ROC curve, and are useful when we do not want to commit to a specific classification threshold. On the other hand, if we do have the threshold, then we can easily maintain a confusion matrix, and consequently maintain many classic metrics, for example, accuracy, recall, $F1$-measure (Gama et al. 2013; Gama 2010), and Kappa-statistic (Bifet and Frank 2010; Žliobaitė et al. 2015).

In a related work, Ataman et al. (2006); Ferri et al. (2002); Brefeld and Scheffer (2005); Herschtal and Raskutti (2004) proposed methods where AUC is optimized as a part of training a classifier. Note that this setting differs from ours: changing the classifier parameters most likely will change the scores of *all* data points, and may change the data point order significantly. On the other hand, we rely on the fact we can maintain the order using a search tree. Interestingly, Calders and Jaroszewicz (2007) estimated AUC using a continuous function which then allowed optimizing the classifier parameters with gradient descent.

Our approaches are useful if we are working in a sliding window setting, that is, we want to compute the relevant statistic using only the last n data points. In other words, we abruptly forget the $(n + 1)$th data point. An alternative option would be to gradually downplay the importance of older data points. A convenient option is to use exponential decay, see for example a survey by Gama et al. (2014). While maintaining the confusion matrix is trivial when using exponential decay but—to our knowledge—there are no methods for maintaining AUC or H-measure under exponential decay.

4 Maintaining AUC

In this section we present a simple approach to maintain AUC in $\mathcal{O}(\log n)$ time. We accomplish this by showing that the *change* in AUC can be computed in $\mathcal{O}(\log n)$ time whenever a new point is added or an existing point is deleted. We rely on the following two propositions that express how AUC changes when adding or deleting a data point. We then show that the quantities occurring in the propositions, namely, the weights (u_1, u_2) and (v_1, v_2) can be obtained in $\mathcal{O}(\log n)$ time.

Proposition 1 (Addition) *Let Z be a set of data points with (n_1, n_2) label counts. Let Y be a set of points having the same score σ. Write $(w_1, w_2) = \sum_{y \in Y} d(y)$. Define also*

$$(u_1, u_2) = \sum_{\substack{z \in Z \\ s(z) < \sigma}} d(z) \quad \text{and} \quad (v_1, v_2) = \sum_{\substack{z \in Z \\ s(z) = \sigma}} d(z).$$

Write $U = n_1 n_2 \times auc(Z)$ and $U' = (n_1 + w_1)(n_2 + w_2) \times auc(Z \cup Y)$. Then

$$U' = U + w_2\left(u_1 + \frac{v_1}{2}\right) + w_1\left(n_2 - u_2 - \frac{v_2}{2}\right) + \frac{w_1 w_2}{2}.$$

Proof We will use Mann–Whitney U statistic, given in Eq. 2 to prove the claim. Let us write $Z' = Z \cup Y$ and define

$$S_i = \{s \mid (s, \ell) \in Z, \ell = i\} \quad \text{and} \quad S'_i = \left\{s \mid (s, \ell) \in Z', \ell = i\right\}, \quad \text{for} \quad i = 1, 2.$$

Equation 2 states that

$$\begin{aligned}
U' &= \sum_{s \in S'_1} \sum_{t \in S'_2} f(s, t) \\
&= w_1 \sum_{t \in S'_2} f(\sigma, t) + w_2 \sum_{s \in S_1} f(s, \sigma) + \sum_{s \in S_1} \sum_{t \in S_2} f(s, t) \\
&= w_1 \sum_{t \in S'_2} f(\sigma, t) + w_2 \sum_{s \in S_1} f(s, \sigma) + U \\
&= w_1 \left(n_2 - u_2 - v_2 + \frac{v_2 + w_2}{2} \right) + w_2 \left(u_1 + \frac{v_1}{2} \right) + U.
\end{aligned}$$

We obtain the claim by rearranging the terms. □

Proposition 2 (Deletion) *Let Z be a set of data points with (n_1, n_2) label counts. Let $Y \subseteq Z$ be a set of points having the same score σ. Write $(w_1, w_2) = \sum_{y \in Y} d(y)$. Define also*

$$(u_1, u_2) = \sum_{\substack{z \in Z \\ s(z) < \sigma}} d(z) \quad \text{and} \quad (v_1, v_2) = \sum_{\substack{z \in Z \\ s(z) = \sigma}} d(z).$$

Write $U = n_1 n_2 \times auc(Z)$ and $U' = (n_1 - w_1)(n_2 - w_2) \times auc(Z \setminus Y)$. Then

$$U' = U - w_2 \left(u_1 + \frac{v_1}{2} \right) - w_1 \left(n_2 - u_2 - \frac{v_2}{2} \right) + \frac{w_1 w_2}{2}.$$

Note that the sign of the last term is the same for both addition and deletion.

Proof We will use Mann–Whitney U statistic, given in Eq. 2 to prove the claim. Let us write $Z' = Z \setminus Y$ and define

$$S_i = \{s \mid (s, \ell) \in Z, \ell = i\} \quad \text{and} \quad S'_i = \left\{s \mid (s, \ell) \in Z', \ell = i\right\}, \quad \text{for} \quad i = 1, 2.$$

Equation 2 states that

$$\begin{aligned}
U &= \sum_{s \in S_1} \sum_{t \in S_2} f(s, t) \\
&= w_1 \sum_{t \in S_2} f(\sigma, t) + w_2 \sum_{s \in S'_1} f(s, \sigma) + \sum_{s \in S'_1} \sum_{t \in S'_2} f(s, t) \\
&= w_1 \sum_{t \in S_2} f(\sigma, t) + w_2 \sum_{s \in S'_1} f(s, \sigma) + U' \\
&= w_1 \left(n_2 - u_2 - v_2 + \frac{v_2}{2} \right) + w_2 \left(u_1 + \frac{v_1 - w_1}{2} \right) + U'.
\end{aligned}$$

We obtain the claim by rearranging the terms. □

Note that normally we would be adding or deleting a single data point, that is, $Y = \{y\}$. However, the propositions also allow us to modify multiple points with the same score.

These two propositions allow us to maintain AUC as long as we can compute (u_1, u_2) and (v_1, v_2). To compute these quantities we will use a balanced search tree T such as red-black tree or AVL tree. Let S be the unique scores of Z. Each score $s \in S$ is given a node $n \in T$.

Moreover, for each node x with a score of s, we will store the total label counts having the same score, $d(x) = \sum_{s(z)=s} d(z)$. The counts $d(x)$ will give us immediately (v_1, v_2).

In addition, we will store $cd(x)$, cumulative label counts of all descendants of x, including x itself. We need to maintain these counts whenever we add or remove nodes from T, change the counts of nodes, or when T needs to be rebalanced. Luckily, since

$$cd(x) = cd(left(x)) + cd(right(x)) + d(x)$$

we can compute $cd(x)$ in constant time as long as we have the cumulative counts of children of x. Whenever node x is changed, only its ancestors are changed, so the cumulative weights can be updated in $\mathcal{O}(\log n)$ time. The balancing in red-black tree or AVL tree is done by using left or right rotation. Only two nodes are changed per rotation (see Fig. 2), and we can recompute the cumulative counts for these nodes in constant time. There are at most $\mathcal{O}(\log n)$ rotations, so the running time is not increased.

