HedgeCut: Maintaining Randomised Trees for Low-Latency Machine Unlearning

Sebastian Schelter
s.schelter@uva.nl
University of Amsterdam

Stefan Grafberger
s.grafberger@uva.nl
University of Amsterdam

Ted Dunning
ted.dunning@gmail.com
Hewlett Packard Enterprise

ABSTRACT

Software systems that learn from user data with machine learning (ML) have become ubiquitous over the last years. Recent law such as the "General Data Protection Regulation" (GDPR) requires organisations that process personal data to delete user data upon request (enacting the "right to be forgotten"). However, this regulation does not only require the deletion of user data from databases, but also applies to ML models that have been learned from the stored data. We therefore argue that ML applications should offer users to *unlearn* their data from trained models in a timely manner. We explore how fast this unlearning can be done under the constraints imposed by real world deployments, and introduce the problem of *low-latency machine unlearning*: maintaining a deployed ML model in-place under the removal of a small fraction of training samples without retraining.

We propose *HedgeCut*, a classification model based on an ensemble of randomised decision trees, which is designed to answer unlearning requests with low latency. We detail how to efficiently implement HedgeCut with vectorised operators for decision tree learning. We conduct an experimental evaluation on five privacy-sensitive datasets, where we find that HedgeCut can unlearn training samples with a latency of around 100 microseconds and answers up to 36,000 prediction requests per second, while providing a training time and predictive accuracy similar to widely used implementations of tree-based ML models such as Random Forests.

ACM Reference Format:
Sebastian Schelter, Stefan Grafberger, and Ted Dunning. 2021. HedgeCut: Maintaining Randomised Trees for Low-Latency Machine Unlearning. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457239

1 INTRODUCTION

Software systems that learn from user data with machine learning (ML) have become ubiquitous over the last years [33], and participate in many critical decision-making processes, e.g., about loans, job applications and medical treatments [36]. Recent laws such as the "right to be forgotten" [14] (Article 17 of the General Data Protection Regulation (GDPR)) require companies and institutions that process personal data to delete user data upon request: "The data subject shall have the right to [...] the erasure of personal data concerning him or her without undue delay [...] where the data subject withdraws consent".

SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457239

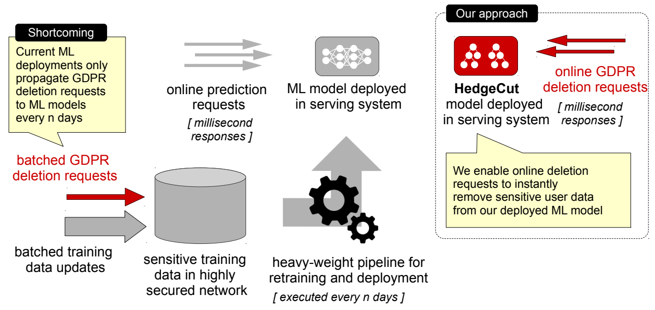

Shortcoming
Current ML deployments only propagate GDPR deletion requests to ML models every n days
online prediction requests
[millisecond responses]
ML model deployed in serving system
Our approach
HedgeCut model deployed in serving system
online GDPR deletion requests
[millisecond responses]
We enable online deletion requests to instantly remove sensitive user data from our deployed ML model
batched GDPR deletion requests
batched training data updates
sensitive training data in highly secured network
heavy-weight pipeline for retraining and deployment
[executed every n days]

Figure 1: Current ML setups update ML models with respect to GDPR deletion requests only when the model is retrained and redeployed via heavy-weight pipelines. We propose *HedgeCut*, a custom ML model which enables online deletion requests in-place that remove sensitive user data from the deployed model with millisecond latencies.

Enacting the right to be forgotten. Making data processing systems GDPR-compliant has been identified as one of the key research challenges for the data management community [35], was a central topic in a recent panel discussion at SIGMOD [3], and is for example addressed by enhancing fundamental data structures with efficient delete operations [31]. Recent research [8, 18, 32, 39] argues that it is not sufficient to merely delete personal user data from primary data stores such as databases, and that machine learning models that have been trained on the stored data also fall under the regulation. This view is supported by Recital 75 of the GDPR [16]: "The risk to the rights and freedoms of natural persons [...] may result from personal data processing [...] where personal aspects are evaluated, in particular analysing or predicting aspects concerning performance at work, economic situation, health, personal preferences or interests, reliability or behaviour, location or movements".

A data management perspective on machine unlearning. Relational databases offer transactional deletes and corresponding updates for materialized views [19] over the data, however there exists no such automated deletion mechanism for ML models derived from the data. The machine learning community has been working on this issue in recent years under the umbrella of *machine unlearning* [8, 9, 18, 21, 26]. Given a model, its training data and a set of user data to unlearn, they propose efficient ways to accelerate the retraining of the model. However, these approaches *lack a data management perspective*, as they ignore the constraints imposed by the complexity of deployment pipelines in real-world ML applications [27, 33, 34] as sketched in Figure 1. ML models are deployed in serving systems that can efficiently answer online prediction requests with low latency, but in order to update a model, heavy-weight pipelines that have to spin up infrastructure, retrain the model and run expensive evaluation workloads have to be executed [1, 5, 27]. Data removal in response to GDPR requests is typically integrated into these pipelines, and will only affect the model once it is being redeployed via the heavyweight pipeline.

As a concrete example, let us look at the high costs of retraining and redeploying models, when using Spark MLlib [25] in a real-world deployment: (1) Before we can even start training, we need to provision machines in the cloud; (2) Next, we need to startup Spark on the cluster, and read the training data for the model from its original location (typically a secured distributed filesystem) into the aggregate main memory of the cluster; (3) Now, we can execute the retraining of the model from scratch based on the updated training data; (4) Afterwards, a set of additional sanity tests is typically run, e.g., inspecting predictions that changed, or comparing the performance of the new model to the performance of the current model in backtesting scenarios [33]; (5) Finally, we initiate the redeployment into the serving system, often with 'canary' and 'rollback' steps [27] to be able to react to flawed model versions; in many cases, this redeployment might require us to spin up new serving machines so that traffic can be rerouted atomically.

Going through this whole process only to unlearn a single training example does not make sense economically and operationally. However, ideally, we would still like to be able to instantly unlearn single examples between regular redeployments. For that, we have to be able to completely bypass the heavy-weight redeployment steps. We can retrieve the data of the user to delete with a point query, and we would like to be able to issue a deletion request with this data to the serving system, to simply update the deployed model in place (instead of having to train and deploy a new model).

Low-latency machine unlearning. In this paper, we propose such a technique to perform *low-latency machine unlearning*: we instantly respond to a GDPR deletion request (refered to as "unlearning request") by updating the deployed ML model in-place to remove the effect of the data of the user, *without having to retrain the model*, as illustrated in the right part of Figure 1. We ensure that the model after the update corresponds to a model that the training algorithm would have produced if it had been trained from scratch without the data of the user to unlearn.

The GDPR law does not specify how soon data must be erased after a deletion request [35], it states the "obligation to erase personal data without undue delay" [14] using "appropriate and effective measures" [15]. Currently, data erasure seems to be a rather tedious and lengthy process in practice; data erasure from active systems in the Google cloud, for example, can take up to two months.[1] We argue that it is an open and important academic question to determine how fast data can be erased from deployed ML models, and that *we should design systems that empower users to exercise their right to be forgotten as timely as possible*, instead of making them wait for several months.

[1]https://cloud.google.com/security/deletion

Model maintenance under unlearning. We focus on supervised classification tasks, and propose a new ML model called *HedgeCut*, which can efficiently unlearn a small fraction of its training data after deployment (Section 4). HedgeCut is able to respond to online unlearning requests with a latency of less than a millisecond, similar to the latency with which traditional models respond to online prediction requests. HedgeCut is a variation of the well-established "Extremely Randomised Trees" (ERT) approach [17], which learns an ensemble of randomized decision trees where attributes and cut-off points to split the data are chosen at random.

We efficiently maintain this tree ensemble under data removal as follows. We introduce a novel notion of *split robustness* to identify splitting decisions in the tree ensemble which potentially change in response to removed data (Section 4.2). We make HedgeCut learn and maintain so-called subtree variants for these cases, e.g., we learn additional subtrees of the randomised tree ensemble which the model uses in cases when the data removal changes the split decision. We argue that such deletion only needs to be possible for a small fraction of the training data, as only a small fraction of the users typically issue GDPR deletion requests (Section 2). We talked to industry practitioners from cloud and e-commerce vendors, who estimate that typically only one in ten-thousand users will issue a deletion request. We design our methods to be able to handle an order of magnitude more deletion requests (one in a thousand users) to be on the safe side.

We discuss an efficient, multi-threaded implementation of HedgeCut in Rust, and describe how to apply vectorisation [4, 22, 41] to accelerate scan-intensive parts of the learning procedure (Section 5). We conduct an extensive experimental evaluation of HedgeCut in Section 6, where we analyse its ability to unlearn user data with low latency (Section 6.2.1), while still providing a high throughput for prediction requests (Section 6.2.2). Furthermore, we compare its training time and predictive performance to popular, optimised Cython implementations of other tree-based methods like Random Forests in Section 6.3.1. We make our implementation available[2] under an open source license.

In summary, this paper makes the following contributions.

