Fully-Dynamic Decision Trees

Marco Bressan[1], **Gabriel Damay**[2], **Mauro Sozio**[2]

[1]University of Milan,
[2]Institut Polytechnique de Paris, Télécom Paris
marco.bressan@unimi.it, gabriel.damay@telecom-paris.fr, sozio@telecom-paris.fr

Abstract

We develop the first fully dynamic algorithm that maintains a decision tree over an arbitrary sequence of insertions and deletions of labeled examples. Given $\epsilon > 0$ our algorithm guarantees that, at every point in time, every node of the decision tree uses a split with Gini gain within an additive ϵ of the optimum. For real-valued features the algorithm has an amortized running time per insertion/deletion of $\mathcal{O}\big(\frac{d \log^3 n}{\epsilon^2}\big)$, which improves to $\mathcal{O}\big(\frac{d \log^2 n}{\epsilon}\big)$ for binary or categorical features, while it uses space $\mathcal{O}(nd)$, where n is the maximum number of examples at any point in time and d is the number of features. Our algorithm is nearly optimal, as we show that any algorithm with similar guarantees requires amortized running time $\Omega(d)$ and space $\widetilde{\Omega}(nd)$. We complement our theoretical results with an extensive experimental evaluation on real-world data, showing the effectiveness of our algorithm.

1 Introduction

Decision trees are a cornerstone of machine learning, and an essential tool in any machine learning library. Given a feature domain $\mathcal{X}$ and a label domain $\mathcal{Y}$, a decision tree is a function $f : \mathcal{X} \mapsto \mathcal{Y}$ that assigns to each $x \in \mathcal{X}$ a label $y \in \mathcal{Y}$ by traversing a tree T from its root node to a leaf. At each node of the tree, an example x is evaluated by some rule that determines which successor should receive x — for instance, a common rule is a simple threshold on some feature. Every leaf is associated with a label, which is the result of the prediction when such a leaf is reached. The problem of constructing an optimal decision tree is NP-hard w.r.t. to several natural objective functions (Shalev-Shwartz and Ben-David 2014). This has led to the introduction of several heuristic approaches, such as ID3, C4.5, C5.0 and CART, which have proven very effective and are now considered state of the art. Typically, those approaches proceed in a greedy fashion by selecting for each node a feature and a splitting value (we shall call such a pair a *split*) that optimize some measure of improvement such as the Gini gain or the information gain. This is repeated until a stopping condition is met, such as the tree reaching a certain height or the number of examples at every leaf falling below some threshold.

Recently, there have been significant efforts to adapt machine learning algorithms to a *fully dynamic* setting, where the algorithm is asked to process an arbitrary list of insertions or deletions. Insertions are typically the result of new data being collected or revealed, while deletions can be the result of noise removal, removal of personal data for privacy concerns, data becoming obsolete, etc. It might not be desirable to make any assumption on such a list of update operations, which motivates the design of fully dynamic algorithms. Most works in fully-dynamic machine learning algorithms have focused on unsupervised tasks such as clustering (Cohen-Addad et al. 2019; Henzinger and Kale 2020; Bateni et al. 2021; Chan, Guerqin, and Sozio 2018) or graph mining (Sawlani and Wang 2020; Bhattacharya et al. 2015; Epasto, Lattanzi, and Sozio 2015; De Stefani et al. 2017).

For decision trees, however, only *incremental* algorithms are known, which handle insertions but not deletions.[1] The state of the art in this case is given by Hoeffding Trees (Domingos and Hulten 2000) and their evolutions such as EFDT or HAT (see (Manapragada et al. 2022) for a survey). Not only are these algorithms incapable of handling deletions, but no good bound on their amortized cost is known, while guarantees hold only if the insertions are i.i.d. from some distribution. Our work represents one of the first studies on fully-dynamic supervised machine learning which has been mostly unexplored so far, to the best of our knowledge.

Defining what kind of decision tree a dynamic algorithm should maintain requires some care. The first natural attempt is to maintain the very same decision tree that the greedy approaches above (ID3, C4.5, etc.) would produce from the current set of examples. Thus, at any time, every node should use a split with maximal gain with respect to the set of examples held by its subtree. The problem with this goal is that, at some point, the gains of the best and second-best splits may differ by just $\mathcal{O}(1/n)$ where n is the total number of examples. In that case, $\mathcal{O}(1)$ updates can turn the second-best split into the best one, possibly forcing a reconstruction of the whole tree. The same happens if one wants splits within multiplicative factors of the best one, as the latter may be in $\mathcal{O}(1/n)$. Hence, in those cases it is unclear whether there is an efficient fully-dynamic algorithm. The next natural goal is maintaining a tree with ϵ-optimal splits, that is, within an

[1]These algorithms are also called *online* decision tree algorithms

additive ϵ of the best ones. We shall call such a decision tree ϵ-feasible. Aiming at an ϵ-feasible tree is reasonable, since excessively small gains are statistically not significant (a gain of, say, 10^{-4} is likely the result of noise) and thus approximating large gains is enough. Indeed, algorithms such as EFDT or HAT try to maintain precisely an ϵ-feasible tree. However, their approach is based on computing exactly the Gini gains. For real-valued features, doing that for every possible split would lead to an $\mathcal{O}(n)$ amortized cost. Hence, they resort to a heuristic which comes at the price of worse results.

In this work we develop an efficient fully-dynamic algorithm for maintaining an ϵ-feasible decision tree. Our first observation is that, in order to change by an additive term ϵ the gain of a given split on a sequence of examples S, one must make $\Omega(\epsilon|S|)$ insertions or deletions. Thus, one could rebuild a subtree after $\Theta(\epsilon|S|)$ updates, without even tracking the gains, with the number of updates covering the rebuilding cost. Intuitively speaking, this is one of the arguments we use in our amortized cost analysis, which is pretty standard. However, this yields amortized time bounds that are *quadratic* in the height h of the tree, because a sequence of updates can force a "cascade" of rebuilds on $\Theta(h)$ subtrees each having height $\Theta(h)$. We show how to bypass this obstacle and save a factor of h in the amortized cost with a "proactive" strategy that rebuilds subtrees slightly larger than necessary. Through a careful amortized analysis based on a few charging arguments, this yields our fully-dynamic algorithm for real-valued features. For categorical features, our algorithm can be improved via a faster tree reconstruction subroutine. Moreover, our algorithm can satisfy constraints more general than just ϵ-feasibility, including pruning at a certain height or guaranteeing splits only if enough examples are available. Finally, we prove that our algorithms are nearly optimal: no algorithm can beat their time or space usage by more than $\operatorname{poly}\log(nd)$ factors, even if one looks at algorithms attaining considerably weaker guarantees. Our contributions can be summarized as follows:

- We present FuDyADT, a deterministic algorithm for maintaining an ϵ-feasible decision tree under an arbitrary sequence of insertions and deletions. It uses $O(nd)$ space, while it has $O\big(\frac{d\log^3 n}{\epsilon^2}\big)$ amortized running time for real-valued features and $O\big(\frac{d\log^2 n}{\epsilon}\big)$ for categorical ones.
- We prove a lower bound of $\tilde{\Omega}(nd)$ on the space requirements and of $\Omega(d)$ on the amortized running time of any fully dynamic algorithm, even in easier settings. This makes FuDyADT optimal up to $\operatorname{poly}\log(nd)$ factors.
- We conduct an extensive experimental evaluation on real-world data, evaluating FuDyADT's speed and accuracy against state-of-the-art tools such as EFDT and HAT.