Given a tree T and a score threshold σ, let us define $lcount(\sigma, T) = \sum_{s(z)x<\sigma} d(x)$, to be the total count of nodes with scores smaller than σ. Computing $lcount(s, T)$ gives us (u_1, u_2) used by Propositions 1–2.

In order to compute $lcount(\sigma, T)$ we will use the procedure given in Algorithm 2. Here, we use a binary search over the tree, and summing the cumulative counts of the left branch. To see the correctness of the algorithm, observe that during the while-loop Algorithm 2 maintains the invariant that $u + cd(left(x))$ is equal to $lcount(s(x), T)$. We should point out that similar queries were considered by Tatti (2018). However, they were not combined with Propositions 1–2.

Algorithm 2: Computes $lcount(\sigma, T)$ using a binary search tree

```
1 x ← root of T;
2 u ← (0, 0);
3 while x or s(x) ≠ σ do
4     if s(x) > σ then
5         x ← left(n);
6     else
7         u ← u + cd(left(x)) + d(x);
8         x ← right(x);
9 return u + cd(left(x));
```

Since T is balanced, the running time of Algorithm 2 is $\mathcal{O}(\log n)$.

In summary, we can maintain T in $\mathcal{O}(\log n)$ time, and we can obtain (u_1, u_2) and (v_1, v_2) using T in $\mathcal{O}(\log n)$ time. These quantities allow us to maintain AUC in $\mathcal{O}(\log n)$ time.

5 Maintaining H-measure

If we were to compute the H-measure from scratch, we first need to compute the convex hull, and then compute the H-measure from the convex hull. In order to maintain the H-measure, we will first address maintaining the convex hull, and then explain how we maintain the actual measure.

5.1 Divide-and-conquer approach for maintaining a convex hull

Maintaining a convex hull under point additions or deletions is a well-studied topic in computational geometry. A classic approach by Overmars and Van Leeuwen (1981) maintains the hull in $\mathcal{O}\left(\log^2 n\right)$ time. Luckily, the same approach with some modifications will work for us.

Before we continue, we should stress two important differences between our setting and a traditional setting of maintaining a convex hull.

First, in a normal setting, the additions and removals are done to new *points in a plane*. In other words, the remaining points do not change over time. In our case, the data point consists of a classifier score and a label, and modifications shift the ROC coordinates of every point. As a concrete example, in a traditional setting, adding a point cannot reveal already existing points whereas adding a new data point can shift the ROC curve enough so that some existing points become included in the convex hull.

Secondly, we do not have the coordinates for all the points. However, it turns out that we can compute the *needed* coordinates with no additional costs.

We should point out that the approach by Overmars and Van Leeuwen (1981) is not the fastest for maintaining the hull: for example an algorithm by Brodal and Jacob (2002) can maintain the hull in $\mathcal{O}(\log n)$ time. However, due to the aforementioned differences adapting this algorithm to our setting is non-trivial, and possibly infeasible.

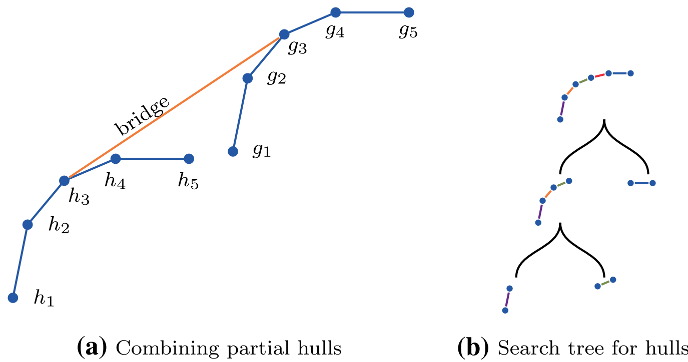
We will explain next the main idea behind the algorithm by Overmars and Van Leeuwen (1981), and then modify it to our needs.

The overall idea behind the algorithm is as follows. A generic convex hull can be viewed as a union of the lower convex hull and the upper convex hull. We only need to compute the upper convex hull, and for simplicity, we will refer to the upper convex hull as the convex hull.

In order to compute the convex hull C for a point set P we can use a conquer-and-divide technique. Assume that we have ordered the points using the x-coordinate, and split the points roughly in half, say in sets R and Q. Then assume we have computed convex hulls, say $H = \left\{h_i\right\}$ and $G = \left\{g_i\right\}$, for R and Q independently.

A key result by Overmars and Van Leeuwen (1981) states that the convex hull C of P is equal to $\left\{h_1, \ldots, h_u, g_v, g_{v+1}, \ldots\right\}$, that is, C starts with H and ends with G. See Fig. 3 for illustration. The segment between h_u and g_v is often referred as a *bridge*.

We can find the indices u and v in $\mathcal{O}(\log n)$ time using a binary search over H and G. In order to perform the binary search we will store the hulls H and G in balanced search trees (red-black tree or AVL tree). Then the binary search amounts to traversing these trees.



(a) Combining partial hulls **(b)** Search tree for hulls

Fig. 3 Left figure: an example of combining two partial convex hulls into one by finding a bridge segment. Right figure: a stylized data structure for maintaining convex hull. Each node corresponds to a partial convex hull (that are stored in separate search trees), a parent hull is obtained from the child hulls by finding the bridge segment. Leaf nodes containing individual data points are not shown

Note that the concatenation and splitting of a search tree can be done in $\mathcal{O}(\log n)$ time. In other words, we can obtain C for partial convex hulls H and G in $\mathcal{O}(\log n)$ time.

In order to maintain the hull we will store the original points in a balanced search tree T;[2] only the leaves store the actual points. Each node in $u \in T$ represents a set of points stored in the descendant leaves of u. See Fig. 3a for illustration.

Let us write $H(u)$ to be the convex hull of these points: we can obtain $H(u)$ from $H(\mathit{left}(u))$ and $H(\mathit{right}(u))$ in $\mathcal{O}(\log n)$ time. So whenever we modify T by adding or removing a leaf v, we only need to update the ancestors of v, and possibly some additional nodes due to the rebalancing. All in all, we only need to update $\mathcal{O}(\log n)$ nodes, which brings the running time to $\mathcal{O}\left(\log^2 n\right)$.

An additional complication is that whenever we compute $H(u)$ we also destroy $H(\mathit{left}(u))$ and $H(\mathit{right}(u))$ in the process, trees that we may need in the future. However, we can rectify this by storing the remains of the partial hulls, and then reversing the join if we were to modify a leaf of u. This reversal can be done in $\mathcal{O}(\log n)$ time.

5.2 Maintaining the convex hull of a ROC curve

Our next step is to adapt the existing algorithm to our setting so that we can maintain the hull of an ROC curve X.