- We introduce the problem of *low-latency machine unlearning*: maintaining a deployed ML model in-place under the removal of a small fraction of samples without retraining (Section 2).
- We propose Hedgecut, a classification model based on an ensemble of randomised decision trees, which incorporates a novel notion of split robustness and efficiently answers unlearning requests for a small fraction of training samples (Section 4).
- We detail how to efficiently implement HedgeCut by designing vectorised operators for decision tree learning (Section 5).
- We conduct an experimental evaluation on five privacy-sensitive datasets, where we find that HedgeCut can unlearn training samples with a latency of around 100 microseconds and answer up to 36,000 prediction requests per second, while providing a training time and predictive accuracy similar to widely used implementations of tree-based ML models such as Random Forests (Section 6).

[2]https://github.com/schelterlabs/hedgecut

2 PROBLEM STATEMENT

In the following, we formalise the problem in the focus of this paper: unlearning a small fraction of training samples from a trained and deployed ML model without reaccessing its training data. We build upon the initial problem introduced in our previous work [32].

Let $t_{\text{learn}}(\mathbf{D}, \theta)$ denote a procedure to learn an ML model f (such as a classifier) from training data $\mathbf{D}$ with hyperparameters θ. Let $\mathbf{D} = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_M\}$ denote the data of M users (our training data), and let $f = t_{\text{learn}}(\mathbf{D}, \theta)$ denote the ML model learned from the user data $\mathbf{D}$. Now, assume that we are required to delete the data $\mathbf{d}_{\text{unlearn}}$ of a particular user from the model. That means we are interested in obtaining another ML model $f' = t_{\text{learn}}(\mathbf{D} \setminus \mathbf{d}_{\text{unlearn}}, \theta)$, which never saw the data $\mathbf{d}_{\text{unlearn}}$ of the user to unlearn during its training. This new model can be trivially obtained by repeating the training procedure t_{learn} on $\mathbf{D} \setminus \mathbf{d}_{\text{unlearn}}$.

Unlearning without retraining. Conducting a complete model retraining is inefficient and costly in real world deployments, where retraining and redeployment require the execution of complex heavy-weight data pipelines. We can furthermore assume that the loss of data from a single user (or a very small number of users) does not require a different choice of hyperparameters (we validate this claim experimentally in Section 6.3.1). Instead, we would be interested in an efficient procedure t_{unlearn} that can inspect $\mathbf{d}_{\text{unlearn}}$ and update the existing model f to the desired model f', such that $f' = t_{\text{unlearn}}(f, \{\mathbf{d}_{\text{unlearn}}\}, \theta) = t_{\text{learn}}(\mathbf{D} \setminus \{\mathbf{d}_{\text{unlearn}}\}, \theta)$.

This approach is refered to as "decremental learning" or "machine unlearning" in the ML literature [9]. Unlearning only needs to take place for the small fraction of users that might want to have their data removed. We talked to industry practitioners from cloud and e-commerce vendors, who estimate that typically only one in ten-thousand users issues a deletion request. Formally, the unlearning procedure t_{unlearn} must satisfy the following property for a small fraction ϵ of unlearnable training samples: $\forall \mathbf{D}_r \subset \mathbf{D}, |\mathbf{D}_r| \leq \epsilon|\mathbf{D}| : t_{\text{unlearn}}(f, \mathbf{D}_r, \theta) = t_{\text{learn}}(\mathbf{D} \setminus \mathbf{D}_r, \theta)$.

The unlearning procedure t_{unlearn} should also exhibit a runtime that is much lower than the runtime of t_{learn} for retraining, ideally similar to the time to compute a prediction for an unseen sample from the model. Note that we also require that t_{unlearn} does not reaccess the training data, indicated by the fact that the function signature of t_{unlearn} does not require access to $\mathbf{D}$.

3 BACKGROUND

We briefly introduce the background for "Extremely Randomised Trees" and split selection in decision trees based on Gini gain, on which our approach builds later on.

Extremely Randomised Trees (ERTs). Tree-based methods are a well-studied, non-linear supervised learning approach, which has been shown to perform well on many problems. Tree-based methods recursively partition the feature space, so that the resulting hyperboxes contain a set of data points that give a strong indication about the target variable.

We focus on an approach to tree learning called *Extremely Randomised Trees (ERT)* [17]. This approach builds an ensemble of decision trees where both the attributes as well as the cut-off points to split the data in a node are chosen at random independent of target attribute, and the decision which split to use is made based on some criterion measuring the "purity" of the resulting splits like information gain or Gini impurity. Even though this approach is conceptually simple, its predictive performance is highly competitive with more popular ML models like Random Forests, and established ML libraries such as scikit-klearn include implementations of this approach.

Algorithm 1 illustrates the splitting procedure for numerical attributes in ERTs performed by the function SPLIT_A_NODE. We select k non-constant attributes as split candidates for a local subset of the training samples, and select the cut point for these attributes at random from the range of the data in the function RANDOM_SPLIT. The best split is selected via the SCORE function which measures the purity of the split afterwards, and we stop splitting the data once the subset to split only contains n_{min} samples or is constant in the attribute values or labels (as shown in STOP_SPLIT).

Algorithm 1 Splitting procedure in Extremely Randomised Trees.

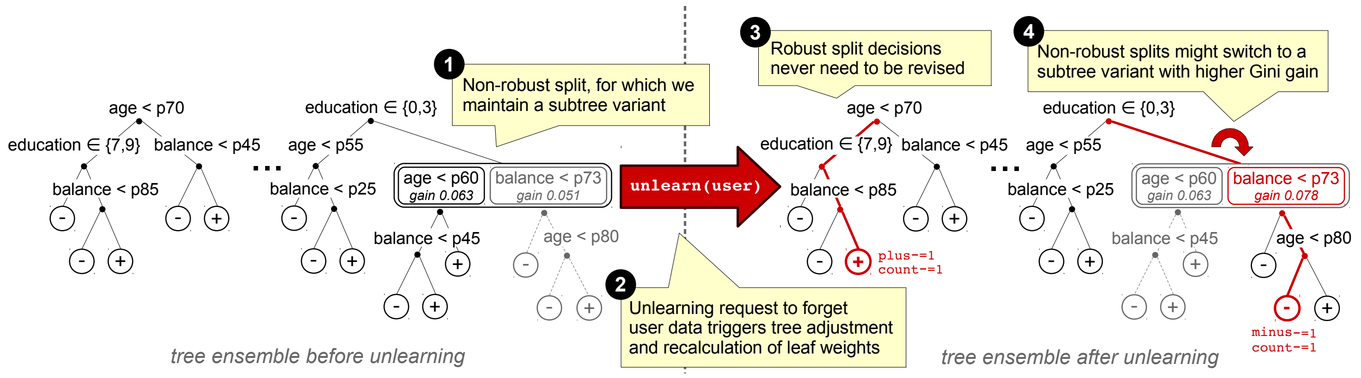
```
1:  function SPLIT_A_NODE($D$)
        Input: local subset of training samples $D$
        Output: a split $[a, a_c]$ or a leaf
2:      if STOP_SPLIT($D$)
3:          return a leaf labeled according to class frequencies in $D$
4:      else
5:          Randomly select $k$ non-constant attributes $\{a_1, \ldots, a_k\}$ in $D$
6:          Generate $k$ splits $\{s_1, \ldots, s_k\}$, where $s_i \leftarrow$ RANDOM_SPLIT($a_i$)
7:          return best split $s^* \leftarrow \text{argmax}_{i=1..k}$ SCORE($s_i$, $D$)

8:  function STOP_SPLIT($D$)
9:      if $|S| \leq n_{\text{min}}$ or all attribute values constant in $D$ or label constant in $D$
10:         return true
11:     return false

12: function RANDOM_SPLIT($a$, $D$)
13:     $a^D_{\text{min}}$, $a^D_{\text{max}} \leftarrow$ minimal and maximal value of $a$ in $D$
14:     $a_c \leftarrow$ random cut point from the range $[a^D_{\text{min}}, a^D_{\text{max}}]$
15:     return $[a < a_c]$
```

Split selection via Gini gain. As mentioned in the previous paragraph, decision tree learning is based on a measure of the "purity" gain of splitting a subset of the training data based on a chosen attribute. Common choices for measuring this purity include normalized information gain [38] or Gini impurity. While empirical studies find that both measures produce equally good models [30] in terms of their predictive performance, the reduction in Gini impurity is often the preferred measure in practice as it does not require the costly computation of logarithms.