Related Work. The works closest to ours are those in the incremental setting. Here, the algorithm receives a stream of examples from a distribution, and has to perform well when compared to the offline tree built on the entire sequence. In this setting, Hoeffding trees (Domingos and Hulten 2000) emerged as one of the most effective approaches, inspiring several variants, even ones capable of handling concept drifts (Hulten, Spencer, and Domingos 2001; Gama, Rocha, and Medas 2003; Manapragada, Webb, and Salehi 2018; Das et al. 2019; Sun et al. 2020; Haug, Broelemann, and Kasneci 2022; Jin and Agrawal 2003; Rutkowski et al. 2013); see (Manapragada et al. 2022) for a survey. These algorithms crucially rely on the examples being i.i.d., which allows them to compute good splits with high probability via concentration bounds (whence the name). Moreover, those algorithms cannot handle efficiently real-valued features, since on those features they would update $\Theta(n)$ counters at each time, even when only insertions are allowed. Our algorithms instead efficiently handle arbitrary sequences of insertions and deletions of examples with real-valued features.

We observe that there are general techniques to turn offline data structures into dynamic ones, see (Bentley and Saxe 1980). Those techniques, however, work only for problems that have a special decomposability property — loosely speaking, the answer to a query (e.g., find $\min(X)$ for some set X) must be quickly computable from the answers to subqueries (e.g., $\min(A\cup B) = \min(\min(A), \min(B))$). In our case, a query corresponds to the label predicted by the tree for a given x. Unfortunately, our problem is far from decomposable and it does not seem solvable via such techniques.

2 Preliminaries

All missing proofs can be found in the full version (Bressan, Damay, and Sozio 2022). We denote the feature and label domains respectively by $\mathcal{X}$ and $\mathcal{Y}$; by default $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{0, 1\}$. We denote by $(x, y) \in \mathcal{X} \times \mathcal{Y}$ a labeled example, by x_j the value of its j-th feature, and by S a multiset of labeled examples. We may treat S as a sequence; this will be clear from the context. We assume examples can be stored in $\mathcal{O}(d)$ bits, while the x_j's can be accessed in time $\mathcal{O}(1)$. We let $S[\ldots]$ be the subset of S matching a condition; e.g. $S[x_j \le t] = \{(x, y) \in S : x_j \le t\}$. A *split* is a pair $(j, t) \in [d] \times \mathbb{R}$. We use the bold font for vectors (e.g. $\boldsymbol{x}$). We use Gini gain to measure split quality.

Definition 2.1. *The Gini index of S is $g(S) = 2\,p_S(1 - p_S)$, where $p_S = \frac{1}{|S|}\sum_{(x,y)\in S} y$. The Gini gain of (j, t) on S is:*

$$G(S, j, t) = g(S) - \left(\frac{|S_-|}{|S|}g(S_-) + \frac{|S_+|}{|S|}g(S_+)\right) \quad (1)$$

where $S_- = S[x_j \le t]$ and $S_+ = S[x_j > t]$. When $|S| = 0$ we define $g(S) = 0$ and $G(S, j, t) = 0$ for all $(j, t) \in [d]\times\mathbb{R}$.

For all $j \in [d]$ let $G(S, j) = \max_{t\in\mathbb{R}} G(S, j, t)$. Hence, $\arg\max_j G(S, j)$ is a feature with maximum Gini gain over S. Finally, we let $G(S) = \max_{j\in[d]} G(S, j)$.

We rely on the following smoothness properties of the Gini index and the Gini gain. Given two multisets/sequences S, S', their edit distance $\triangle(S, S')$ is the minimum number of insertions and deletions to obtain S' from S, and their relative edit distance is $\triangle^*(S, S') = \frac{\triangle(S,S')}{\max(|S|,|S'|)}$.

Lemma 2.2. *Let S, S' be multisets of labeled examples.*

1. $|G(S, j, t) - G(S', j, t)| \le 12\triangle^*(S, S'), \forall (j, t) \in [d]\times\mathbb{R}$
2. $|g(S) - g(S')| \le 2.5\,\triangle^*(S, S')$.

Decision trees. A decision tree is a triple (T, Σ, L), where $T = (V, A)$ is a directed binary tree rooted at $r(T)$, and Σ

and L are functions that assign splits to internal nodes and labels to the leaves. More formally $\Sigma = \{\sigma_v : v \in V(T)\}$ where $\sigma_v = (j_v, t_v) \in [d] \times \mathbb{R}$ for every internal node v of T, while $L = \{L_v : v \in V(T)\}$ where $L_v \in \{0, 1\}$ for every leaf v of T. For any $x \in \mathcal{X}$ and any internal vertex v of T let $\mathrm{succ}(v, x)$ be the left child of v if $x_j \leq t$ and the right child of v otherwise, where $\sigma_v = (j, t)$. For any $v \in V(T)$ let $P_v = (v_0, \ldots, v_\ell)$ be the unique path from $v_0 = r(T)$ to $v_\ell = v$. For a multiset S, denote by S_v the set of examples $x \in S$ such that $\mathrm{succ}(v_i, x) = v_{i+1}$ for all $i = 0, \ldots, \ell - 1$; this is the subset of S *associated* to v. For every $x \in \mathcal{X}$ let $v(x)$ be the leaf x is associated to. The labeling given by T is the function $T : \mathcal{X} \to \mathcal{Y}$ such that $T(x) = L_{v(x)}$ for every $x \in \mathcal{X}$. We denote by T_v the subtree of T rooted at v and by $(T, \Sigma, L)_v$ the decision subtree rooted at v.

Algorithms. A fully-dynamic decision tree algorithm $\mathcal{A}$ is defined as follows. The input of $\mathcal{A}$ is an *update sequence* U of requests of three types: insertion, $\textsc{ins}(x, y)$; deletion, $\textsc{del}(x, y)$; labeling, $\textsc{lab}(x)$. Each such sequence U induces an *active multiset* of labeled examples S obtained by inserting/deleting the examples following the order of the sequence. Suppose $\mathcal{A}$ has processed an update sequence U. We say $\mathcal{A}$ is *coherent* with a decision tree T if, for every $x \in \mathcal{X}$, any further request $\textsc{lab}(x)$ makes $\mathcal{A}$ output $T(x)$. The *query time* of $\mathcal{A}$ is the worst-case time it takes to $\mathcal{A}$ to output $T(x)$. Our goal is to construct a fully dynamic algorithm $\mathcal{A}$ that has low query time and, at every point in time, is coherent with a decision tree T that is $\boldsymbol{\epsilon}$-feasible with respect to the current active set S (see below).