First of all, adding or removing data points shifts the remaining points. To partially rectify this issue, we will use non-normalized coordinates $R = \left(r_0, \ldots, r_m\right)$ given in Eq. 1. We can do this because scaling does not change the convex hull.

Consider adding or removing a data point z which is represented by a leaf $u \in T$. The points in R associated with smaller scores than $s(z)$ will not shift, and the points in R associated with larger scores than $s(z)$ will shift by the same amount. Consequently, the only partial hulls that are affected are the ancestors of u. This allows us to use the update algorithm of Overmars and Van Leeuwen (1981) for our setting as long as we can obtain the coordinates of the points.

[2] This is a different tree than the trees used for storing convex hulls.

Our second issue is that we do not have access to the coordinates r_i. We approach the problem with the same strategy as when we were computing AUC.

Let U be the search tree of a convex hull H. Let $u \in U$ be a node with coordinates r_i. We will define and store $d(u)$ as the coordinate difference $r_i - r_{i-1}$. Let s_i be the score corresponding to r_i. Then Eq. 1 implies that $d(u) = \sum_{s_{i-1} < s(z) \leq s_i} d(z)$.

In addition, we will store $cd(u)$, the total sum of the coordinate differences of descendants of u, including u itself.

Let u be the root of U. The coordinates, say p, of u in U are $cd(left(u)) + d(u)$. Moreover, the coordinates of the left child of u are

$$p - d(u) - cd(right(left(u))),$$

and the coordinates of the right child of u are

$$p + d(right(u)) + cd(left(right(u))).$$

In other words, we can compute the coordinates of children in U in constant time if we know the coordinates of a parent.

When combining two hulls, the binary search needed to find the bridge is based on descending U from root to the correct node. During the binary search the algorithm needs to know the coordinates of a node which we can now obtain from the coordinates of the parent. In summary, we can do the binary search in $\mathcal{O}(\log n)$ time, which allow us to maintain the hull of a ROC curve in $\mathcal{O}(\log^2 n)$ time.

For completeness we present the pseudo-code for the binary search in Appendix.

5.3 Maintaining *H*-measure

Now that we have means to maintain the convex hull, our next step is to maintain the H-measure. Note that the only non-trivial part is L given in Eq. 5.

Assume that we have n data points Z with n_k data points having class k. Let $Y = (y_0, \ldots, y_m)$ be the convex hull of the ROC curve computed from Z. Let $(d_1, \ldots, d_m)$ the non-normalized differences between the neighboring points, that is,

$$d_{i1} = n_1(y_{i1} - y_{(i-1)i}) \quad \text{and} \quad d_{i2} = n_2(y_{i2} - y_{(i-1)2}).$$

We will now assume that π_k occurring in Eq. 5 are computed from the same data as the ROC curve, that is, $\pi_k = n_k/n$. We can rewrite the first term in Eq. 5 as

$$\begin{aligned}
\sum_{i=0}^{m} \pi_1(1 - y_{i1}) \int_{c_i}^{c_{i+1}} cu(c)dc &= \frac{1}{n} \sum_{i=0}^{m} \sum_{j=i+1}^{m} d_{j1} \int_{c_i}^{c_{i+1}} cu(c)dc \\
&= \frac{1}{n} \sum_{j=1}^{m} d_{j1} \sum_{i=0}^{j-1} \int_{c_i}^{c_{i+1}} cu(c)dc \\
&= \frac{1}{n} \sum_{j=1}^{m} d_{j1} \int_{0}^{c_j} cu(c)dc.
\end{aligned}$$

Similarly, we can express the second term of Eq. 5 as

$$\sum_{i=0}^{m} \pi_2 y_{i2} \int_{c_i}^{c_{i+1}} (1-c)u(c)dc = \frac{1}{n} \sum_{i=0}^{m} \sum_{j=1}^{i} d_{j2} \int_{c_i}^{c_{i+1}} (1-c)u(c)dc$$
$$= \frac{1}{n} \sum_{j=1}^{m} d_{j2} \sum_{i=j}^{m} \int_{c_i}^{c_{i+1}} (1-c)u(c)dc$$
$$= \frac{1}{n} \sum_{j=1}^{m} d_{j2} \int_{c_j}^{1} (1-c)u(c)dc.$$

If we use the beta distribution for u, Eq. 6 reduces to

$$L = \frac{1}{nB(1,\alpha,\beta)} \sum_{j=1}^{m} d_{j1} B(c_j, \alpha+1, \beta) + d_{j2}(B(1,\alpha,\beta+1) - B(c_j,\alpha,\beta+1)). \tag{7}$$

Let us now consider values c_j. Because we assume that π_k are estimated from the testing data, we have $\pi_k = n_k / n$, so the values c_j, given in Eq. 4, reduce to

$$c_j = \frac{\pi_2 (y_{j2} - y_{(j-1)2})}{\pi_2 (y_{j2} - y_{(j-1)2})) + \pi_1 (y_{j1} - y_{(j-1)1})} = \frac{\pi_2 d_{j2} / n_2}{\pi_1 d_{j1} / n_1 + \pi_2 d_{j2} / n_2} = \frac{d_{j2}}{d_{j1} + d_{j2}}.$$

In summary, the terms of the sum in Eq. 7 depend *only* on the coordinate differences d_j. We should stress that this is only possible if we assume that π_k are computed from the same data as the ROC curve. Otherwise, the terms n_k will not cancel out when computing c_j.

Let T be a binary tree representing a convex hull. The sole dependency on d_j allows us to use T to maintain the H-measure. In order to do that, let $v \in T$ be a node with the coordinate difference $(d_1, d_2) = d(v)$. Let $c = d_2 / (d_1 + d_2)$. We define

$$h(v) = d_1 B(c, \alpha+1, \beta) + d_2(B(1,\alpha,\beta+1) - B(c,\alpha,\beta+1)).$$

We also maintain $ch(v)$ to be the sum of $h(u)$ of all descendants u of v, including v. Note that maintaining $ch(v)$ can be done in a similar fashion as $cd(v)$.

Finally, Eq. 7 implies that $L = \frac{ch(root(T))}{nB(1,\alpha,\beta)}$, allowing us to maintain the H-measure in $\mathcal{O}(\log^2 n)$ time.

6 Approximating *H*-measure

In our final contribution we consider the case where π_k are not computed from the same dataset as the ROC curve. The consequence is that we no longer can simplify c_j so that it only depends on d_j, and we cannot express L as a sum over the nodes of the tree representing the convex hull.

We will approach the task differently. We will still maintain the convex hull H. We then select a *subset* of points from H from which we compute the H-measure from scratch. This subset will be selected carefully. On one hand, the subset will yield an ϵ-approximation. On the other hand, the subset will be small enough so that we still obtain polylogarithmic running time.