The gain in Gini impurity for the general case of splitting a dataset into a left and right partition for a classification task with K different classes is measured as follows:

$$\sum_{c=1}^{K} p(c)p(\neg c) - \left[w_l \sum_{c=1}^{K} p_l(c)p_l(\neg c) + w_r \sum_{c=1}^{K} p_r(c)p_r(\neg c) \right]$$

where $\sum_{c=1}^{K} p(c)p(\neg c)$ denotes the impurity before conducting a split, with $p(c)$ being the probability of picking an element of class c at random from the data, and $p(\neg c)$ the probability of picking an element from a different class. When evaluating a split, we partition the data into a left and right part according to the split, and compute the weighted impurity $w_l \sum_{c=1}^{K} p_l(c)p_l(\neg c)$ of the left partition, as well as the weighted impurity $w_r \sum_{c=1}^{K} p_r(c)p_r(\neg c)$ of the right partition. Here, $p_l(c)$ and $p_r(c)$ denote the probabilities of



1 Non-robust split, for which we maintain a subtree variant
2 Unlearning request to forget user data triggers tree adjustment and recalculation of leaf weights
3 Robust split decisions never need to be revised
4 Non-robust splits might switch to a subtree variant with higher Gini gain
unlearn(user)
age < p70
education ∈ {7,9}
balance < p45
balance < p85
education ∈ {0,3}
age < p55
balance < p25
age < p60 gain 0.063
balance < p73 gain 0.051
balance < p45
age < p80
plus-=1
count-=1
minus-=1
count-=1
balance < p73 gain 0.078
tree ensemble before unlearning
tree ensemble after unlearning

Figure 2: Hedgecut learns an ensemble of randomised decision trees with randomly chosen splits on randomly chosen attributes, and maintains this ensemble under the removal ("unlearning") of a certain amount of training samples. Some trees contain non-robust splits ❶ where the split decision (based on Gini gain) might change after the removal of data ❷. Hedgecut updates leaf label statistics ❸, and maintains subtree variants for cases where split decisions change. It activates subtree variants after data removal if the model would have chosen a different split in light of the removed data ❹.

encountering an element of class c in the left or right partition. Subtracting the sum of the partition impurities from the original impurity gives the reduction which we aim to maximise.

4 APPROACH

We give an overview over the main idea behind HedgeCut in Section 4.1, before we describe in detail how we determine the robustness of split decisions in HedgeCut (Section 4.2), how we train HedgeCut ensembles (Section 4.3), and how we conduct prediction and unlearning (Sections 4.4 & 4.5).

4.1 Overview

The goal of this paper is to design a classification model that can efficiently unlearn a small fraction of its training data without retraining. We focus on tree-based ML methods, which are trained by recursively partitioning the training data, locally optimising a criterion such as Gini gain (Section 3). Variants of tree-based methods such as "Extremely Randomised Trees" (ERTs) exhibit a high amount of independence in the computations which they apply (due to the randomised choice of attributes and cut points for splitting), which makes this approach a prime candidate for enhancement with unlearning functionality.

We design our proposed model *HedgeCut* based on ERTs. The left side of Figure 2 illustrates how a HedgeCut tree ensemble looks like after training (before unlearning), and the right side gives an example of how this ensemble is maintained during unlearning. The ensemble on the left consists of decision trees which apply randomly chosen splits to the data based on which the path through the tree for a given example is chosen. Ultimately, the leafs of a tree dictate the predicted label (positive or negative class). HedgeCut differentiates between robust splits (which cannot change under the removal of small number of records) and non-robust splits ❶, which might change later on. For these splits, HedgeCut maintains so-called subtree variants that are inactive initially. In case of the unlearning of a training sample ❷, HedgeCut updates the tree ensemble to account for the removed data. Robust split decisions need not be revised and HedgeCut only updates the leaf statistics to account for the removed sample ❸. For non-robust splits, HedgeCut recomputes the Gini gain of maintained subtree variants and potentially activates a previously inactive subtree variant ❹, which might become the preferred split after the removal of a training sample.

4.2 Quantifying the Robustness of Split Decisions against Data Removal

At the core of the learning procedure of decision trees is the selection of splits to partition the data. HedgeCut (and ERTs) select a set of attributes and cut points at random, and then choose the split candidate with the highest Gini gain to further evolve the tree (line 7 in Algorithm 1).

Robustness of split decisions. In our scenario, we want the model to also handle the deletion of records after training, so a natural question to ask is how robust the split decisions are in the light of data removal. There can be cases where one split candidate exhibits a higher Gini gain at training time than another candidate, but where the decision would be reversed if we removed one more record from the data to split later on. In order to handle such cases, we need to quantify the robustness of a split decision in terms of the number of records that can be removed from the data without affecting the split decision. This *target robustness* r is inherently linked to the maximum number of removable records that we want our model to be able to handle. This number $r = \epsilon|\mathbf{D}|$ is specified by the user as a fraction ϵ of the overall size of the training data $|\mathbf{D}|$ that we want to be able to remove. If we train on 10,000 examples and we want the model to be robust against the removal of 0.1% of the data, then $r = 10$ and we are interested in determining which split decisions in our trees could change if we removed at most 10 records.

Quantification of the robustness of a split decision. In the following, we describe how we quantify the robustness of split decisions. Assume that we have selected a set of split candidates for the data $\mathbf{D}$, and now want to determine whether the best split

s^* (with the highest Gini gain) is robust against the removal of r records with respect to another split candidate t.

The following property must hold for s^* to be robust with respect to t: There can be no subset $\mathbf{P}_r$ of the partition of training samples $\mathbf{P}$ to split with r or less records, which we could remove from $\mathbf{P}$ to have t obtain a higher Gini gain $G(t, \mathbf{P} \setminus \mathbf{P}_r)$ than the gain $G(s^*, \mathbf{P} \setminus \mathbf{P}_r)$ of s^*: $\nexists\, \mathbf{P}_r \subset \mathbf{P} : |\mathbf{P}_r| \leq r \wedge G(s^*, \mathbf{P} \setminus \mathbf{P}_r) < G(t, \mathbf{P} \setminus \mathbf{P}_r)$.

We present a greedy approach to test for this robustness property in Algorithm 2. The function WEAKEN_SPLIT computes how much the removal of a single record can reduce the difference between the scores of the splits s^* and t. For both splits, we have statistics denoting how many positive and negative records they assign to their left and right partitions (from which we compute the Gini gain). The function enumerates all eight possible record removal settings (Does the record have a positive or negative label? Is it assigned to the left or the right partition of the best split? Is it assigned to the left or right partition of the split candidate?) in lines 11-13, computes the updated split statistics $\hat{s}$ and $\hat{t}$ to simulate the removal and inspects the resulting reduction of the score difference $\delta = G(\hat{s}) - G(\hat{t})$ in line 17. It determines the updated split statistics $\hat{s}$ and $\hat{t}$ for the setting which most impacted the score difference, as well as the corresponding difference δ^* (line 18). Note that some removal settings might not be applicable, e.g., when split partitions do not contain positive or negative records anymore. There can also be rare cases where several record removals result in an equal score difference δ^*. Furthermore, our greedy algorithm will not determine the correct answer if any of the counts in the split is smaller than the deletion budget r. As a consequence, we reject split candidates and repeat the candidate generation, when this is the case (see Section 4.3 for details).

Algorithm 2 Quantification of split robustness.

```
1: function IS_ROBUST($s^*$, $t$, $r$)
       Input: Best split statistics $s^*$, candidate split statistic $t$, target robustness $r$
       Output: Whether split is robust and how many removals were tested
2:     $\hat{s}^* \leftarrow s^*$
3:     $\hat{t} \leftarrow t$
4:     for $i$ in $0 \ldots r$
5:         $(\delta, \hat{s}^*, \hat{t}) \leftarrow$ WEAKEN_SPLIT($\hat{s}^*$, $\hat{t}$)
6:         if $\delta < 0$
7:             Split non-robust, decision can be reversed by removing $i$ records
8:     Split robust, decision cannot be reversed by removal of $r$ records

9: function WEAKEN_SPLIT($s$, $t$)
       Input: split statistic $s$, split statistic $t$
       Output: Minimal score difference $\delta^*$ after removal of a single record and corresponding split statistics $\hat{s}$, $\hat{t}$
10:    $\Delta \leftarrow \emptyset$
11:    for label $l$ in $\{+, -\}$
12:        for split decision $d_s$ with respect to $s$ in { left, right}
13:            for split decision $d_t$ with respect to $t$ in { left, right}
14:                if $s$ and $t$ can be updated with $(l, d_s, d_t)$
15:                    $\hat{s} \leftarrow$ Update $s$ for removal of a record with configuration $(l, d_s, d_t)$
16:                    $\hat{t} \leftarrow$ Update $t$ for removal of a record with configuration $(l, d_s, d_t)$
17:                    Add $(\hat{s}, \hat{t})$ to $\Delta$
18:    Find $\delta^* = \min_{(\hat{s}, \hat{t}) \in \Delta} G(\hat{s}) - G(\hat{t})$ minimizing the score gap between $s$ and $t$
19:    return Minimal score difference and its corresponding statistics $(\delta^*, \hat{s}, \hat{t})$
```

The function IS_ROBUST repeatedly invokes WEAKEN_SPLIT (at most r times) and updates the split statistics s and t with the most impactful removal, until one of two conditions occurs: (i) if the score difference δ is negative, then the split decision has been reversed by the record removals conducted; (ii) if we did not encounter a negative score difference δ after r record removal trials, we determine that the split is robust against the removal of r records.

Even though the Gini impurity function itself is rather simple and even concave (it is easy to show that its Hessian is negative semidefinite), the function in the focus of our greedy algorithm is difficult to analyze analytically unfortunately. This function (i) consists of the difference of the weighted combinations of Gini impurities on the modified split statistics $\hat{s}$ and $\hat{t}$ for the pair of splits; and (ii) its values live on a grid formed by the potential modifications of the split statistics, and (iii) the steps that we can take in this space are constrained by the fact that we need to make consistent modifications (e.g., the record to remove from both splits has to have the same label).