3 A Fully Dynamic Decision Tree Algorithm

This section presents FuDyADT (Fully Dynamic Amortized Decision Tree). As argued in Section 1, one of our goals is to ensure that every node of the tree uses a split whose gain is within an additive ϵ of the maximum. FuDyADT satisfies a stricter guarantee, called $\boldsymbol{\epsilon}$-feasibility, which allows to also prune the tree at some height or at leaves with few examples.

Definition 3.1. *Let $k, h \in \mathbb{N}$, and let $\boldsymbol{\epsilon} = (\alpha, \beta)$ where $\alpha, \beta \in (0, 1]$. A decision tree (T, Σ, L) is $\boldsymbol{\epsilon}$-feasible, with pruning thresholds (k, h), w.r.t. a multiset S of labeled examples if for every $v \in V(T)$:*

1. *if $|S_v| \leq k$ or $g(S_v) = 0$ or $\mathrm{depth}_T(v) = h$ then v is a leaf, else if $g(S_v) \geq \alpha$ then v is an internal node*
2. *if $\sigma_v = (j, a)$ then $G(S_v, j, a) \geq G(S_v, j', a') - \beta$ for all $(j', a') \in [d] \times \mathbb{R}$*
3. *if v is a leaf then L_v is a majority label of S_v*

For any fixed pruning thresholds k, h we say that a fully dynamic algorithm $\mathcal{A}$ is $\boldsymbol{\epsilon}$-feasible if, at any point in time, $\mathcal{A}$ is coherent with a decision tree (T, Σ, L) that is $\boldsymbol{\epsilon}$-feasible with respect to the current active set. When $k = 1$ and $h = \infty$ and $\alpha = \beta$, $\boldsymbol{\epsilon}$-feasibility reduces to the following condition: if $g(S_v) = 0$ then v is a leaf, and if $g(S_v) \geq \alpha$ then v is internal and use an α-optimal split. This is the ϵ-optimality condition of Section 1 used by incremental algorithms such as Hoeffding trees and EFDT. We prove:

Theorem 3.2. *Let $\mathcal{X} = \mathbb{R}^d$, let k, h be positive integers, let $\alpha, \beta \in (0, 1]$, and let $0 < \epsilon < \min\left(\frac{1}{k+1}, \frac{\alpha}{5}, \frac{\beta}{12.5}\right)$. There is a deterministic (α, β)-feasible fully dynamic decision tree algorithm with pruning thresholds k, h that has query time $O(h^*)$, uses space $O(nd)$, and has amortized running time per update $O\left(\frac{dh^* \log^2 n}{\epsilon}\right) = O\left(\frac{d \log^3 n}{\epsilon^2}\right)$, where $h^* \leq h$ and n are respectively the maximum height of the tree and the maximum size of the active set at any time.*

Algorithm 1 FuDyADT.Update

```
1:  procedure Update((T, Σ, L), (x, y), o)
2:      P_{v_{κ_ℓ}} ← v_{κ_1}, ..., v_{κ_ℓ} with v_{κ_1} = r(T), v_{κ_ℓ} = v(x)
3:      update D_{v_{κ_ℓ}} and D^L_{v_{κ_ℓ}} according to (x, y), o
4:      L_{v_{κ_ℓ}} ← any majority label in D^L_{v_{κ_ℓ}}
5:      for i = 1, ..., ℓ do
6:          c(v_{κ_i}) ← c(v_{κ_i}) + 1
7:          if c(v_{κ_i}) > ε · s(v_{κ_i}) then
8:              ŝ ← 2^⌈log s(v_{κ_i})⌉
9:              j ← min{j' ∈ {0, ..., i} : s(v_{κ_{j'}}) ≤ ŝ}
10:             (T', Σ', L') ← Build(S_{v_{κ_j}}, i)
11:             (T, Σ, L)_{v_{κ_j}} ← (T', Σ', L')
12:             return
```

Theorem 3.2 can be improved for categorical features, that is, when $\mathcal{X} = A^d$ for some fixed finite set A; this includes the case of binary features, $A = \{0, 1\}$.

Theorem 3.3. *If $\mathcal{X} = A^d$ for a finite set A then the amortized time bound of Theorem 3.2 can be improved to $O\left(\frac{d \log^2 n}{\epsilon}\right)$.*

The rest of this section describes FuDyADT and proves Theorem 3.2, except for the $\boldsymbol{\epsilon}$-feasibility part, which is proven in the full version, as is Theorem 3.3. Before moving on, let us give some intuition on FuDyADT. The algorithm consists of the two routines Update and Build below. Those routines maintain a decision tree T, and, for each leaf v of T, dictionaries D_v and D_v^L storing respectively the multiset S_v associated to v and the frequency histogram of the labels of S_v. At every insertion or deletion of an example (x, v), Update computes the leaf $v = v(x)$ where x ends up, and updates D_v and D_v^L consequently, see line 3. Then, Update checks if any subtree should be rebuilt. To this end, for every vertex $u \in V(T)$ it maintains two counters, $s(u)$ and $c(u)$, storing respectively the size of the multiset on which the subtree T_u was rebuilt the last time and the number of updates that reached u since that time. As soon as $c(u) > \epsilon \cdot s(u)$ for some $u \in V(T)$, Update invokes Build to rebuild an appropriately chosen supertree of T_u.

Let us move to the bounds of Theorem 3.2. Proving those bounds requires some care in charging the cost of rebuilding the subtrees to the update requests. To this end, we need the following two simple results proven in the full version. From now on, by "time t" we mean the t-th invocation of Update.

Lemma 3.4. *Let (T, Σ, L) be the result of $t \geq 0$ invocations of* Update. *Then $(1 - \epsilon) \cdot s^t(v) \leq |S_v^t| \leq (1 + \epsilon) \cdot s^t(v)$ for every $v \in V(T)$, where $s^t(v)$ is the value of $s(v)$ at time t.*

Lemma 3.5. *Let (T, Σ, L) be a decision tree built on a sequence S. If every $v \in V(T)$ uses a split with gain at least $\gamma > 0$ w.r.t. S_v, then T has height $\mathcal{O}(\log |S| \,/\, \gamma)$.*

Algorithm 2 FUDYADT.BUILD

1: **procedure** BUILD(S, η)
2: $r \leftarrow$ new vertex, $c(r) \leftarrow 0$, $s(r) \leftarrow |S|$
3: **if** $|S| \leq k$ or $g(S) \leq \frac{\alpha}{2}$ or $\eta = h$ **then**
4: store S in a dynamic dictionary D_r
5: and its labels in a dynamic dictionary D_r^L
6: $(T, \Sigma, L) \leftarrow$ decision tree with $T = (\{r\}, \emptyset)$
7: $L_r \leftarrow$ any majority label in D_r^L
8: **else if** $g(S) > \frac{\alpha}{2}$ **then**
9: $(j, a) \leftarrow \arg\max\{G(S, \hat{\imath}, \hat{a}) : (\hat{\imath}, \hat{a}) \in [d] \times \mathbb{R}\}$
10: $T_1 \leftarrow$ BUILD($S[x_j \leq a], \eta + 1$)
11: $T_2 \leftarrow$ BUILD($S[x_j > a], \eta + 1$)
12: $(T, \Sigma, L) \leftarrow$ decision tree with root r, T_1, T_2 as left, right subtrees, and split $\sigma_r = (j, a)$
13: **return** (T, Σ, L)

We can now prove:

Lemma 3.6. BUILD *and* UPDATE *can be implemented so that any $\mathcal{T}$ invocations of* UPDATE *take time*

$$O\left(\frac{\mathcal{T} \cdot d \cdot h \cdot (\log n)^2}{\epsilon}\right) = O\left(\frac{\mathcal{T} \cdot d \cdot (\log n)^3}{\epsilon^2}\right) \tag{2}$$

Proof. First we describe the data structures and the time taken by the basic operations of BUILD and UPDATE. Using a self-balancing tree for D_v we ensure search, insert, update, and deletion in time $O(d \log N)$, and enumeration in time $O(dN)$ — recall that every element takes $O(d)$ bits — where N is the number of distinct entries in the data structure. The same for D_v^L, which has at most 2 distinct entries. Thus the block at line 3 of BUILD(S, i) runs in time $\mathcal{O}(d|S| \log |S|)$.