We start by rewriting Eq. 5. Given a function $x : [0,1] \to \mathbb{R}^+$, let us define

$$L_1(x) = \int_0^1 \pi_1 x(c) c u(c) dc, \quad \text{and} \quad L_2(x) = \int_0^1 \pi_2 x(c)(1-c)u(c) dc.$$

Consider the values $\{y_i\}$ and $\{c_i\}$ as used in Eq. 5. We define two functions $f, g : [0,1] \rightarrow \mathbb{R}^+$ as

$$\begin{aligned} g(c) &= y_{i2}, \quad \text{where} \quad c_i \leq c < c_{i+1}, \quad \text{and} \quad g(1) = 1, \\ f(c) &= 1 - y_{i1}, \quad \text{where} \quad c_i \leq c < c_{i+1}, \quad \text{and} \quad f(1) = 0. \end{aligned} \tag{8}$$

We can now write Eq. 5 as $L = L_1(f) + L_2(g)$.

We say that a function x' is an ϵ-approximation of a function x if $|x(c) - x'(c)| \leq \epsilon x(c)$. The following two propositions are immediate.

Proposition 3 *Let x' be an ϵ-approximation of x, then*

$$\left|L_1(x) - L_1(x')\right| \leq \epsilon L_1(x) \quad \text{and} \quad \left|L_2(x) - L_2(x')\right| \leq \epsilon L_2(x).$$

Proposition 4 *Let f and g be defined as in Eq. 8, and let f' and g' be respective ϵ-approximations. Define*

$$H = 1 - \frac{L_1(f) + L_2(g)}{L_{max}} \quad \text{and} \quad H' = 1 - \frac{L_1(f') + L_2(g')}{L_{max}}.$$

Then $|H - H'| \leq \epsilon(1 - H)$.

In other words, if we can approximate f and g, we can also approximate the H-measure. Note that the guarantee is $\epsilon(1 - H)$, that is, the approximation is more accurate when H is closer to 1, that is, a classifier is accurate.

Next we will focus on estimating g.

Proposition 5 *Assume $\epsilon > 0$. Let Y be the convex hull of an ROC curve. Let Q be a subset of Y such that for each y_i, there is $q_j \in Q$ such that*

$$q_j = y_i \quad \text{or} \quad q_{j2} \leq y_{i2} \leq q_{(j+1)2} \leq (1+\epsilon) q_{j2}. \tag{9}$$

Let g be the function constructed from Y as given by Eq. 8, and let g' be a function constructed similarly from Q. Then g' is an ϵ-approximation of g.

Proof Let (c_i) be the slope values computed from Y using Eq. 4, and let (c'_i) be the slope values computed from Q.

Due to convexity of Y, the slope values have a specific property that we will use several times: fix index j, and let i be the index such that $y_i = q_j$. Then

$$c'_j \leq c_i \quad \text{and} \quad c'_{j+1} \geq c_{i+1}. \tag{10}$$

Assume $0 < c < 1$. Let i be an index such that $c_i \leq c < c_{i+1}$, consequently $g(c) = y_{i2}$. Similarly, let j be an index such that $c'_j \leq c < c'_{j+1}$, so that $g'(c) = q_{j2}$. Let a be an index such that $q_j = y_a$.

If $g(c) = g'(c)$, then we have nothing to prove. Assume $g(c) < g'(c) = q_{j2}$.

Assume $q_{j2} > (1+\epsilon)q_{(j-1)2}$. Then Eq. 9 implies that $y_{a-1} = q_{j-1}$, and so $c_a = c'_j \leq c < c_{i+1}$. Thus, $i \geq a$, and $g(c) = g(c_i) \geq g(c_a) = g'(c'_j) = g'(c)$, which is a contradiction.

Assume $q_{j2} \leq (1+\epsilon)q_{(j-1)2}$. Let b be an index such that $y_b = q_{j-1}$. Then Eq. 10 implies

$$c_{b+1} \leq c'_j \leq c < c_{i+1}.$$

Thus, $b < i$ and so $q_{(j-1)2} = y_{b2} = g(c_b) \leq g(c_i) = g(c)$. This leads to

$$\left|g'(c) - g(c)\right| = q_{j2} - g(c) \leq (1+\epsilon)q_{(j-1)2} - g(c) \leq (1+\epsilon)g(c) - g(c) = \epsilon g(c),$$

proving the proposition.

Now, assume $g(c) > g'(c) = q_{j2}$.

Assume $q_{(j+1)2} > (1+\epsilon)q_{j2}$. If $y_{a+1} \notin Q$, then Eq. 9 leads to a contradiction. Thus $y_{a+1} = q_{j+1}$ and so $c'_j \leq c < c'_{j+1} = c_{a+1}$. Thus, $i \leq a$, and $g(c) = g(c_i) \leq g(c_a) = g'(c'_j) = g'(c)$, which is a contradiction.

Assume $q_{(j+1)2} \leq (1+\epsilon)q_{j2}$. Let b be an index such that $y_b = q_{j+1}$. Then Eq. 10 implies

$$c_i \leq c < c'_{j+1} \leq c_b.$$

Thus $i < b$ or $g(c) = y_{i2} \leq y_{b2} = q_{(j+1)2}$. This leads to

$$\left|g(c) - g'(c)\right| \leq q_{(j+1)2} - q_{j2} \leq (1+\epsilon)q_{j2} - q_{j2} = \epsilon q_{j2} = \epsilon g'(c) < \epsilon g(c),$$

proving the proposition. □

A similar result also holds for $L_1(f)$. We omit the proof as it is very similar to the proof of Proposition 5.

Proposition 6 *Assume $\epsilon > 0$. Let Y be a convex hull of a ROC curve. Let Q be a subset of Y such that for each y_i, there is $q_j \in Q$ such that*

$$q_j = y_i \quad \text{or} \quad 1 - q_{(j+1)1} \leq 1 - y_{i1} \leq 1 - q_{j1} \leq (1+\epsilon)(1 - q_{(j+1)1}).$$

Let f be the function constructed from Y as given by Eq. 8, and let f' be a function constructed similarly from Q. Then f' is an ϵ-approximation of f.

The above propositions lead to the following strategy. Only use a subset of the ROC curve to compute the H-measure; if we select the points carefully, then the relative error will be less than ϵ.

Let us now focus on estimating $L_2(g)$. Assume that we have the convex hull $Y = \{y_0, \ldots, y_m\}$ of a ROC curve stored in a search tree T. Consider an algorithm given in Algorithm 3 which we call Subset.

Algorithm 3: SUBSET(u, p, q, ϵ), outputs truncated part of the convex hull tree. Here, u is the current node, p and q are the minimum and the maximum coordinates of the subtree rooted at u, and ϵ is the approximation guarantee.

```
1 if q_2 > (1 + ε)p_2 then
2 |   z ← p + d(u) + cd(left(u));
3 |   Report z;
4 |   SUBSET(left(u), p, z, ε);
5 |   SUBSET(right(u), z, q, ε);
```

The pseudo-code traverses T, and maintains two variables p and q that bound the points of the current subtree. If $q_2 \leq (1+\epsilon)p_2$, then we can safely ignore the current subtree, otherwise we output the current root, and recurse on both children. It is easy to see that $Q = \{y_0, y_m\} \cup \text{SUBSET}(r, 0, cd(r))$ satisfies the conditions of Proposition 5.