As a consequence, we validate our greedy algorithm with a randomised experiment. We pick a robustness r, and randomly generate a pair of split candidates by choosing the sample size, the total number of positive and negative records as well as the number of positive and negative records on both sides of the split at random from a uniform distribution. For each such generated split pair, we enumerate all possible 8^r modifications and inspect the corresponding scores to see if the pair is robust against the removal of r records. We compare the ground truth answer found by enumeration of the search space to the answer for the pair found by our greedy algorithm. We repeat this experiment 1,000,000 times for robustnesses in the range from 2 to 5, 100,000 times for robustnesses of 6 & 7, and 50,000 times for $r = 8$. The experiment generates a large number of both robust and non-robust splits (up to 30% for $r = 7$), and the decisions of our greedy algorithm are consistent with the correct decisions found by enumeration in all cases. This empirical test is not rigorous, but it does provide some confidence that the greedy algorithm is a sound way to test for alternative splits that have the potential to improve significantly in the presence of the deletion of training examples.

4.3 Learning HedgeCut Ensembles

In the following, we describe how HedgeCut learns an ensemble of randomised trees based on our notion of split robustness in Algorithm 3.

Searching for robust splits. In contrast to ERTs, we make Hedgecut find splits based on globally proposed percentiles of the distribution of continuous features. This is a common technique used in popular tree-based models such as XgBoost [10]. This variant uses the same proposals for split finding at all levels of the tree. We discretize continuous features into buckets according to the quantiles of their distribution in the training data. We choose this approach, as the standard formulation of ERT needs to find the minimum and maximum of the local sample subset per split (line 13 in Algorithm 1), which is hard to maintain under data removal. An additional advantage of this method is that it enables a memory-efficient feature representation for numeric features (e.g., with 8-bit integers). Our implementation operates on a discretization of the distribution of continuous features with twenty buckets (e.g., the 5th, 10th, 15th, ... percentiles). This discretisation is applied in a preprocessing step on the training data before we learn the HedgeCut ensemble.

In HedgeCut's splitting procedure, it randomly selects K non-constant features and generates split candidates based on the global proposals for them (lines 10-12). For categorical features we select a random subset of values from the domain of the feature and check for inclusion in this set; for continuous attributes, we select a random cut point from the percentile range of the feature (shown in the function RANDOM_SPLIT in line 25). HedgeCut then computes the split statistics and the corresponding Gini gain (as described in Section 3) for each split candidate. Note that some split candidates might not split the data in a local node at all (as we select cut points globally and not based on the local samples), in such cases we will simply ignore the split candidate as it provides no Gini gain.

After identifying the best split candidate from the randomly chosen set of attributes and corresponding thresholds, HedgeCut computes the robustness of the best split candidate to the remaining split candidates (line 13). Thereby, we determine whether the split decision could change later on due to potential unlearning operations. If the split is found to be non-robust, HedgeCut discards the current set of split candidates, generates new candidates and repeats the split scoring and robustness evaluation procedure (lines 9-13). This process is repeated at most B times, where B is a user-defined parameter that is typically set to a small value such as five. If a robust best split is found, we partition the data, evolve the tree, and recursively invoke the split finding procedure on the generated partitions (lines 14-17).

Algorithm 3 Hedgecut tree ensemble learning.

```
1:  function HC_ENSEMBLE($D$)
        Input: training set $D$, number of trees $M$
        Output: tree ensemble $T = \{T_1, \ldots, T_M\}$
2:    for $i = 1$ to $M$
3:      Generate tree $T_i \leftarrow$ HC_TREE($D$)
4:    return $T$

5:  function HC_TREE($D$)
        Input: training samples $D$, target robustness $r$, minimal leaf size $n_{\min}$, maximum
      number of trials to search for a robust split $B$
        Output: Node or leaf of classification tree $t$
6:    if $|D| \leq n_{\min}$ or all candidate attributes constant in $D$ or label constant in $D$
7:      return Leaf labeled with class frequencies in $D$
8:    else
9:      while no robust split found
10:       Randomly select $k$ non-constant attributes attributes $\{a_1, \ldots, a_k\}$ in $D$
11:       Generate $k$ splits $\{s_1, \ldots, s_k\}$, where $s_i \leftarrow$ RANDOM_SPLIT($a_i$)
12:       Determine best split $s^* \leftarrow \text{argmax}_{i=1..k}$ GINI_GAIN($s_i$, $D$)
13:       Check robustness of best split $s^*$ against other splits with IS_ROBUST
14:     if robust split $s^*$ found
15:       $D_l$, $D_r \leftarrow$ split $D$ according to $s^*$
16:       Build subtrees $t_l \leftarrow$ HC_TREE($D_l$) and $t_r \leftarrow$ HC_TREE($D_r$)
17:       return Node applying split $s^*$ with subtrees $t_l$ and $t_r$
18:     else if maximum number of tries $B$ exceeded
19:       Initialize subtree variants $\Gamma \leftarrow \emptyset$
20:       for non-robust split candidate $s_i$
21:         $D_{il}$, $D_{ir} \leftarrow$ split $D$ according to $s_i$
22:         Build subtrees $t_{il} \leftarrow$ HC_TREE($D_{il}$) and $t_{ir} \leftarrow$ HC_TREE($D_{ir}$)
23:         Add subtree variant $(s_i, t_{il}, t_{ir})$ to $\Gamma$
24:       return Maintenance node for the subtree variants $\Gamma$

25: function RANDOM_SPLIT($a$)
26:   if $a$ is numerical
27:     $a_c \leftarrow$ random cut point from the quantile range of $a$
28:     return $[a < a_c]$
29:   if $a$ is categorical
30:     $A_s \leftarrow$ random subset of values from domain of $a$
31:     return $[a \in A_s]$
```

Maintenance nodes for subtree variants of non-robust splits. If we still encounter non-robust splits after B trials, we fall back to growing all *subtree variants* for the split candidates that could become viable if records would be removed. We recursively grow subtrees for all split candidates as if each of them would have been chosen as the best split, and create a special *maintenance node* which contains the statistics for each split candidate and its corresponding subtrees (lines 18-24). This maintenance node maintains the current Gini gain for each split candidate and can thereby always delegate operations (like prediction) to the current best split. Furthermore, we need to maintain the subtree variants under data removal, as will be discussed in detail in the following sections.

4.4 Predicting with a HedgeCut Ensemble

Predicting the label for an unseen record d with a Hedgecut ensemble T works analogous to ERTs. We ask every individual tree T_i of the ensemble to predict a label for the record d and return the majority label as the final prediction. The prediction procedure for each tree is initially invoked with the root node of each tree as argument. We determine whether the current tree element is a leaf node or not. In the case of a leaf node, we simply return the prediction derived from the leaf statistics, by comparing the number of positive records assigned to the leaf to the overall number of records in the leaf. If the current node is not a leaf node, it must be a split node and we evaluate the corresponding split criterion to determine whether the record to predict for will be assigned to the left or right subtree of t. Afterwards, we recursively continue with the determined subtree. In the case of a maintenance split node, we visit the current best alternative subtree for the split criterion evaluation and use its child nodes for the recursive invocation of our prediction function.

4.5 Unlearning a Training Example

We finally discuss the procedure to unlearn a record d from the training set which is shown in Algorithm 4. In general, the algorithm is similar to the previously discussed prediction, where we traverse each tree for the sample, with the difference that we update leafs and non-robust splits in this case. We dispatch the unlearning of a record d for a tree ensemble T to each individual tree T_i of the ensemble by invoking the function HC_UNLEARN for each root node of the particular tree.

We now traverse each tree in HC_UNLEARN. When we encounter a leaf node, we update its statistics with respect to the record to remove d as shown in lines 2-5. We decrement the number of samples assigned to the leaf, as well as the number of positive samples in case d has a positive label. If we encounter a split node instead, we distinguish between two cases: (i) if the split node is robust, we simply apply the split criterion to decide whether to assign d to the left or right partition of the split and recursively invoke HC_UNLEARN with the corresponding subtree (lines 7-9); (ii) If we encounter a non-robust split, we invoke the HC_UNLEARN procedure for all its alternative subtrees to propagate the removal of d to all tree variants. Afterwards, we update the split statistics of all alternative subtrees with respect to the removal of d, and update the current best alternative subtree for the non-robust split node (lines 11-14).

Algorithm 4 Unlearning a training sample.

```
1: function HC_UNLEARN_FROM_TREE($t$, $d$)
     Input: classification tree node $t$, training sample $d$ to unlearn
2:   if $t$ is a leaf node
3:     Reduce count of samples assigned to leaf $t$ by 1
4:     if $d$ belongs to the positive class
5:       Reduce count of positive samples assigned to leaf $t$ by 1
6:   else
7:     if $t$ is a robust split node
8:       $t_d \leftarrow$ subtree to which $t$ assigns $d$
9:       HC_UNLEARN($t_d$, $d$)
10:    else
11:      for subtree variant $t_a$ of non-robust split node $t$
12:        HC_UNLEARN($t_a$, $d$)
13:        Update Gini gain of $t_a$ based on removal of $d$
14:      Re-score subtree variants of $t$
```

5 IMPLEMENTATION

We conduct a multi-threaded single machine implementation of HedgeCut in Rust 1.45, and discuss our design decisions and performance optimisations.