If instead the condition at line 3 fails, then BUILD must compute (j, a). To this end one proceeds as follows. First, for each $j \in [d]$ one computes the projection $S_{|j}$ of S on the j-th feature (keeping the label as well). Then one sorts $S_{|j}$ according to the feature values in time $\mathcal{O}(|S| \log |S|)$. Next, one scans $S_{|j}$ and finds the threshold t^* for which a split on j yields maximum gain in time $\mathcal{O}(|S|)$. To this end one just needs to keep label counts for the subsequence formed by the first i examples in $S_{|j}$, so that the gain a split at that point would yield can be computed in time $\mathcal{O}(1)$ from the counts of the first $(i-1)$ examples. Summarizing, one can compute the optimal split (j, a) in time $\mathcal{O}(d|S| \log |S|)$. Since $|S[x_j \leq a]| + |S[x_j > a]| = |S|$, it follows that BUILD$(S, i)$ always runs in time $O(d|S| \log |S|(h + \log |S|))$, which is in $O(d|S| \log n(h + \log n))$ since $|S| \leq n$ by definition of n. For UPDATE, computing $v_{\kappa_1}, \ldots, v_{\kappa_\ell}$ takes time $O(h)$, while performing any INS or DEL operation on $S_{v_{\kappa_\ell}}$ takes time $O(d \log |S^t|) = O(d \log n)$. Finally, computing the input $S_{v_{\kappa_j}}$ of BUILD takes time $O(|S_{v_{\kappa_j}}|)$ by visiting $T_{v_{\kappa_j}}$ and listing the data structures at its leaves.

Now, we bound the total time taken by $\mathcal{T}$ successive invocations of UPDATE. Let $B = \{t \in [\mathcal{T}] :$ BUILD is invoked at time $t\}$. For every $t \in B$ let $b(t)$ be such that $v_{b(t)}$ is the vertex v_{κ_j} on which BUILD is invoked. The total running time $\mathrm{cost}(\mathcal{T})$ of the $\mathcal{T}$ invocations satisfies:

$$\mathrm{cost}(\mathcal{T}) \leq \sum_{t=1}^{\mathcal{T}} O(h + d \log n) + \sum_{t \in B} O\Big((h + \log n) \cdot \log n \cdot d \cdot |S^t_{v_{b(t)}}|\Big) \tag{3}$$

The first term contributes $O(\mathcal{T}(h + d \log n))$. We now bound the second term. For every $t \in B$ consider the t-th execution of UPDATE. Let $s^t(v)$ be the value of $s(v)$ right before BUILD is invoked, and $v_{i(t)}$ be the vertex that satisfies the condition at line 7 of UPDATE. Note that $v_{i(t)}$ is by construction a descendant of $v_{b(t)}$. Finally, for $v \in \{v_{b(t)}, v_{i(t)}\}$ let $c^t(v)$ and $s^t(v)$ be the values of $c(v)$ and $s(v)$ right after line 6 is executed with $v_{\kappa_i} = v$. Then:

$$\sum_{t \in B} |S^t_{v_{b(t)}}| \leq 2 \cdot \sum_{t \in B} s^t(v_{b(t)}) \tag{4}$$

$$\leq 4 \cdot \sum_{t \in B} s^t(v_{i(t)}) \tag{5}$$

$$\leq \frac{4}{\epsilon} \cdot \sum_{t \in B} c^t(v_{i(t)}) \tag{6}$$

$$\leq \frac{4}{\epsilon} \cdot \sum_{t \in B} c^t(v_{b(t)}) \tag{7}$$

where (4) follows from Lemma 3.4 noting that $\epsilon \leq 1$, (5) and (6) follow respectively from lines 8-9 and line 7 of UPDATE, and (7) follows from the fact that $c^t(v) \leq c^t(u)$ if v is a descendant of u. Now observe that at every time t at most h counters $c(v)$ are increased by one unit; therefore,

$$\sum_{t \in B} c^t(v_{b(t)}) \leq |B| \cdot h \leq \mathcal{T} \cdot h \tag{8}$$

We conclude that $\sum_{t \in B} |S^t_{v_{b(t)}}| \leq \mathcal{T} \frac{4h}{\epsilon}$. Plugging this bound in (3) and noting that the second sum dominates, we obtain:

$$\mathrm{cost}(\mathcal{T}) = O\left(\mathcal{T} \cdot (h + \log n) \cdot \log n \cdot d \cdot \frac{h}{\epsilon}\right) \tag{9}$$

Next we prove a second bound on $\mathrm{cost}(\mathcal{T})$; the final bound comes from taking the minimum.

Consider the t-th execution of UPDATE, let (x, y) the example that is inserted or removed, and recall that $P_{v(x)}$ is the path from the root of the tree to the leaf $v(x)$ determined by x. Let $C^t = \{v_{k_1}, \ldots, v_{k_M}\}$ be the set of all vertices of $P_{v(x)}$ such that BUILD$(S^{t_i}_{v_{k_i}}, i)$ is executed at some time $t_i \geq t$.

If $C^t \neq \emptyset$ then we call C^t a *charging set*. We wish to bound the maximum size of C^t, which might be seen as the number of BUILD operations performed per update. We shall prove the following two properties:

P1: $\forall t \in [\mathcal{T}], |C^t| \leq \lceil \log(n) \rceil$
P2: $\forall t \in B, c^t(v_{b(t)}) \leq \sum_{\tau=1}^{t} \mathbb{1}_{v_{b(t)} \in C^\tau}$

We start with P1. We argue that there cannot be distinct nodes v_{k_i} and v_{k_j} in C^t such that $\lceil \log s^\tau(v_{k_i}) \rceil = \lceil \log s^\tau(v_{k_j}) \rceil$ for any $\tau \in [t_i, t_j]$. Suppose $\tau = t_i$; v_{k_i} cannot be an ancestor

of v_{k_j}, for otherwise v_{k_j} would not be connected to the root node at time t_j and $\textsc{Build}(S^{t_j}_{v_{k_j}}, j)$ would not be executed. If v_{k_j} is an ancestor of v_{k_i}, Update would not have performed $\textsc{Build}(S^{t_i}_{v_{k_i}}, i)$ at time t_i, in that, there is at least one other node (v_{k_j}) which is closer to the root and that would have been selected instead (line 9 of Update). The claim holds for every τ, as $s^\tau(v_{k_j}) = s^{t_i}(v_{k_j})$, for every $\tau \in [t_i, t_j]$. Therefore $|C^t| \leq \lceil \log n \rceil$.