A similar traverse can be also done in order to estimate $L_1(f)$. However, we can estimate both values with the same subset by replacing the if-condition with $q_2 > (1+\epsilon)p_2$ **or** $1 - q_1 > (1+\epsilon)(1-p_1)$.

Proposition 7 SUBSET *runs in* $\mathcal{O}\left((1+\epsilon^{-1})\log^2 n\right)$ *time.*

Proof Given a node v, let us write T_v to mean the subtree rooted at v. Write p_v and q_v to be the values of p and q when processing v.

Let V be the reported nodes by SUBSET. Let $W \subseteq V$ be a set of m nodes that have two reported children. Let $\{h_1, \ldots, h_m\}$ be the non-normalized 2nd coordinate of nodes in W, ordered from smallest to largest.

Fix i and let u and v be the nodes corresponding to h_i and h_{i+1}. Assume that $v \notin T_u$. Let $r = \mathit{right}(u)$ be the right child of u. Then $T_r \cap W = \emptyset$ as otherwise h_i and h_{i+1} would not be consecutive. We have $h_{i+1} \geq q_{r2} > (1+\epsilon)p_{r2} = (1+\epsilon)h_i$.

Assume that $v \in T_u$ which immediately implies that $u \notin T_v$. Let $r = \mathit{left}(u)$ be the left child of v. Then $T_r \cap W = \emptyset$, and we have $h_{i+1} = q_{r2} > (1+\epsilon)p_{r2} \geq (1+\epsilon)h_i$.

In summary, $h_{i+1} > (1+\epsilon)h_i$. Since $\{h_i\}$ are integers, we have $h_2 \geq 1$. In addition, $h_m \leq n$ since the original data points (from which the ROC curve is computed) do not have weights.

Consequently, $n \geq h_m \geq (1+\epsilon)^{m-2}$. Solving m leads to $m \in \mathcal{O}\left(\log_{1+\epsilon} n\right) \subseteq \mathcal{O}\left((1+\epsilon^{-1})\log n\right)$.

Given $v \in W$, define $k(v)$ to be the number of nodes in $V \setminus W$ that have v as their youngest ancestor in W. The nodes contributing to $k(v)$ form at most two paths starting from v. Since the height of the search tree is in $\mathcal{O}(\log n)$, we have $k(v) \in \mathcal{O}(\log n)$.

Finally, we can bound $|V|$ by

$$|V| = \sum_{v \in W} 1 + k(v) \in \mathcal{O}(m \log n) \subseteq \mathcal{O}\left((1+\epsilon^{-1})\log^2 n\right),$$

concluding the proof. □

6.1 Speed-up

It is possible to reduce the running time of Subset to $\mathcal{O}\left(\log^2 n + \epsilon^{-1} \log n\right)$. We should point out that in practice Subset is probably a faster approach as the theoretical improvement is relatively modest but at the same time the overheads increase.

There are several ways to approach the speed-up. Note that the source of the additional $\log n$ term is that in the proof of Proposition 7, we have $k(v) \in \mathcal{O}(\log n)$. The loose bound is due to the fact that we are traversing a search tree balanced on tree height. We will modify the search procedure, so that we can show that $k(v) \in \mathcal{O}(1)$ which will give us the desired outcome. More specifically, we would like to traverse the hull using a search tree balanced using the 2nd coordinate.

The best candidate to replace the search tree for storing the convex hull is a weight-balanced tree (Nievergelt and Reingold 1973). Here, the subtrees are (roughly) balanced based on the number of children. The problem is that this tree, despite its name, does not allow weights for nodes. Moreover, the algorithm relies on the fact that the nodes have no weights.

It is possible to extend the weight-balanced trees to handle the weights but such modification is not trivial. Instead we demonstrate an alternative approach that is possible using only stock search structures.

We will do this by modifying the search tree T in which the nodes correspond to the partial hulls, see Fig. 3b.

Let Z be the current set of points and let $P = \{(s, \ell) \in Z \mid \ell = 2\}$ be the points with label equal to 2. Set $N = Z \setminus P$. We store P in a tree T of bounded balance; the points are only stored in leaves. Each leaf, say u, also stores all points in N that follow immediately u. These points are stored in a standard search tree, say L_u, so that we can join two trees or split them when needed. Any points in N that are without a preceding point in P are handled and stored separately.

Note that L_u correspond to a vertical line when drawing the ROC curve. Consequently, a point in the convex hull will always be the last point in L_u for some u. This allows us to define the weight $d(u)$ of a leaf u in T as $(m, 1)$, where m is the number of nodes in L_u. We now apply the convex hull maintenance algorithm on T. As always, we maintain the cumulative weights $cd(u)$ for the non-leaf nodes.

In order to approximate the H-measure we will use a variant of Subset, except that we will traverse T instead of traversing the hull. The pseudo-code is given in Algorithm 4. At each node we output the bridge, if it is included in the final convex hull. The condition is easy to test, we just need to make sure that it does not overlap with the previously reported bridges. Since we output both points of the bridge, this may lead to duplicate points, but we can prune them as a post-processing step. Finally, we truncate the traversal if the subtree is sandwiched between two bridges that are close enough to each other. It is easy to see that the output of SubsetAlt satisfies the conditions in Proposition 5 so we can use the output to estimate $L_2(g)$. In order to estimate $L_1(f)$ we duplicate the procedure, except we swap the labels and negate the scores which leads to a mirrored ROC curve.

Algorithm 4: SubsetAlt(u, o, p, q, ϵ), outputs truncated part of the convex hull tree. Here, u is the current node, o are the minimum coordinates of the subtree rooted at u, p and q are the coordinate bounds based on already reported bridges, and ϵ is the approximation guarantee.

```
1  if q_2 > (1 + ε)p_2 then
2      x, y ← o+ end points of the bridge related to u;
3      if y_2 > q_2 then
4          SubsetAlt(left(u), o, p, q, ε);
5      else if p_2 > x_2 then
6          SubsetAlt(right(u), o + cd(left(u)), p, q, ε);
7      else
8          Report x, y;
9          SubsetAlt(left(u), o, p, x, ε);
10         SubsetAlt(right(u), o + cd(left(u)), y, q, ε);
```

Proposition 8 SubsetAlt *runs in* $\mathcal{O}\left(\log^2 n + \epsilon^{-1} \log n\right)$ *time*.

Proof Let T be the tree traversed by SubsetAlt. Let us write T_v to be the subtree rooted at v.

Let $n(v)$ be the number of nodes in T_v, and let $\ell(v)$ be the number of leaves in T_v. Note that $n(v) = 2\ell(v) + 1$.

Let v be a child of u. Since T is a weight-balanced tree (Nievergelt and Reingold 1973), we have

$$\alpha \leq \frac{1 + \ell(v)}{1 + \ell(u)} = \frac{1 + 1 + 2\ell(v)}{1 + 1 + 2\ell(u)} = \frac{1 + n(v)}{1 + n(u)} \leq 1 - \alpha, \quad \text{where} \quad \alpha = \frac{1 - \sqrt{2}}{2}. \quad (11)$$

Let us write $o(v)$ to be the 2nd origin coordinate of T_v. Note that $o(v)$ corresponds to the variable o_2 in SubsetAlt when v is processed.