Exploiting parallelism. For a start, we leverage the fact that each tree T_i in a HedgeCut ensemble is completely independent of the other trees, which allows us to execute both training and inference in an embarassingly parallel manner. We learn trees in parallel on copies of the input data, and we collect the predictions from the trees at inference time in parallel as well. The unlearning procedure can also be run in parallel on all the trees. We leverage fork-join parallelism with a thread pool and work stealing, provided by the `rayon`[3] crate.

Vectorised computation of Gini gain. The main computational effort during tree learning consists of computing the Gini gain of different split candidates to decide which one to choose to evolve a tree (line 12 in Algorithm 3). In the following, we describe how we apply techniques from vectorised query processing [4, 22, 41] to accelerate these computations.

As discussed in Section 3, the Gini gain for the general case of K different classes is measured as follows: $\sum_{c=1}^{K} p(c)p(\neg c) - \left[w_l \sum_{c=1}^{K} p_l(c)p_l(\neg c) + w_r \sum_{c=1}^{K} p_r(c)p_r(\neg c)\right]$, where $\sum_{c=1}^{K} p(c)p(\neg c)$ denotes the impurity before conducting a split, with $p(c)$ being the probability of picking an element of class c at random from the data, and $p(\neg c)$ the probability of picking an element from a different class. When evaluating a split, we partition the data into a left and right part according to the split, and compute the weighted impurity $w_l \sum_{c=1}^{K} p_l(c)p_l(\neg c)$ of the left partition, as well as the weighted impurity $w_r \sum_{c=1}^{K} p_r(c)p_r(\neg c)$ of the right partition. Here, $p_l(c)$ and $p_r(c)$ denote the probabilities of encountering an element of class c in the left or right partition. The weights w_l and w_r denote the probabilities of a sample ending in the left or right partition, respectively. Subtracting the sum of the partition impurities from the original impurity gives the reduction which we aim to maximize. When evaluating different split candidates, we can ignore the impurity of the data before splitting (which is independent of the choice of how to split the data), and only compute the impurity reduction of the split $w_l \sum_{i=1}^{K} p_l(i)p_l(\neg i) + w_r \sum_{i=1}^{K} p_r(i)p_r(\neg i)$, which can be simplified to $w_l\,[2p_l(\oplus)\,p_l(\ominus)] + w_r[2\,p_r(\oplus)\,p_r(\ominus)]$ in the case of binary classification with $K = \{\oplus, \ominus\}$, on which we focus in this paper. We compute this quantity from four different counts that we need to obtain from the data: the overall number of samples n, the number of samples in the left partition k_l, as well as the number of the positive samples in the left partition $k_{l\oplus}$, and the number of positive samples in the right partition $k_{r\oplus}$. We compute the weights as the fraction of samples that are assigned to the left and right partition $w_l = k_l/n$ and $w_r = (n - k_l)/n$, and estimate the probabilities of encountering a positive samples in a partition as $p_l(\oplus) = k_{l\oplus}/k_l$ and $p_r(\oplus) = k_{r\oplus}/(n - k_l)$. Therefore, the computation of the Gini gain reduces to scanning the data, and counting the number of records which satisfy the following three predicates: (i) Does the record have a positive label? (ii) Does the record have a positive label and the split assigns it to its left partition? (iii) Does the record have a positive label and the split assigns it to its right partition?

We design two vectorised counting approaches[4] (one for discretised continuous features and one for categorical features) based on SIMD ("Single-Instruction-Multiple-Data") instructions, which compute the desired quantities n_l, n_{l+} and n_{r+} from a data sample of length n and a split to evaluate.

Discretised continuous features. We leverage 8-bit integers (`u8` in Rust) to represent the discretized quantiles of the feature values as well as the cut off value for the split comparison. Our main processing loop operates on a batch of 16 records in parallel and first loads their feature values as well as label into SIMD registers. Next, we compare the feature values to the cut offs via `_mm_cmplt_epi8` where the boolean outcome denotes whether the record is assigned to the left or right partition of the split. We apply a boolean AND (and a NAND respective) to the result of the comparison and the label vector to get the boolean indicators for the conjunction of the test whether a record is positive and assigned to the left (and respective to the right) partition of the split. We finally load the resulting bit indicator into an integer register via `_mm_movemask_epi8` and count the number of set ones in the bit vector using Rust's `count_ones()` function which will make use of the CPU's POPCNT instruction for counting if it is available.

Categorical attributes. Computing the counts required for determining the Gini gain on categorical attributes is more difficult, as we have to test for set inclusion instead of just comparing to a threshold. Our vectorised implementation handles categorical attributes with a cardinality of up to 32, and we adaptively use non-SIMD code for categorical attributes with a higher cardinality. For categorical attributes, we test whether a particular record is assigned to the left (or right) partition of a split by checking whether the attribute's value is contained in a randomly chosen subset of the domain of the attribute (the bits set in the `subset` variable). Our vectorised implementation operates on a batch of four 32-bit values.

Further optimisations. We additionally remove branches in the non-vectorised Gini gain code via predication (replacing conditional statements with additions of boolean variables). Analogous to scikit-learn, we partition the training data in-place after deciding on a particular split, and recursively invoke the split finding procedure with pointers (references to mutable slices in Rust) to the

[3]https://crates.io/crates/rayon

[4]https://github.com/schelterlabs/hedgecut/blob/master/src/scan.rs

dataset	#users	#num	#cat	#data points
income	32,560	4	8	390K
heart disease	70,000	5	6	770K
credit	150,000	8	-	1.2M
recidivism	7,214	4	6	110K
purchase data	12,330	10	7	210K

Table 1: Number of users, numerical features (#num), categorical features (#cat) and sizes of the datasets.

resulting subpartitions. We apply a copy-on-write data structure using Rust's `std::borrow::Cow` to propagate the constant attribute identifiers during the learning procedure. This allows us to avoid heap allocations in cases where the set of constant attributes does not change.

6 EVALUATION

We experimentally evaluate HedgeCut to showcase that our approach can unlearn training samples in less than a millisecond (Section 6.2.1), and is able to perform unlearning as part of high throughput serving workloads (Section 6.2.2) with the same accuracy as a retrained model (Section 6.3.1). Furthermore, we compare HedgeCut to popular tree-based ML models like Random Forests in terms of accuracy and training cost (Sections 6.3.2 & 6.4.1), measure the benefits of our vectorised Gini gain computation (Section 6.4.2), and explore the impact of certain parameters of HedgeCut (Section 6.5).

6.1 Setup

We describe the data, baseline models and evaluation metrics used for our experiments.

Datasets. We evaluate HedgeCut on binary classification tasks for five publicly available datasets from various domains. We aim for these datasets to be representative of data for which GDPR deletion requests might be issued, and therefore select privacy-critical datasets only, where each row represents sensitive personal data of an individual. Table 1 lists the summary statistics of these datasets.

Adult income. This dataset[5] contains 390K data points in 32,560 records of demographic and financial data, with four numerical and eight categorical attributes, and the target variable denotes whether a person earns more than 50,000 dollars per year or not.

Medical records about heart disease. This dataset[6] contains 770K data points in 70,000 patient records comprised of five numerical and six categorical measurements with respect to cardiovascular diseases, and the target variable denotes the presence of a heart disease.

Credit information. This dataset[7] contains 1.2M data points of financial information in eight numerical attributes for 150,000 people and the target variable denotes whether a person has experienced financial distress.

[5]https://archive.ics.uci.edu/ml/datasets/Adult
[6]https://www.kaggle.com/sulianova/cardiovascular-disease-dataset
[7]https://www.kaggle.com/c/GiveMeSomeCredit

Recidivism. This dataset[8] contains 110K data points in four numerical and 6 categorical attributes of demographic data and data about prior engagements with law enforcement and the judicial system for 7,214 individuals arrested in Broward County, Florida, in 2013 and 2014. The target variable denotes whether a person was charged with new crimes over the next two years.

Online purchase behavior data. This dataset[9] contains 210K data points in ten numerical and seven categorical attributes about browsing behavior in 12,330 individual sessions of an online shop, and the target variable denotes whether the session ended in a purchase or not.

Ethical note: We are aware of the controversies and dangers of applying automated decision making via ML to real-world problems like judicial decisions [36] and want to stress that our experiments should not be understood as an endorsement of such approaches. We select these datasets and tasks only with the goal to showcase HedgeCut's applicability to remove personal sensitive data from models trained on real-world data in an academic context.

Baseline algorithms. We compare HedgeCut against several implementations of common tree-based classification methods from the popular library scikit-learn [28], using version 0.22.1: *Extremely Randomised Trees*, the classic ERT algorithm from [17], discussed in Section 3, which is the basis for HedgeCut; the well-known *Random Forest* [6] algorithm, which trains an ensemble of decision trees on samples of the data with bootstrap aggregation and random selection of features; and a single *Decision Tree* learned with an optimised version of the CART algorithm [7]. Note that even though scikit-learn is a Python library, the tree-based algorithms are implemented in highly optimised Cython code, which is translated into C code and compiled to machine code.