For P2 we proceed as follows. Let $t_0 < t$ be such that $c(v_{b(t)})$ is set to 0 by Build (i.e., the point in time when $v_{b(t)}$ was created by Build, line 2), and let $q = c^t(v_{b(t)})$. By construction of Update, there are q distinct times $t_0 + 1 \leq \tau_1 < \cdots < \tau_q \leq t$ such that, for every $i \in [q]$, we have $c^{\tau_i}(v_{b(t)}) = c^{\tau_i - 1}(v_{b(t)}) + 1$ and $v_{b(t)} \in C^{\tau_i}$, proving P2.

We obtain the following chain of inequalities:

$$\sum_{t \in B} c^t(v_{b(t)}) \leq \sum_{t \in B} \sum_{\tau=1}^{t} \mathbb{1}_{v_{b(t)} \in C^\tau} \leq \sum_{t \in B} |C^\tau| \tag{10}$$

where the inequalities follow respectively from P2 and from the fact that $v_{b(t)} \neq v_{b(t')}$ for $t \neq t'$. Using P1 and $B \subseteq [\mathcal{T}]$, we conclude that $\sum_{t \in B} c^t(v_{b(t)}) = O(\mathcal{T} \cdot \log n)$. Plugging this bound into (3) yields:

$$\mathrm{cost}(\mathcal{T}) = O\left(\mathcal{T} \cdot (h + \log n) \cdot \log n \cdot d \cdot \frac{\log n}{\epsilon}\right) \tag{11}$$

Taking the minimum of (9) and (11) yields that $\mathrm{cost}(\mathcal{T})$ is in

$$\mathcal{O}\left(\mathcal{T} \cdot (h + \log n) \cdot \log n \cdot d \cdot \frac{\min(h, \log n)}{\epsilon}\right) \tag{12}$$

As $(x+y)\min(x,y) \leq 2xy$ for $x, y \geq 0$ we conclude that $\mathrm{cost}(\mathcal{T}) = \mathcal{O}\big(\mathcal{T} d h (\log n)^2/\epsilon\big)$. Noting that $h \in \mathcal{O}\big(\frac{\log n}{\epsilon}\big)$ by Lemma 3.5 concludes the proof. □

4 Lower Bounds

This section proves lower bounds on the space and amortized running time used by any fully dynamic algorithm for our problem. These bounds hold even under significant relaxations of both the input access model and the constraints of Section 3. Notably, they hold for randomized algorithms that can fail with constant probability under inputs provided by oblivious adversaries.

To state our bounds we need some more definitions. A label y is $\boldsymbol{\epsilon}$-feasible for $x \in \mathcal{X}$ w.r.t. S if there is a decision tree (T, Σ, L) that is $\boldsymbol{\epsilon}$-feasible w.r.t. S such that $T(x) = y$. Note that there might be multiple $\boldsymbol{\epsilon}$-feasible labels for x. A decision tree algorithm $\mathcal{A}$ is *weakly* $(\boldsymbol{\epsilon}, \delta)$-feasible w.r.t. S if for every $x \in \mathcal{X}$ there is a decision tree (T_x, Σ_x, L_x) that is $\boldsymbol{\epsilon}$-feasible w.r.t. S and such that $\Pr(\mathcal{A}_S(x) = T_x(x)) \geq \delta$. Note that $(\boldsymbol{\epsilon}, \delta)$-feasibility is much weaker than $\boldsymbol{\epsilon}$-feasibility: not only it allows the algorithm to fail, but it does not even require it to be coherent with any given $\boldsymbol{\epsilon}$-feasible tree.

Theorem 4.1. *Let $k^*, h^* \geq 1$, and let $\boldsymbol{\epsilon} = (\alpha, \beta)$ with $0 \leq \alpha \leq 1$ and $0 \leq \beta < \frac{1}{24}$. Any weakly $(\boldsymbol{\epsilon}, \frac{3}{4})$-feasible fully dynamic algorithm with pruning thresholds k^*, h^* uses space $\Omega\big(\frac{n \cdot d}{k \cdot \log n}\big)$, where d is the number of features and n is the maximum size of the active set at any point in time.*

Proof. We reduce from the following classic two-party communication problem called Index. Alice is given a string $x \in \{0,1\}^N$ and Bob is given an integer $i \in [N]$. Alice is allowed to send one message $\mathcal{M} \in \{0,1\}^*$ to Bob, which, after receiving $\mathcal{M}$, outputs a single bit. The goal of Bob is to output precisely x_i. It is well known that for Bob to succeed with probability greater than $\frac{3}{4}$ we must have $|\mathcal{M}| = \Omega(N)$, see (Henzinger and Kale 2020).

We reduce Index to the construction of an $(\boldsymbol{\epsilon}, \frac{3}{4})$-feasible fully dynamic algorithm. For some positive integers N, D, Alice is given an arbitrary string in $\{0,1\}^{ND}$ representing a matrix $A \in \{0,1\}^{N \times D}$. Bob is given a pair $(\kappa, \ell) \in [N] \times [D]$ and must output $A_{\kappa\ell}$. By the lower bound above, Alice must send to Bob $\Omega(ND)$ bits in order for Bob to succeed with probability greater than $\frac{3}{4}$.

The reduction is as follows. Let $k = k^*$. First, Alice computes the following sequence S of $|S| = N \cdot D \cdot 2k$ examples. Let $\bar{D} := \lceil \log(N + D) + 1 \rceil$, and for all $i \in [N + D]$ let $\boldsymbol{b}_i \in \{0,1\}^{\bar{D}}$ be the binary representation of i. For simplicity and w.l.g. we assume k to be an even integer. For every $i \in [N + D]$ Alice constructs $2k$ examples $(\boldsymbol{x}_i^1, y_i^1), \ldots, (\boldsymbol{x}_i^{2k}, y_i^{2k})$ with $\mathcal{X} = \{0,1\}^{D+\bar{D}}$ and $\mathcal{Y} = \{0,1\}$, as follows. For every $i \in [N + D]$ and every $h \in [2k]$, the last $\bar{D}$ bits of $\boldsymbol{x}_i^h$ correspond to the string $\boldsymbol{b}_i$ (i.e., $x_{ij}^h = b_{ij}$ for all $j \in [D + 1, D + \bar{D}]$), and $y_i^h = \mathbb{1}_{h > k}$. The remaining bits of $\boldsymbol{x}_i^h$ are defined as follows. If $i \in [N]$, then for all $j \in [D]$:

$$x_{ij}^h := \begin{cases} 1 - A_{ij}, & h \in [k]; \\ A_{ij}, & h \in [k+1, 2k]; \end{cases}$$

while if $i \in [N + 1, N + D]$, then for all $j \in [D]$ and $h \in [2k]$:

$$x_{ij}^h := \begin{cases} 1, & j \in [D] \setminus \{i - N\}, h \bmod 2 = 0; \\ 0, & \text{otherwise}; \end{cases}$$

Let $\mathcal{A}$ be any $(\boldsymbol{\epsilon}, \frac{3}{4})$-feasible fully dynamic algorithm with $\beta < \frac{1}{24}$. Alice asks $\mathcal{A}$ to add every element of S, then she sends a snapshot of its memory to Bob, which resumes the execution of $\mathcal{A}$. Next, Bob asks $\mathcal{A}$ to perform $\textsc{del}(\boldsymbol{x}, y)$ for every $y \in \{0,1\}$ and every $\boldsymbol{x} \in \{0,1\}^{D+\bar{D}}$ terminating with the $\bar{D}$-bits binary string $\boldsymbol{b}_i$, for all $i \in [N + D] \setminus \{\kappa, N + \ell\}$. Finally, Bob asks $\mathcal{A}$ to label $\mathbf{1}$, and outputs the answer.