Let V be the set of nodes whose bridges we output, and let U be the set of nodes in T for which $\ell(u) > \epsilon o(u)$.

We will prove the claim by showing that $V \subseteq U$ and $|U| \in \mathcal{O}\left(\log^2 n + \epsilon^{-1} \log n\right)$.

To prove the first claim, let $v \in V$. Let p and q match the variables of SubsetAlt when v is visited. The points p and q correspond to the two leaves of T_v. In other words, $q_2 - p_2 \leq \ell(v)$, and $o(v) \leq p_2$. Thus,

$$\ell(v) \geq q_2 - p_2 > \epsilon p_2 \geq \epsilon o(v).$$

This proves that $v \in U$.

To bound $|U|$, let $W \subseteq U$ be a set of m nodes that have two children in U.

Define $\left(h_1, \ldots, h_m\right) = (o(right(v)) \mid v \in W)$ to be the sequence of the (non-normalized) 2nd coordinates of the right children of nodes in W, ordered from the smallest to the largest.

Fix i. Let $u \in W$ be the node for which $o(right(u)) = h_i$, and let $v \in W$ be the node for which $o(right(v)) = h_{i+1}$.

Assume that $h_i \leq o(v)$. Since $v \in W$, we have

$$h_{i+1} = o(right(v)) = o(v) + \ell(left(v)) > o(v) + \epsilon o(left(v)) = (1 + \epsilon)o(v) \geq (1 + \epsilon)h_i.$$

Assume that $h_i > o(v)$. Then $u \in T_{left(v)}$, and consequently $v \notin T_{right(u)}$. Thus, $T_{right(u)} \cap W = \emptyset$ as otherwise h_i and h_{i+1} are not consecutive. Since $right(u) \in U$, we have

$$h_{i+1} \geq o(right(u)) + \ell(right(u)) \geq (1+\epsilon)o(right(u)) = (1+\epsilon)h_i.$$

In summary, we have $h_{i+1} > (1+\epsilon)h_i$. Note that $h_1 \geq 1$. In addition, $h_m \leq n$ since the original data points (from which the ROC curve is computed) do not have weights.

Consequently, $n \geq h_m \geq (1+\epsilon)^{m-1}$. Solving m leads to

$$m \in \mathcal{O}\left(\log_{1+\epsilon} n\right) \subseteq \mathcal{O}\left((1+\epsilon^{-1})\log n\right).$$

Given $v \in W$, define $k(v)$ to be the number of nodes in $V \setminus W$ that have v as their youngest ancestor in W. The nodes contributing to $k(v)$ form at most two paths starting from v. Since the height of the search tree is in $\mathcal{O}(\log n)$, we have $k(v) \in \mathcal{O}(\log n)$.

Assume that $\epsilon > \alpha/2$ (recall that $\alpha = \frac{1}{2}(1-\sqrt{2})$). Then

$$|V| = \sum_{v \in W} 1 + k(v) \in \mathcal{O}(m \log n) \subseteq \mathcal{O}\left((1+\epsilon^{-1})\log^2 n\right) \subseteq \mathcal{O}\left(\log^2 n\right),$$

proving the proposition.

Assume that $\epsilon \leq \alpha/2$. Let $v \in W$ with $k(v) > 0$. Recall that the nodes corresponding to $k(v)$ form at most two paths. Let $u_1, \ldots, u_j$ be such a path.

Let w be a child of u_1 for which $w \notin U$. We have

$$\begin{aligned}
1 + \ell(u_1) &\leq \alpha^{-1}(1 + \ell(w)) && \text{(Eq. 11)}\\
&\leq \alpha^{-1}(1 + \epsilon o(w)) && \text{(w} \notin \text{U)}\\
&\leq \alpha^{-1}(1 + \epsilon(o(u_1) + \ell(u_1))) && (w \text{ is a child of } u_1)\\
&\leq \alpha^{-1}(1 + \epsilon o(u_1)) + \ell(u_1)/2, && (\epsilon \leq \alpha/2)
\end{aligned}$$

which in turns implies $1 + \ell(u_1) \leq 2\alpha^{-1}(1 + \epsilon o(u_1))$.

Applying Eq. 11 iteratively and the fact that $u_j \in U$, we see that

$$\begin{aligned}
1 + \epsilon o(u_1) &\leq 1 + \epsilon o(u_j) && (u_j \text{ is a child of } u_1)\\
&< 1 + \ell(u_j) && (u_j \in U)\\
&\leq (1-\alpha)^{j-1}(1 + \ell(u_1)) && \text{(Eq. 11 applied } j-1 \text{ times)}\\
&\leq (1-\alpha)^{j-1} 2\alpha^{-1}(1 + \epsilon o(u_1)).
\end{aligned}$$

Solving for j leads to

$$j \leq 1 + \log_{1-\alpha} \alpha/2 \in O(1),$$

and consequently $k(v) \in O(1)$. We conclude that

$$|V| = \sum_{v \in W} 1 + k(v) \in \mathcal{O}(m) \subseteq \mathcal{O}\left((1+\epsilon^{-1})\log n\right),$$

proving the proposition. □

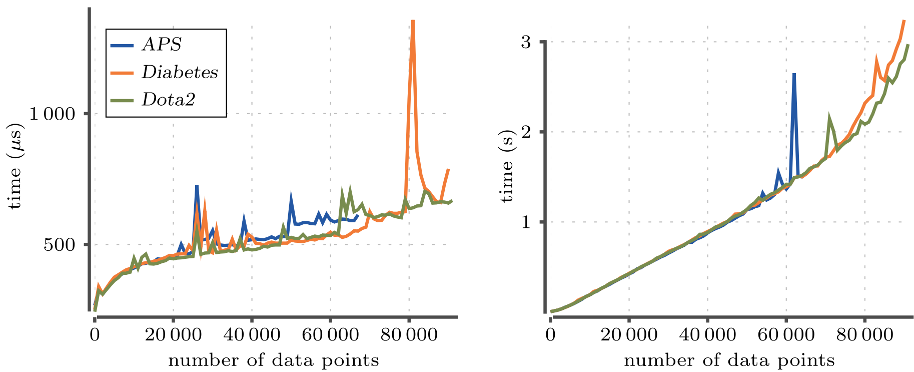


Fig. 4 Running time for computing AUC 1000 times as a function of the number of data points. Left figure: our approach. Right figure: baseline method by computing AUC from the maintained, sorted data points. Note that the time units are different