If not reported otherwise, we configure all algorithms with the default hyperparameter settings recommended in the documentation of scikit-learn.[10]. For the comparison between ERT and HedgeCut, we make sure that both algorithms run with identical settings taken from the original ERT paper [17]: learning an ensemble of 100 trees with a minimal leaf size of two, picking a random fraction of attributes for split testing proportional to the square root of the total number of attributes, and we make both use the Gini gain for split selection. The random forest classifier operates with similar default parameters (training an ensemble of 100 trees with Gini gain as split selection criterion).

Metrics. We experiment using our single machine implementation in Rust 1.45 on a machine with an i7-8650U CPU @ 1.90GHz with four cores and 16GB of RAM, running Ubuntu Linux 16.04. We report the runtime for all algorithms (both for training and prediction), in which we do not include the time to load and parse the input datasets. We report the accuracy of the resulting classifiers on a randomly chosen, held-out test set comprised of 20% of the data points. For all metrics, we report both the mean and standard deviation from a series of experiments.

[8]https://github.com/propublica/compas-analysis/
[9]https://archive.ics.uci.edu/ml/datasets/Online+Shoppers+Purchasing+Intention+Dataset
[10]https://scikit-learn.org/stable/modules/ensemble.html#forest

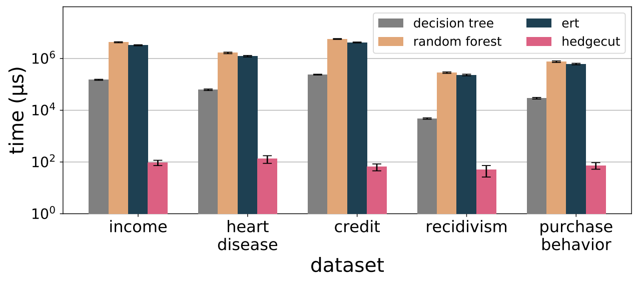


Figure 3: Time to unlearn a training example with HedgeCut compared to retraining the baselines models without this example, plotted in microseconds on a logarithmic scale. HedgeCut conducts unlearning in 100 μs, while the baselines require more than three orders of magnitude more time to retrain in the majority of cases.

Reproducibility. We use publicly available datasets and make both our experimentation code available at *https://github.com/schelterlabs/hedgecut*.

6.2 Unlearning

6.2.1 Low-latency unlearning. We design our first experiment with respect to the main goal of our work, which is to enable low-latency unlearning of training samples from a trained model. We train a model for HedgeCut and the baselines for all five datasets. Next, we choose a random training example to remove. We unlearn this training example with HedgeCut, and retrain the baseline algorithms on the training set without the training example (as they do not possess unlearning capabilities). We measure the time for unlearning and retraining and plot the mean and standard deviation of the runtimes. We repeat this procedure for 0.1% of the training examples of each datasets and repeat the whole process on three different train/test splits of the datasets.

Figure 3 illustrates the resulting runtimes in microseconds (plotted on a logarithmic scale) for the different algorithms. HedgeCut is able to consistently unlearn a single training example in around 100 μs for all datasets, while the retraining of the baseline models takes more than three orders of magnitude longer in the majority of cases. Note that the retraining will even be longer in real-world setups as we did not include the data loading and parsing time in our measurements. This experiments shows that HedgeCut can unlearn training examples with a low latency well beyond a millisecond, which makes it usable for usage in ML serving systems, which typically have an SLA of answering prediction requests in several dozen milliseconds (including network round-trip time).

6.2.2 Unlearning under prediction load. Our next experiment focuses on the impact that unlearning requests have on the throughput of prediction workloads. We mimic a deployed model in an ML serving system under heavy prediction load by having each model predict for items from the test set 100,000 times, and we measure how many prediction requests HedgeCut can answer per second. We repeat this procedure, but "mix-in" unlearning requests for 0.1% of the training examples, by replacing randomly selected prediction requests with unlearning requests. We repeat this experiment ten times for each dataset, and plot the resulting mean throughput (and the standard deviation) in Table 2. We observe that HedgeCut can generate well over 10,000 predictions per second with up to 36,667 for the relatively small `recidivism` dataset, which is again an indication that our model is a suitable candidate for deployment in demanding ML serving setups. We observe no significant difference between the throughput with and without unlearning requests. We confirm this observation with a two-sample Kolmogorov-Smirnov test on the numbers per dataset, which indicates no distributional difference as well.

dataset	predictions/sec	predictions/sec with unlearning
income	20,127 (±1,253)	20,041 (±1,227)
heart	13,192 (±512)	13,238 (±759)
credit	17,030 (±390)	16,891 (±731)
recidivism	36,667 (±3,032)	37,349 (±2,656)
purchase data	26,120 (±1,795)	26,435 (±1,738)

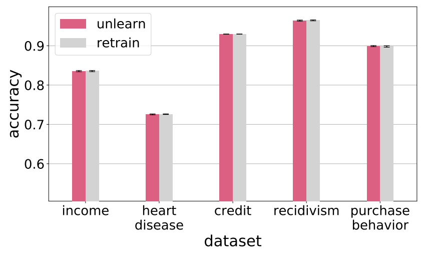
Table 2: Prediction throughput for HedgeCut per dataset without and with mixed-in unlearning requests. Unlearning requests do not decrease its overall prediction throughput.

6.3 Predictive Performance

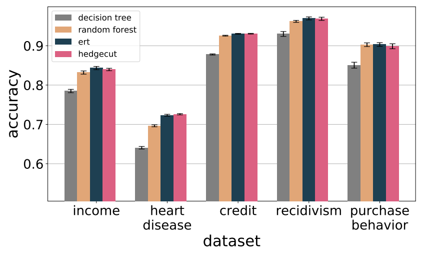
6.3.1 Predictive performance under unlearning. Our next experiment is meant to validate the predictive performance of HedgeCut after unlearning the maximum number of training examples. We repeatedly observed that unlearning even a single training example can change some of the predictions of a classifier, therefore we want to validate that unlearning and retraining lead to identical performance. We train HedgeCut on an 80% training set randomly chosen from each dataset, have it unlearn 0.1% of the training samples which we choose at random, and measure the accuracy of the resulting model on the held-out test data afterwards. We compare this to the accuracy achieved by a second HedgeCut model (without unlearning) which we retrain from scratch on the training data without the unlearned samples. We repeat this procedure twenty-five times for all five datasets and plot the resulting mean accuracies (as well as their standard deviations) in Figure 4(a).

We do not observe a difference in the predictive performance of the model that unlearned the training samples compared to the model that was re-trained without them. We compute the absolute mean difference between these two variants, which is always less than 0.0004, and conducted a two-sample Kolmogorov-Smirnov test between the accuracies for each dataset, which finds no distributional difference. We confirm that the HedgeCut model with unlearned data points exhibits the same predictive performance as the retrained model.

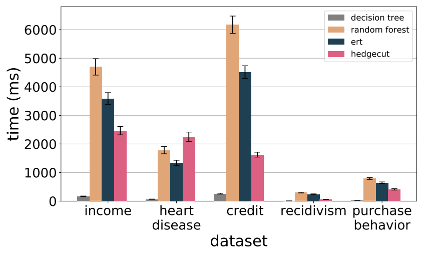
6.3.2 Predictive performance of HedgeCut in comparison to the baselines. Next, we measure the out-of-the box predictive performance of HedgeCut. We compare HedgeCut and our baselines with their default hyperparameters on our five datasets. The goal of this experiment is to showcase that HedgeCut's out-of-the-box accuracy is on par with ERTs and commonly used ensemble-based tree methods such as Random Forests [6]. Note that the general



(a) Predictive performance of a HedgeCut model that unlearned 0.1% of training samples compared to a HedgeCut model that was retrained without these training samples. The model with unlearning exhibits the same accuracy as the retrained model.



(b) Predictive performance of HedgeCut compared to the baselines for our five datasets. HedgeCut provides high-accuracy results and performs on par with Random Forest in the majority of cases.



(c) Comparison of the training time of HedgeCut and the baseline methods. Among the ensemble-based methods, which provide high accuracy, our HedgeCut implementation outperforms both the random forest and ERT in four out of five datasets.

Figure 4: Evaluation of the predictive performance and training time of HedgeCut. HedgeCut provides the same accuracy after unlearning user data compared to a retrained model (Figure 4(a)). Furthermore, HedgeCut's accuracy on our five datasets is on par with the accuracy of extremely randomised trees and a random forest model (Figure 4(b)), while HedgeCut exhibits a lower training time than these methods in four out of five cases (Figure 4(c)).

predictive performance of ERTs and Random Forests is well established, we refer readers to the extensive study in the original ERT paper [17] for details. We generate a random train/test split of the data, train each model on the train data and measure the accuracy of the model on the unseen test data. We repeat this procedure ten times and plot the resulting mean accuracy (and the corresponding standard deviation) in Figure 4(b). The results exhibit the following pattern, which is consistent over all datasets: The three ensemble-based methods (Random Forest, ERT, HedgeCut) provide a higher accuracy than the single decision tree. ERT and HedgeCut give the best performance, closely followed by Random Forest. These results confirm that HedgeCut provides on-par performance with ERT and Random Forest, and might be used as a drop-in replacement in scenarios where these classifiers are deployed.