First, we claim that Bob outputs $A_{\kappa\ell}$. To prove this, note that the active set received by $\mathcal{A}$ is:

$$\widehat{S} = \big\{(\boldsymbol{x}_\kappa^h, y_\kappa^h) : h \in [k]\big\} \cup \big\{(\boldsymbol{x}_{N+\ell}^h, y_{N+\ell}^h) : h \in [k]\big\}$$

We prove that any decision tree that is $\boldsymbol{\epsilon}$-feasible with respect to $\widehat{S}$ labels the example $\mathbf{1}$ with $A_{\kappa\ell}$. To this end we show that in any such tree, (i) the root splits on feature ℓ, (ii) the child v of the root corresponding to feature ℓ equal to 1 is a leaf with label $A_{\kappa\ell}$. For (i), we prove that $\widehat{S}$ does not meet any stopping condition, and that j is the only β-optimal feature. The claim on the stopping condition is immediate. For the optimality of ℓ, we claim that:

$$G(\widehat{S}, j) = \begin{cases} 1/6, & j = \ell \\ 1/8, & j \in [D] \setminus \{\ell\} \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

To begin, note that $g(\widehat{S}) = \frac{1}{2}$. Let $\widehat{S}_1$ and $\widehat{S}_2$ be the two subsequences obtained by splitting $\widehat{S}$ on j. When $j \in \ell$, $\widehat{S}_1$ contains k examples with identical labels, so $g(\widehat{S}_1) = 0$, while $\widehat{S}_2$ contains $3k$ examples of which $2k$ have identical label, so $g(\widehat{S}_2) = \frac{4}{9}$. Thus $G(\widehat{S}, \ell) = \frac{1}{2} - \left(\frac{1}{4} \cdot 0 + \frac{3}{4} \cdot \frac{4}{9}\right) = \frac{1}{6}$. When $j \in [D] \setminus \{\ell\}$, both $\widehat{S}_1$ and $\widehat{S}_2$ contain k examples of $\left\{(\boldsymbol{x}^h_\kappa, y^h_\kappa) : h \in [k]\right\}$ with identical label, as well as k examples of $\left\{(\boldsymbol{x}^h_{N+\ell}, y^h_{N+\ell}) : h \in [k]\right\}$ with $\frac{k}{2}$ labels to 0 and $\frac{k}{2}$ labels to 1. Thus, in both $\widehat{S}_1$ and $\widehat{S}_2$ one label occurs precisely on a fraction $\frac{3}{4}$ of the examples. Hence $g(\widehat{S}_1) = g(\widehat{S}_2) = 2 \cdot \frac{3}{4} \cdot \frac{1}{4} = \frac{3}{8}$, and $G(\widehat{S}, j) = \frac{1}{8}$. In every other case, either $\widehat{S}_1 = \widehat{S}$, or $\widehat{S}_1 = \left\{(\boldsymbol{x}^h_\kappa, y^h_\kappa) : h \in [2k]\right\}$ and $\widehat{S}_2 = \left\{(\boldsymbol{x}^h_{N+\ell}, y^h_{N+\ell}) : h \in [2k]\right\}$, which implies $g(\widehat{S}_1) = g(\widehat{S}_2) = \frac{1}{2}$ and $G(\widehat{S}, j) = 0$. Since $\beta < \frac{1}{24} = \frac{1}{6} - \frac{1}{8}$, we conclude that ℓ is the only β-optimal feature, as desired.

For (ii), note that the subsequence of $\widehat{S}$ having the ℓ-th feature set to 1 has all labels equal to $A_{\kappa\ell}$. This implies that the corresponding child v of the root is a leaf, since it meets at least one of the stopping conditions, and that it assigns label $A_{\kappa\ell}$. This proves that Bob returns $A_{\kappa\ell}$.

To prove the space lower bound, note that S consists of $n = 2k(N + D)$ examples, each of which can be encoded in $d = D + \bar{D} = O(D + \log(D + N))$ bits. For $D = O(N)$, this yields $n = O(kN)$ and $d = O(D \log N)$ and therefore $ND = \Omega\left(\frac{nd}{k \log n}\right)$. Recalling that Alice must send $\Omega(ND)$ bits to Bob concludes the proof. □

We conclude this section with a lower bound on the running time of any fully dynamic algorithm. Clearly, if the model requires the algorithm to read every labeled example (x, y) upon arrival, then a lower bound of $\Omega(nd)$ is trivial. However, we show that an $\Omega(nd)$ bounds holds even if we do not require the algorithm to read the examples; instead, at any point in time we allow the algorithm to access in time $\mathcal{O}(1)$ the j-th feature of *any* example in the current active set. We call this the *matrix access model*. Again, we prove the bound for weakly $(\boldsymbol{\epsilon}, \delta)$-feasible algorithms.

Theorem 4.2. *Let $k, h \geq 1$ and $\alpha, \beta \in [0, \frac{1}{2})$. For arbitrarily large n and d there exist sequences of n* INS *and* DEL *operations over $\{0,1\}^d \times \{0,1\}$ such that, in the matrix access model, any weakly $(\boldsymbol{\epsilon}, 2/3)$-feasible fully dynamic algorithm has expected running time $\Omega(nd)$.*

5 Experiments

We compare FUDYADT against two state-of-the-art algorithms for incremental decision tree learning, EFDT (Manapragada, Webb, and Salehi 2018) and HAT (Bifet and Gavaldà 2009), using the MOA software (Bifet et al. 2010). Similarly to FUDYADT, EFDT and HAT aim at keeping ϵ-optimal splits, which they do with high probability when the examples are i.i.d. from a distribution. Due to space limitations, the results for HAT and other experiments can be found in the full version of the paper (Bressan, Damay, and Sozio 2022).

Settings. We implemented FUDYADT in C++ [2]. We conducted all experiments on an Ubuntu 20.04.2 LTS server equipped with 144 Intel(R) Xeon(R) Gold 6154 @ 3.00GHz CPUs and 264 GB of RAM. We observe that the algorithms have not been implemented in the same programming language, which limits the relevance of the runtime comparison.