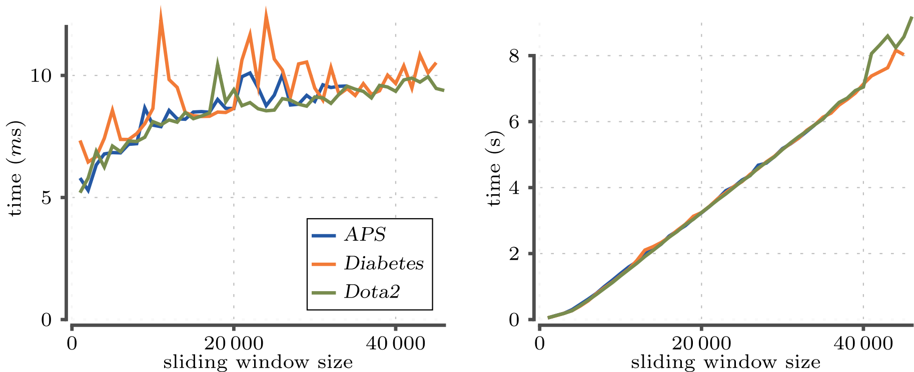


Fig. 5 Running time for computing AUC 10,000 times in a sliding window as a function of the size of the sliding window. Left figure: our approach. Right figure: baseline method by computing AUC from the maintained, sorted data points. Note that the time units are different

7 Experimental evaluation

In this section we present our experimental evaluation. Our primary focus is computational time. We implemented our algorithm using C++.[3] For convenience, we refer our algorithms as DynAuc, Hexact, and Happrox.

We used 3 datasets obtained from UCI repository:[4] *APS* contains APS failure in Scania trucks, *Diabetes* contains medical information of diabetes patients, here the label is whether the patient has been readmitted to a hospital, *Dota2* describes the character selection and the outcome of a popular competitive online computer game.

[3] Code is available at https://version.helsinki.fi/dacs/.

[4] http://archive.ics.uci.edu/ml/datasets.php.

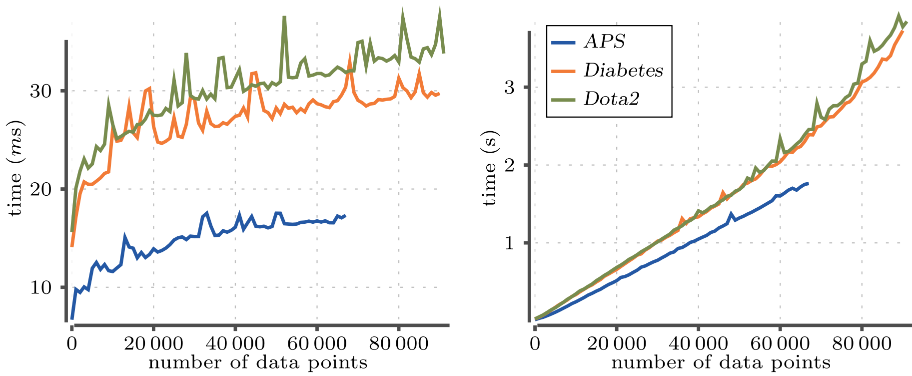


Fig. 6 Running time for computing the H-measure 1000 times as a function of the number of data points. Left figure: our approach. Right figure: baseline method computing from sorted data points. Note that the time units are different

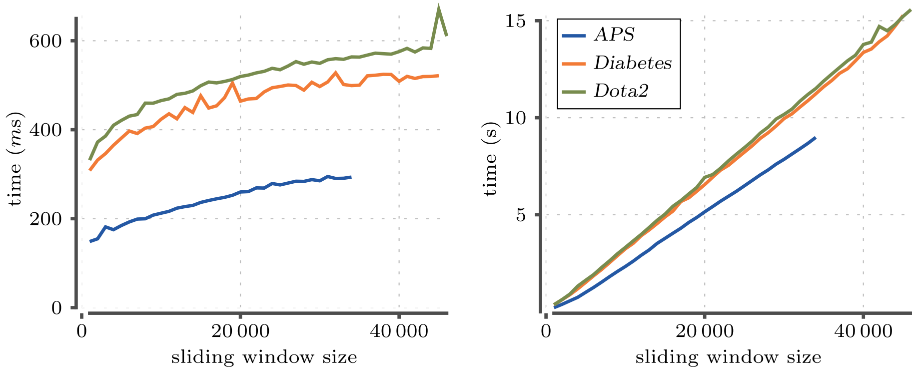


Fig. 7 Running time for computing H-measure 10 000 times in a sliding window as a function of the size of the sliding window. Left figure: our approach. Right figure: baseline method computing from sorted data points. Note that the time units are different

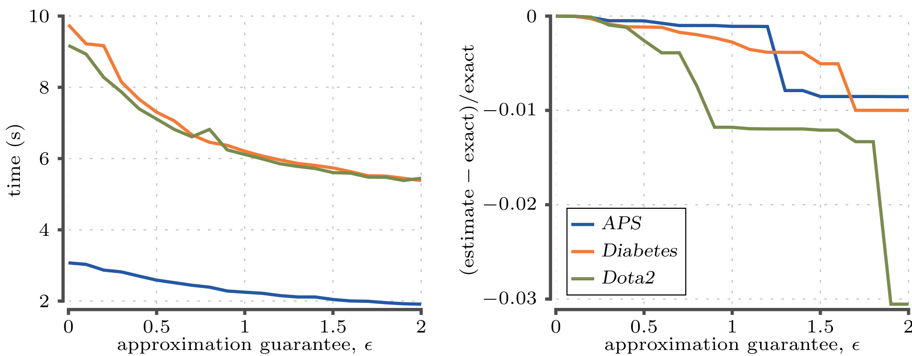


Fig. 8 Approximative H-measure as a function of approximation guarantee ϵ. Left figure: running time. Right figure: absolute difference to the correct value

We imputed the missing values with the corresponding means, and encoded the categorical features as binary features. We then proceeded to train a logistic regressor using 1/10th of the data, and used remaining data as testing data. When computing the *H*-measure we used beta distribution with $\alpha = \beta = 2$.

In our first experiment, we tested maintaining AUC as opposed to computing AUC by maintaining the points sorted and computing the AUC from the sorted list (Brzezinski and Stefanowski 2017). Given a sequence $z_1, \ldots, z_n$ of scores and labels, we compute AUC for $z_1, \ldots, z_i$ for every i. In the dynamic algorithm, this is done by simply adding the latest point to the existing structure. We record the time difference after 1000 additions.

From the results shown in Fig. 4 we see that DynAuc is about 10^4 times faster, though we should point out that the exact ratio depends heavily on the implementation. More importantly, the needed time increases logarithmically for DynAuc and linearly for the baseline. The spikes in running time of DynAuc are due to self-balancing search trees.

Next, we compare the running time of computing AUC in a sliding window. We use the same baseline as in the previous experiment, and record the running time after sliding a window for 10 000 steps. From the results shown in Fig. 5 we see that DynAuc is faster than the baseline by several orders of magnitude with the needed time increasing logarithmically for DynAuc and linearly for the baseline.

We repeat the same experiments but now we compare maintaining the *H*-measure against computing it from scratch from sorted data points. From the results shown in Figs. 6 and 7 we see that Hexact is about 10–10^2 times faster, and the time grows polylogarithmically for Hexact and linearly for the baseline. Similarly, the spikes in running time of Hexact are due to self-balancing search trees. Interestingly, Hexact is faster for *APS* than for the other datasets. This is probably due to the imbalanced labels, making the ROC curve relatively skewed, and the convex hull small.