6.4 Training Cost

6.4.1 Training time in comparison to the baselines. Next, we compare the training time of HedgeCut to the baselines, with the goal of showing that HedgeCut is competitive with respect to its training time, even though it has to conduct additional work like robustness scoring. This experiment also serves a validation of the careful implementation and optimisation decisions described in Section 5. We make HedgeCut, ERT and Random Forest all train an ensemble of 100 trees with Gini gain as splitting criterion. We tried to enable multi-threading for both ERT and Random Forest, by setting their parameter `n_jobs` to -1 (allowing them to use all cores), however this resulted in a drastic increase of their runtimes. We therefore leverage their faster single-threaded variants for comparison. We train models for each dataset, and measure the training time. We repeat this procedure ten times and plot the resulting mean training times (as well as the corresponding standard deviations) in Figure 4(c).

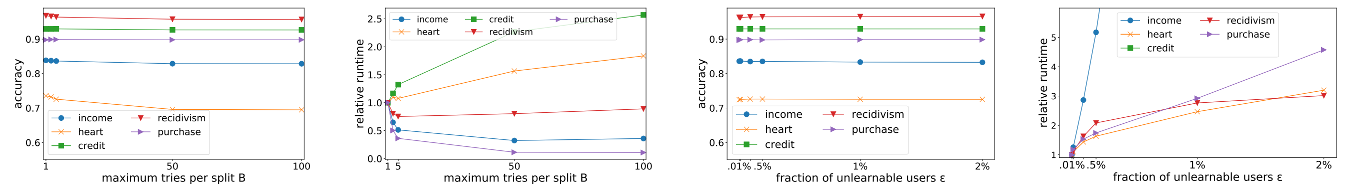
We make several observations. First, we find that the decision tree (which only learns a single tree instead of an ensemble) provides the lowest runtime, and outperforms the ensemble-based methods in all cases. However, this method does not provide a competitive predictive performance as we have seen in the previous experiment in Section 6.3.2. When comparing the ensemble-based methods, we find that both the ERT and HedgeCut outperform Random Forest in all cases, while all models can be trained in a couple of seconds for our datasets. In four out of five datasets, HedgeCut outperforms ERT, even though it has to conduct additional work for robustness searches. We conclude that our HedgeCut implementation provides a competitive runtime compared to the widely-used implementations of popular ensemble methods like scikit-learn's Random Forest. We note that the baseline implementations could potentially also be benefit from our parallelisation and vectorisation techniques.

We would also like to stress that the short training times observed here are a result of the lab conditions of this experiment, and are not representative of the time and the cost to retrain and redeploy a model in a real world deployment. Redeployment not only requires model retraining, but also data preproccesing, data validation steps, and complex deployment operations [27, 29]. Our approach allows us to avoid these costs for GDPR deletion requests, as we can directly update the deployed model in place.

6.4.2 Benefits of Vectorisation. In this experiment, we measure the performance improvements gained by our vectorised Gini gain implementations. We run the following two micro-benchmarks: We compare the non-SIMD code to the non-SIMD code with branches removed and our vectorised SIMD implementations from Section 5. In addition, we reimplement an existing SIMD-based Gini gain computation technique for decision tree learning from mlpack for comparison, which only leverages instruction-level parallelism for summation of the label counts per split.[11]

We execute the Gini gain computation for continuous features on 96,214 records from the `credit` dataset, where we test for a cut off in the `past_due` attribute. The non-optimised code taskes 849 μs, removing branching reduces this to 588μs (-31%), and the usage of SIMD instructions in our vectorised implementation decreases

[11]https://www.mlpack.org/doc/mlpack-3.1.1/doxygen/gini__gain_8hpp_source.html



(a) Impact of the maximum number of tries per split B on the accuracy of HedgeCut. Low values ($B < 10$) result in a higher accuracy, and the accuracy stays constantly lower for larger values, where HedgeCut will find more robust but lower quality splits.

(b) Impact of the maximum number of tries per split B on the training time of HedgeCut. The effect is dataset dependent with a growing runtime for larger values (due to longer robustness searches). There is a sweet spot at $B = 5$ for all datasets.

(c) Impact of the fraction of unlearnable users ϵ on the accuracy of HedgeCut. As expected, the accuracy is not affected by the ϵ parameter.

(d) Impact of the fraction of unlearnable users ϵ on the training time of HedgeCut. Larger fractions increase the runtime, as a higher number of subtree variants have to be trained. The impact is dataset dependent and is low for the range of ϵ up to 0.1% (unlearning every 1000th user).

Figure 5: Sensitivity of the accuracy and runtime of Hedgecut with respect to the maximum number of tries per split and the robustness of the resulting model. The accuracy is largely unaffected which makes these parameters easy to tune for runtime. Over all datasets, setting the maximum number of tries per split B to 5 and the fraction of unlearnable users ϵ to 0.1% (every thousandth user) constitutes a sweet spot with high accuracy and managable runtime.

the runtime further to 420μs (-50.5%). The mlpack variant only slightly improves upon the non-optimised code with a runtime of 847μs. We execute the gini gain computation for categorical attributes on 9,863 records from the purchase behavior purchase dataset, where we test for set membership in the `browser_type` attribute. The non-optimised code taskes 82 μs, removing branching reduces this to 58μs (-29%), and the vectorised version decreases the runtime further to 44μs (-46%). Again, the mlpack variant only marginally improves the runtime over the non-optimised code with 75μs. In summary we find that our vectorised implementations double the performance of the scan-based Gini gain computations. We attribute the low performance of the mlpack variant to the fact that it has been designed for classical decision tree learning, where the overall best split for a sample needs to be determined (and the summations of the label counts can become a bottleneck). ERTs on the other hand only evaluate a small number of randomly chosen split candidates, and the main computational bottleneck are the threshold comparisons which we accelerate with SIMD (Section 5).

6.5 Parameter Sensitivity of HedgeCut

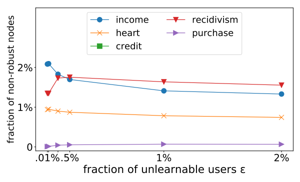
Finally, we evaluate the sensitivity of the accuracy and runtime of HedgeCut to different parameter settings. We start with the maximum number of tries B to search for robust splits per node (line 18 in Algorithm 3). We vary this parameter between 1 and 100, and train and evaluate models on all datasets, while recording the accuracy and training time. We repeat this procedure ten times per dataset. Figure 5(a) plots the resulting mean accuracies per dataset. Small values ($B < 10$) result in a higher accuracy, and the accuracy stays constantly lower for larger values, which we attribute to the fact that HedgeCut will find more robust but lower quality splits when allowed a high number of trials. We illustrate the impact of the maximum number of tries per split B on the training time of HedgeCut in Figure 5(b) (relative to the training time for $B = 1$). The effect is dataset dependent with an initial drop of the runtime for small positive values of B (which allows us to avoid to train many subtree variants), and a growing runtime for larger values (due to longer robustness searches). We observe a sweet spot at $B = 5$ with respect to accuracy and training time for all datasets.

Next, we investigate the impact of the parameter ϵ specifying the fraction of unlearnable users in the model. We vary this parameter between 0.01% (removal of every ten-thousandth user, the fraction estimated by industry practitioners) and 2% (deletion request by every 50th user, an unrealistic value), and again train and evaluate models on all datasets, while recording the accuracy and training time. Figure 5(c) shows that the accuracy of the produced models is unaffected by this parameter, which is expected as this parameter only results in the training of more subtree variants in the ensemble.

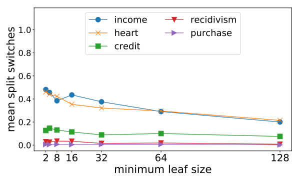
Figure 5(d) illustrates the impact of the fraction of unlearnable users ϵ on the training time of HedgeCut (relative to the training time for $\epsilon = 0.01\%$). Larger fractions increase the runtime, as a higher number of subtree variants have to be trained. The impact is dataset dependent (e.g., only a slight increase for the heart disease dataset but a rather drastic increase for the income data). However, the impact of this parameter is low in the range of 0.01% to 0.1% (unlearning every thousandth user), which corresponds to the fraction of unlearnable users estimated by industry practitioners.

We additionally report the fraction of non-robust nodes in the trees in Figure 6(a). The ratio is dataset dependent and relatively low (smaller than 2% in the majority of cases). For larger values of ϵ, we observe a corresponding increase in the overall number of nodes in the trees however. This increase is below two for $\epsilon \leq 0.1\%$ in all cases, and stays below four for higher values of ϵ for all datasets (with the exception of the income dataset, where observe a steeper growth, which explains its steep runtime increase in Figure 5(b)). We attribute this increase in the tree size to the generation of more less discriminative (but still robust) splits during the repetitions in our split search procedure 4.3, dictated by the maximum number of tries per split B. In summary, we find that Hedgecut compensates for larger values of ϵ by generating larger trees with a relatively constant fraction of non-robust nodes.

We run an additional experiment, where we inspect the impact of the minimal leaf size per tree on the number of split switches (changes of alternative subtrees for non-robust nodes) at prediction



(a) Impact of the fraction of unlearnable users ϵ on the fraction of non-robust nodes. The impact is low for the range of ϵ up to 0.1%.



(b) Impact of the leaf size on the number of split switches per tree during unlearning. The number of switches is low (less than one per tree on average) and decreases with larger leaf sizes.