Datasets. Our datasets are shown in Table 1. We have chosen them among standard datasets for classification; some of them, such as INSECTS, feature the so-called concept drift. Not all datasets have binary labels. For the INSECTS datasets, we assigned label 1 to the union of `male` classes. For every other dataset, we assigned label 1 to the majority class.

Input models. We consider three input models. Let $(x_1, y_1), \ldots, (x_T, y_T)$ be the sequence of examples as given by the dataset at hand (typically in chronological order). The simplest model is when only insertions are allowed aka *incremental* model. Formally, at every $t \in [T]$ the algorithm receives $\text{INS}(x_t, y_t)$. This model is supported by all algorithms (FUDYADT, EDFT, HAT), hence we use it to compare them against each other. The next two models involve deletions and thus are supported only by FUDYADT. The first one is the *sliding window* model (SW): given an integer $W \geq 1$ for all $t \in [T]$ the algorithm receives $\text{INS}(x_t, y_t)$, preceded by $\text{DEL}(x_{t-W+1}, y_{t-W+1})$ if $t \geq W$. The second one is the *random update* model (RU): for all $t \in [T]$, with probability $1/2$ the algorithm receives $\text{INS}(x_t, y_t)$ and with probability $1/2$ it receives $\text{DEL}(x, y)$ where (x, y) is chosen uniformly at random from the active set S^t.

Metrics. As is customary in the literature, we evaluate how well each algorithm predicts the label of the next example before "seeing" it. Formally, if (x_t, y_t) is the t-th example appearing in the input sequence, then we compute $\hat{y}_t = \text{LAB}(x_t)$ *before* the algorithm sees (x_t, y_t). We then compute the F1-score of the label sequence $\hat{\boldsymbol{y}} = (\hat{y}_t)_{t \geq 1}$ against the ground-truth $\boldsymbol{y} = (y_t)_{t \geq 1}$,

$$\text{F1}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{2 \cdot P(\boldsymbol{y}, \hat{\boldsymbol{y}}) \cdot R(\boldsymbol{y}, \hat{\boldsymbol{y}})}{P(\boldsymbol{y}, \hat{\boldsymbol{y}}) + R(\boldsymbol{y}, \hat{\boldsymbol{y}})} \tag{14}$$

where $P(\boldsymbol{y}, \hat{\boldsymbol{y}})$ and $P(\boldsymbol{y}, \hat{\boldsymbol{y}})$ are respectively the precision and recall of $\hat{\boldsymbol{y}}$ against $\boldsymbol{y}$. The F1-score is in $[0, 1]$ with higher values denoting better results.

Parameters. For FUDYADT, we let $\alpha = 0$, $\beta = 0$ $k = 1$, $h \in \{5, 10\}$, and we manually set $\epsilon \in [0, 2]$. Note that it breaks the condition of theorem 3.2. It allows us to test the effect of ϵ without fine-tuning of the other parameters. The parameters of EFDT and HAT are set to the original values specified by the authors; we only vary the so-called grace period in $\{100, 500, 1000\}$ to find the value yielding highest F1-score. For the SW model we use $W \in \{100, 1000\}$. In all our experiments, we first build a decision tree for the first W examples, then we apply the models above to the remaining sequence. Several parameter configurations show similar trends. We only report the most interesting results.

FUDYADT versus EFDT. We compare the F1-scores of EFDT and FUDYADT when allowed the same amortized time. To this end we tuned FUDYADT's ϵ to make its running time very close to (and never exceeding) that of EFDT.

[2] https://github.com/GDamay/dynamic-tree

	d	# of examples	1-class
Electricity	8	45 311	UP
Forest Covertype	54	581 011	2
INSECTS v1-v5	33	24 150 – 79 986	*-male
KDDCUP99	41	494 021	smurf.
NOAA Weather	8	18 159	1
Poker	10	829 201	0

Table 1: Datasets statistics.

	EFDT		FuDyADT		
	RT	F1	RT	F1	ϵ
Electricity	1.65	83.64	1.53	**90.33**	0.15
Forest Covertype	42.47	83.64	42.37	**90.33**	0.29
INSECTS v1	4.85	88.96	4.51	**92.17**	1.00
INSECTS v2	3.13	87.40	3.09	**92.53**	0.92
INSECTS v3	7.54	92.51	7.43	**94.76**	1.00
INSECTS v4	6.30	91.15	6.02	**91.91**	0.95
INSECTS v5	6.84	89.85	6.77	**93.34**	1.00
KDDCUP99	17.72	97.98	17.43	**99.91**	0.17
NOAA Weather	0.73	80.78	0.73	**81.43**	0.36
Poker	16.26	79.69	16.14	**86.07**	1.03

Table 2: Running time in seconds (labeled *RT*) and F1-score (labeled *F1*) of EFDT and FuDyADT in the incremental model. The last column shows the value of ϵ in Update.

The results are shown in Table 2; remarkably, FuDyADT outperforms consistently EFDT in terms of F1-score. One of the possible reasons is that FuDyADT can guarantee to be relatively close to the optimal Gini gain, even without computing it explicitly. In contrast, EFDT resorts to an approximation which might be relatively poor, given that maintaining an optimal Gini gain is expensive.

FuDyADT on SW and RU. Next, we studied the performance of FuDyADT in the SW and RU models (recall that EFDT/HAT do not work here). For the SW model, we set $h = 10$, $k = 1$, $\alpha = 0$, and $W = 100$ for Electricity and $W = 1000$ otherwise. Figure 1 shows the F1 score as a function of ϵ (subfigures a-c) and the average time per update in milliseconds in logarithmic scale as a function of ϵ (subfigures d-f). The smaller ϵ is, the more often subtrees are recomputed, yielding a higher F1 score and amortized running time. This behavior is clear in the Electricity and Poker datasets, where from $\epsilon = 0$ to $\epsilon = 1$ the F1 score decreases by roughly 0.1 and the running time increases by three orders of magnitude. A good tradeoff could be $\epsilon = 0.1$, where the F1-score is close to that of $\epsilon = 0$ but with an amortized running time per update smaller by orders of magnitude (≈ 0.5ms). For INSECTS the F1-score is much more stable. All other datasets and parameter settings yielded very similar qualitative behaviors. Figure 2 shows the average running time for the RU model, showing similar trends to Figure 1.