In our final experiment we use approximative *H*-measure, without the speed-up described in Sect. 6.1. Here, we measure the *total* time to compute the *H*-measure for $z_1, \ldots, z_i$ for every i as a function of ϵ. Figure 8 shows the running time as well as the difference to the correct score when using the whole data.

Computing the *H*-measure from scratch required roughly 1 minute for *APS*, and 2.5 minutes for *Diabetes* and *Dota2*. On the other hand, we only need 10 seconds to obtain accurate result, and as we increase ϵ, the running time decreases. As we increase ϵ, the error grows but only modestly (up to 3%), with Happrox underestimating the exact value.

8 Conclusions

In this paper we considered maintaining AUC and the *H*-measure under addition and deletion. More specifically, we show that we can maintain AUC in $\mathcal{O}(\log n)$ time, and the *H*-measure in $\mathcal{O}(\log^2 n)$ time, assuming that the class priors are obtained from the testing data. We also considered the case, where the class priors are not obtained from the testing data. Here, we can approximate the *H*-measure in $\mathcal{O}((\log n + \epsilon^{-1}) \log n)$ time.

We demonstrate empirically that our algorithms, DynAuc and Hexact, provide significant speed-up over the natural baselines where we compute the score from the sorted, maintained data points.

When computing the *H*-measure the biggest time saving factor is maintaining the convex hull, as the hull is typically smaller than all the data points used for creating the ROC

curve. Because of the smaller size of the hull, the tricks employed by HAPPROX, provide less of a speed-up. Still, for larger values of ϵ, the speed-up can be almost 50%.

Appendix: Binary search for computing the bridge

Algorithm 5 contains a pseudo-code for finding the bridge of two convex hulls. The algorithm is a variation of the search described by Overmars and Van Leeuwen (1981). The main modification here is obtaining the ROC coordinates of the points.

Due to notational convenience, we write $p \preceq q$, where p and q are two points in a plane, if the slope of p is smaller than or equal to the slope of q.

Algorithm 5: BRIDGE(C_1, C_2), given two partial convex hulls C_1 and C_2, constructs a joint convex hull C by finding the end point of C_1 section and the starting point of C_2 section in C. Returns the end points and the coordinates of the bridge.

```
1  u1 ← root of C1;
2  u2 ← root of C2;
3  s1 ← s(u1);
4  s2 ← s(u2) + cd(u1);
5  σ ← x-coordinate of cd(root(C1)));
6  while u1 or u2 changes do
7      (x, y) ← intersection of lines s1 + d(next(u1)) × t and s2 + d(u2) × t;
8      if left(u1) and d(u1) ⪯ s2 − s1 then
9          s1 ← s1 + s(left(u1)) − s(u1);
10         u1 ← left(u1);
11     else if right(u2) and s2 − s1 ⪯ d(next(u2)) then
12         s2 ← s2 + s(right(u2));
13         u2 ← right(u2);
14     else if right(u1) and s2 − s1 ≺ d(next(u1)) and
15         (not left(u2) or s2 − s1 ⪯ d(u2) or x ≤ σ) then
16         s1 ← s1 + s(right(u1));
17         u1 ← right(u1);
18     else if left(u2) and d(u2) ≺ s2 − s1 and
19         (not right(u1) or d(next(u1)) ⪯ s2 − s1 or x ≥ σ) then
20         s2 ← s2 + s(left(u2)) − s(u2);
21         u2 ← left(u2);
22 return u1, u2, s1, s2;
```

Funding Open Access funding provided by University of Helsinki including Helsinki University Central Hospital. This project is funded by the University of Helsinki.

Availability of data and material Public datasets are available at http://archive.ics.uci.edu/ml/datasets.php.

Code Availability Code is available at https://version.helsinki.fi/dacs/.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the

material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Ataman, K., Streetr, W., & Zhang, Y. (2006). Learning to rank by maximizing auc with linear programming. In *IEEE international joint conference on neural networks, IJCNN'06, 2006* (pp. 123–129).

Bifet, A., & Frank, E. (2010). Sentiment knowledge discovery in twitter streaming data. In *Discovery science* (pp. 1–15). Springer.

Bouckaert, R.R. (2006). Efficient AUC learning curve calculation. In *Australasian joint conference on artificial intelligence* (pp. 181–191).

Brefeld, U., & Scheffer, T. (2005). Auc maximizing support vector learning. In *Proceedings of the ICML 2005 workshop on ROC analysis in machine learning*

Brodal, G.S., & Jacob, R. (2002). Dynamic planar convex hull. In *The 43rd annual IEEE symposium on foundations of computer science, 2002. Proceedings* (pp. 617–626). IEEE

Brzezinski, D., & Stefanowski, J. (2017). Prequential AUC: Properties of the area under the ROC curve for data streams with concept drift. *KAIS, 52*(2), 531–562.

Calders, T., & Jaroszewicz, S.: Efficient AUC optimization for classification. In *PKDD* (pp. 42–53).

Drummond, C., & Holte, R. C. (2006). Cost curves: An improved method for visualizing classifier performance. *Machine Learning, 65*(1), 95–130.

Ferri, C., Flach, P., & Hernández-Orallo, J. (2002). Learning decision trees using the area under the ROC curve. In *ICML* (vol. 2, pp. 139–146)

Flach, P.A., Hernández-Orallo, J., & Ramirez, C.F.: A coherent interpretation of auc as a measure of aggregated classification performance. In: ICML (2011)

Gama, J. (2010). *Knowledge discovery from data streams*. Boca Raton: CRC Press.

Gama, J., Sebastião, R., & Rodrigues, P. P. (2013). On evaluating stream learning algorithms. *Machine Learning, 90*(3), 317–346.

Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., & Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM Computing Surveys, 46*(4), 44.

Hand, D. J. (2009). Measuring classifier performance: A coherent alternative to the area under the ROC curve. *Machine Learning, 77*(1), 103–123.

Herschtal, A., & Raskutti, B. (2004). Optimising area under the roc curve using gradient descent. In *Proceedings of the twenty-first international conference on machine learning* (p. 49). ACM.

Mann, H.B., & Whitney, D.R. (1947). On a test of whether one of two random variables is stochastically larger than the other. In *The Annals of Mathematical Statistics* (pp. 50–60).

Nievergelt, J., & Reingold, E. M. (1973). Binary search trees of bounded balance. *SIAM Journal on Computing, 2*(1), 33–43.

Overmars, M. H., & Van Leeuwen, J. (1981). Maintenance of configurations in the plane. *Journal of computer and System Sciences, 23*(2), 166–204.

Tatti, N. (2018). Efficient estimation of auc in a sliding window. In *ECML PKDD* (pp. 671–686).

Žliobaitė, I., Bifet, A., Read, J., Pfahringer, B., & Holmes, G. (2015). Evaluation methods and decision theory for classification of streaming data with temporal dependence. *Machine Learning, 98*(3), 455–482.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.