Figure 6: Sensitivity of the ratio of robust to non-robust nodes and the number of required split switches at prediction time to various parameters in Hedgecut.

time. For that, we train an ensemble of 100 trees with an increasing minimal leaf size of 2, 4, 8, 16, 32, 64 and 128, and remove 0.1% of users from the trained model afterwards (chosen at random). We repeat this experiment 20 times and report the mean number of split switches per tree for each dataset in Figure 6(b). We find that the number of switches is low (less than one per tree on average) and decreases with larger leaf sizes, which is expected as trees with larger leaf sizes contains less intermediate nodes.

In summary, we find over all datasets that setting the maximum number of tries per split B to 5 and the fraction of unlearnable users ϵ to 0.1% (every thousandth user) constitutes a sweet spot with high accuracy and manageable runtime. In practice, data scientists could start from this sweet spot and run some of the outlined parameter sensitivity experiments to determine well-working parameters for their data.

7 RELATED WORK

Ensuring that data processing technology adheres to legal and ethical standards in a fair and transparent manner [36] is an important research direction, which attracts attention from law makers and governments. The direction of efficient data erasure [3] is addressed by redesigning fundamental building blocks of data processing systems [31]).

The machine learning community has pioneered work on removing data from models under the umbrella of "decremental learning" for support vector machines [8, 9, 21]. In contrast to our work, these gradient based approaches always restart the training and therefore need to reaccess the training data for updating the model. This characteristic introduces operational complexity as model training and serving (and the storage of training data) are typically handled in different systems and infrastructures [1, 27]. The acceleration of such retraining is in the focus of recent work [26, 39]. We refer to the discussion in [32] for details on why gradient descent requires reaccessing the training data, and is therefore not applicable in our problem setting.

Ginart et. al. [18] explore a problem setting similar to ours for stochastic algorithms, in particular for variants of k-Means clustering. We build upon our initial work [32], which proposes efficient decremental update techniques for decremental learning. This work however only targets very basic ML models, does not account for the fact that only a small fraction of the users will typically issue deletion requests, and some of the discussed approaches (like the nearest neighbors classifier) still need to memorize the training data.

An orthogonal technique to protect the privacy of user data in machine learning use cases is differential privacy [12]. However, it is very difficult to design differentially private algorithms (even for experts [11]), and this approach requires a decision on the limit of the acceptable privacy loss in practice.

Decision trees [7] and their ensemble-based variants [6, 10, 17] belong to the most widely used models for supervised machine learning, available in popular ML libraries such as scikit-learn [28]. The instability of the splitting decisions of decision trees under data changes is a known phenomenon [13], and sometimes addressed by proposing conjunctive predicates [23], which is difficult to implement efficiently. While some incremental tree learning approaches are known, they require recursive reorganisation of the tree, which is hard to parallelize and scale-out [37] and additionally need to memorize the training data [20]. There has also been research on on accelerating the runtime of decision trees at the inference stage with task- and instruction-level parallelism for an already trained model [2, 40], which is complementary to our methods for accelerating the training of the tree models with vectorisation techniques.

8 CONCLUSION & FUTURE WORK

We introduced the problem of low-latency machine unlearning, which is concerned with maintaining a deployed ML model in place under the removal of a small fraction of training samples without retraining. We proposed a classification model called HedgeCut for this setting, which is based on an ensemble of randomised decision trees. We detailed how to efficiently implement HedgeCut and conducted an experimental evaluation on five privacy-sensitive datasets, where we find that HedgeCut can unlearn training samples with a latency of less than a millisecond and can answer up to 36,000 prediction requests per second. We also found the training time and predictive accuracy of HedgeCut to be similar to widely used implementations of tree-based ML models such as Random Forests.

In future work, we plan to extend HedgeCut to support regression scenarios. Furthermore, we aim to investigate how to further reduce latency during prediction and unlearning by switching to a different data structure than a list of hashmaps after training for the tree ensemble, and by leveraging instruction-level parallelism [40]. Another important direction for future work is to explore whether HedgeCut can also support online learning of the decision trees, in addition to maintaining them under data deletion. Furthermore, we will explore an implementation of HedgeCut in Differential Dataflow [24] to integrate it into complex data processing pipelines.

REFERENCES

[1] Pierre Andrews, Aditya Kalro, and Alexander Sidorov. 2016. Productionizing machine learning pipelines at scale. *ML Systems workshop at ICML* (2016).

[2] Nima Asadi, Jimmy Lin, and Arjen P De Vries. 2013. Runtime optimizations for tree-based machine learning models. *IEEE TKDE* 26, 9 (2013), 2281–2292.

[3] Manos Athanassoulis. 2020. Let's talk about deletes! https://blogs.bu.edu/mathan/2020/06/29/lets-talk-about-deletes/.

[4] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. *CIDR*, Vol. 5. 225–237.

[5] Joos-Hendrik Böse, Valentin Flunkert, Jan Gasthaus. 2017. Probabilistic demand forecasting at scale. *VLDB* 10, 12 (2017), 1694–1705.

[6] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[7] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. 1984. *Classification and regression trees.* CRC press.

[8] Yinzhi Cao and Junfeng Yang. 2015. Towards making systems forget with machine unlearning. *IEEE Symposium on Security and Privacy* (2015), 463–480.

[9] Gert Cauwenberghs and Tomaso Poggio. 2001. Incremental and decremental support vector machine learning. *NeurIPS* (2001), 409–415.

[10] Tianqi Chen 2016. Xgboost: A scalable tree boosting system. *KDD.*

[11] Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. 2018. Detecting violations of differential privacy. *CCS* (2018), 475–489.

[12] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. *ToC* (2006).

[13] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The elements of statistical learning.* Vol. 1. Springer.

[14] GDPR.eu. Article 17: Right to be forgotten. https://gdpr.eu/article-17-right-to-be-forgotten.

[15] GDPR.eu. Recital 74: Responsibility and liability of the controller. https://gdpr.eu/recital-74-responsibility-and-liability-of-the-controller/.

[16] GDPR.eu. Recital 75: Risks to the rights and freedoms of natural persons. https://gdpr.eu/recital-75-risks-to-the-rights-and-freedoms-of-natural-persons/.

[17] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine learning* 63, 1 (2006), 3–42.

[18] Antonio Ginart, Melody Y. Guan, Gregory Valiant, and James Zou. 2019. Making AI Forget You: Data Deletion in Machine Learning. *NeurIPS* (2019).

[19] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering* 18.

[20] Geoff Hulten, Laurie Spencer, and Pedro Domingos. 2001. Mining time-changing data streams. *KDD.* 97–106.

[21] Masayuki Karasuyama and Ichiro Takeuchi. 2009. Multiple Incremental Decremental Learning of Support Vector Machines. *NeurIPS* (2009), 907–915.

[22] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *VLDB* 11, 13 (2018), 2209–2222.

[23] Ruey-Hsia Li and Geneva G Belford. 2002. Instability of decision tree classification algorithms. *KDD.* 570–575.

[24] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. *CIDR* (2013).

[25] Xiangrui Meng, Joseph Bradley, , et al. 2016. Mllib: Machine learning in apache spark. *JMLR* 17, 1 (2016), 1235–1241.

[26] Seth Neel, Aaron Roth, and Saeed Sharifi-Malvajerdi. 2020. Descent-to-Delete: Gradient-Based Methods for Machine Unlearning. arXiv:stat.ML/2007.02923

[27] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *ML Systems workshop at NeurIPS*

[28] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, et al. 2011. Scikit-learn: Machine learning in Python. *JMLR* 12 (2011), 2825–2830.

[29] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data lifecycle challenges in production machine learning: a survey. *SIGMOD Record* 47, 2 (2018), 17–28.

[30] Laura Elena Raileanu and Kilian Stoffel. 2004. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence* 41, 1 (2004), 77–93.

[31] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. *SIGMOD.*

[32] Sebastian Schelter. 2020. "Amnesia"–A Selection of Machine Learning Models That Can Forget User Data Very Fast. *CIDR* (2020).

[33] Sebastian Schelter, Felix Biessmann, Tim Januschowski,et al. 2018. On Challenges in Machine Learning Model Management. *IEEE Data Engineering Bulletin* (2018).

[34] David Sculley, Gary Holt, et al. 2015. Hidden technical debt in machine learning systems. *NeurIPS.* 2503–2511.

[35] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and benchmarking the impact of GDPR on database systems. *PVLDB* (2020).

[36] Julia Stoyanovich, Bill Howe, and H.V. Jagadish. 2020. Responsible Data Management. *VLDB* 13, 12 (2020), 3474–3489.

[37] Paul E Utgoff, Neil C Berkman, and Jeffery A Clouse. 1997. Decision tree induction based on efficient tree restructuring. *Machine Learning* 29, 1 (1997), 5–44.

[38] Louis Wehenkel and Mania Pavella. 1991. Decision trees and transient stability of electric power systems. *Automatica* 27, 1 (1991), 115–134.

[39] Yinjun Wu, Edgar Dobriban, and Susan B. Davidson. 2020. DeltaGrad: Rapid retraining of machine learning models. arXiv:cs.LG/2006.14755

[40] Ting Ye, Hucheng Zhou, Will Y Zou, et al. 2018. Rapidscorer: fast tree ensemble evaluation by maximizing compactness in data level parallelization. *KDD.*

[41] Jingren Zhou and Kenneth A Ross. 2002. Implementing database operations using SIMD instructions. *SIGMOD.* 145–156.