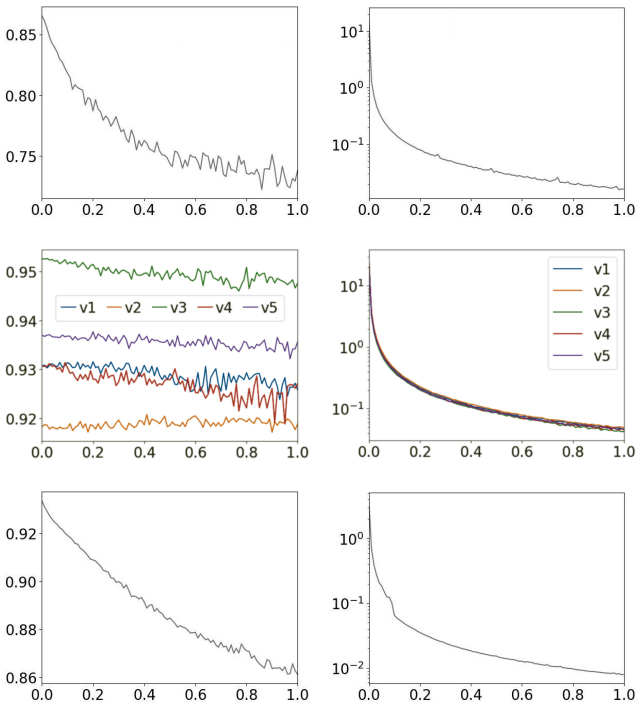

Figure 1: Performance of FuDyADT in the SW model on the Electricity, INSECTS and Poker datasets (top to bottom), in terms of F1-score (left) and amortized milliseconds per update (right) as a function of ϵ.

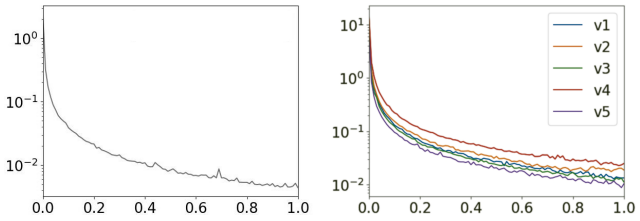

Figure 2: Amortized running time per update (in milliseconds) of FuDyADT in the RU model on the Electricity (left) and INSECTS (right) datasets. The Poker dataset yields similar results.

6 Conclusions and Future Work

We developed the first fully dynamic algorithm for maintaining ϵ-feasible decision trees, while we proved it to be nearly optimal in terms of space and amortized time. Our work shows that many well-known decision tree algorithms, whether offline like CART or incremental like EDFT, can be made fully dynamic with a small loss in the quality of the decision tree and a small overhead in the amortized running time. Our work leaves open the natural question of whether these results can be strengthened from amortized to worst-case. We believe this is an exciting direction for future research in fully-dynamic supervised machine learning.

Acknowledgements

The work of Gabriel Damay and Mauro Sozio was partially supported by the French National Agency (ANR) under project APY (ANR-20-CE38-0011), while it has been carried out partially in the frame of a cooperation between Huawei Technologies France SASU and Telecom Paris (Grant no. YBN2018125164). Marco Bressan has been supported in part by the EU Horizon 2020 ICT-48 research and innovation action under grant agreement 951847, project ELISE (European Learning and Intelligent Systems Excellence), and by the FAIR (Future Artificial Intelligence Research) project, funded by the NextGenerationEU program within the PNRR-PE-AI scheme (M4C2, investment 1.3, line on Artificial Intelligence). We thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions.

References

Bateni, M.; Esfandiari, H.; Jayaram, R.; and Mirrokni, V. S. 2021. Optimal Fully Dynamic k-Centers Clustering. *CoRR*, abs/2112.07050.

Bentley, J. L.; and Saxe, J. B. 1980. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1(4): 301–358.

Bhattacharya, S.; Henzinger, M.; Nanongkai, D.; and Tsourakakis, C. E. 2015. Space- and Time-Efficient Algorithm for Maintaining Dense Subgraphs on One-Pass Dynamic Streams. In *Proc. of ACM STOC*.

Bifet, A.; and Gavaldà, R. 2009. Adaptive Learning from Evolving Data Streams. In *Proc. of IDA*, 249–260. Berlin, Heidelberg: Springer-Verlag.

Bifet, A.; Holmes, G.; Kirkby, R.; and Pfahringer, B. 2010. MOA: Massive Online Analysis. *J. Mach. Learn. Res.*, 11: 1601–1604.

Bressan, M.; Damay, G.; and Sozio, M. 2022. Fully-Dynamic Decision Trees. *CoRR*, abs/2212.00778.

Chan, T. H.; Guerqin, A.; and Sozio, M. 2018. Fully Dynamic *k*-Center Clustering. In *Proc. of WWW*.

Cohen-Addad, V.; Hjuler, N.; Parotsidis, N.; Saulpic, D.; and Schwiegelshohn, C. 2019. Fully Dynamic Consistent Facility Location. In *Proc. of NeurIPS*.

Das, A.; Wang, J.; Gandhi, S. M.; Lee, J.; Wang, W.; and Zaniolo, C. 2019. Learn Smart with Less: Building Better Online Decision Trees with Fewer Training Examples. In *Proc. of IJCAI*, 2209–2215.

De Stefani, L.; Epasto, A.; Riondato, M.; and Upfal, E. 2017. TRIÈST: Counting Local and Global Triangles in Fully Dynamic Streams with Fixed Memory Size. *ACM Trans. Knowl. Discov. Data*.

Domingos, P.; and Hulten, G. 2000. Mining High-Speed Data Streams. In *Proc. of ACM KDD*, 71–80.

Epasto, A.; Lattanzi, S.; and Sozio, M. 2015. Efficient Densest Subgraph Computation in Evolving Graphs. In *Proc. of WWW*.

Gama, J. a.; Rocha, R.; and Medas, P. 2003. Accurate Decision Trees for Mining High-Speed Data Streams. In *Proc. of ACM KDD*, 523–528.

Haug, J.; Broelemann, K.; and Kasneci, G. 2022. Dynamic Model Tree for Interpretable Data Stream Learning. In *Proc. of IEEE ICDE*, 2562–2574.

Henzinger, M.; and Kale, S. 2020. Fully-Dynamic Coresets. In *Proc. of ESA*, volume 173 of *LIPIcs*, 57:1–57:21.

Hulten, G.; Spencer, L.; and Domingos, P. 2001. Mining Time-Changing Data Streams. In *Proc. of ACM KDD*, 97–106.

Jin, R.; and Agrawal, G. 2003. Efficient Decision Tree Construction on Streaming Data. In *Proc. of ACM KDD*, 571–576.

Manapragada, C.; Gomes, H. M.; Salehi, M.; Bifet, A.; and Webb, G. I. 2022. An eager splitting strategy for online decision trees in ensembles. *Data Mining and Knowledge Discovery*, 36(2): 566–619.

Manapragada, C.; Webb, G. I.; and Salehi, M. 2018. Extremely Fast Decision Tree. In *Proc. of ACM KDD*, 1953–1962.

Rutkowski, L.; Pietruczuk, L.; Duda, P.; and Jaworski, M. 2013. Decision Trees for Mining Data Streams Based on the McDiarmid's Bound. *IEEE Transactions on Knowledge and Data Engineering*, 25(6): 1272–1279.

Sawlani, S.; and Wang, J. 2020. Near-optimal fully dynamic densest subgraph. In *Proc. of ACM STOC*, 181–193.

Shalev-Shwartz, S.; and Ben-David, S. 2014. *Understanding Machine Learning: From Theory to Algorithms*. USA: Cambridge University Press.

Sun, J.; Jia, H.; Hu, B.; Huang, X.; Zhang, H.; Wan, H.; and Zhao, X. 2020. Speeding up Very Fast Decision Tree with Low Computational Cost. In *Proc. of IJCAI*, 1272–1278.