

An Efficient Compilation-Based Approach to Explaining Random Forests Through Decision Trees

Alnis Murtovi, Maximilian Schlüter^a and Bernhard Steffen^b

TU Dortmund University, Germany

{alnis.murtovi, maximilian.schlueter, bernhard.steffen}@tu-dortmund.de

Keywords: Random Forests, Decision Trees, Explainable AI, Compilation-Based Explainability.

Abstract: Tree-based ensemble methods like Random Forests often outperform deep learning models on tabular datasets but suffer from a lack of interpretability due to their complex structures. Existing explainability techniques either offer approximate explanations or face scalability issues with large models. In this paper, we introduce a novel compilation-based approach that transforms Random Forests into single, semantically equivalent decision trees through a recursive process enhanced with optimizations and heuristics. Our empirical evaluation demonstrates that our approach is over an order of magnitude faster than current state-of-the-art compilation-based methods while producing decision trees of comparable size.

1 INTRODUCTION


In recent years, machine learning has seen a significant rise in popularity. Although deep learning methods are prevalent in areas such as natural language processing, computer vision, and speech recognition, tree-based approaches, including *Random Forests* (Breiman, 2001) and *gradient boosted trees* (Friedman, 2001), frequently surpass deep learning models when working with tabular datasets (Grinsztajn et al., 2022; Shwartz-Ziv and Armon, 2022; Borisov et al., 2022). Tree ensemble methods rely on combining numerous decision trees. However, the interpretability of these models remains a significant challenge, particularly for ensemble methods that combine numerous decision trees, resulting in less transparent models (Guidotti et al., 2019).


A considerable amount of research has focused on enhancing the explainability of tree ensembles and their predictions. Widely used techniques, such as LIME (Ribeiro et al., 2016) and SHAP (Lundberg, 2017), adopt a model-agnostic approach by analyzing only the input-output relationships to produce explanations. However, due to the inherent limitations of these methods, they offer approximations, meaning that the explanations they generate are neither strictly necessary nor fully sufficient to account for the model's outcomes.

To address these limitations, logic-based approaches have been developed to offer formal and rigorous explanations by examining the internal structure of the models. These methods aim to provide guarantees about the necessity and sufficiency of feature subsets or value ranges that influence predictions, leveraging solvers such as SAT, SMT, and MaxSAT (Ignatiev et al., 2019a; Izza et al., 2023; Izza and Marques-Silva, 2021; Ignatiev et al., 2019b; Ignatiev et al., 2022; Audemard et al., 2023).

An alternative to the computation of explanations are compilation-based approaches, which compile the model into a representation from which explanations can be derived more efficiently (Marques-Silva, 2024). However, a significant limitation of these approaches is their inability to handle large and complex models effectively (Marques-Silva, 2024). For instance, the work of (Gossen and Steffen, 2021) compiles a Random Forest into a single, semantically equivalent Algebraic Decision Diagram (ADD) to provide a more interpretable representation of the model. In (Murtovi et al., 2025a) the authors propose a compilation-based approach to transform boosted trees into ADDs, and how to compute abductive and inflated explanations from these ADDs. While these approaches provide a more efficient way to generate explanations, they suffer from scalability issues, especially when dealing with large models, as the compilation process can be computationally expensive.

In this paper, we present a compilation-based approach that transforms a random forest into a single,

^a  <https://orcid.org/0000-0002-5100-7259>

^b  <https://orcid.org/0000-0001-9619-1558>

semantically equivalent decision tree. Our approach is based on a recursive process that constructs the decision tree by replacing the leaves of the first tree in the ensemble with the next tree in the ensemble, continuing until the entire forest is integrated. By applying optimizations and heuristics, we address the scalability issues prevalent in existing compilation-based methods. The resulting decision tree is semantically equivalent to the original random forest and can be used to explain its predictions in a more efficient manner. Specifically, the technique presented in (Murtovi et al., 2025a) can be used to generate abductive/inflated explanations from the decision tree.

The main contributions of this paper are as follows:

- We present a novel compilation-based approach to compile Random Forests into a single semantically equivalent decision tree.
- We propose several optimizations and heuristics to improve the efficiency of this transformation process.
- We evaluate our approach by comparing it with the state-of-the-art approach presented in (Gossen and Steffen, 2021).

To evaluate our approach, we consider the following research questions:

- RQ1.** Can our approach transform random forests into decision trees more efficiently than existing state-of-the-art methods?
- RQ2.** How does the size of decision trees generated by our method compare to those produced by state-of-the-art techniques?

Our experimental results demonstrate that our approach is over an order of magnitude faster than the current state-of-the-art (Gossen and Steffen, 2021), while producing decision trees that are only slightly larger. We believe this is a step towards scalable and efficient compilation-based methods for explaining tree ensemble models.

The remainder of this paper is structured as follows: Sec. 3 introduces decision trees, and random forests. In Sec. 2 we discuss related work on explaining random forests. Sec. 4, details our approach to transform random forests into decision trees. In Sec. 5 we propose several optimizations and heuristics to our approach, that improve on the time to transform the random forest, and the size of the resulting decision tree. We evaluate our approach in Sec. 6 by comparing it with the state-of-the-art approach presented in (Gossen and Steffen, 2021). Finally, Sec. 7 concludes the paper and outlines potential directions for future work.

2 RELATED WORK

In this section, we review related work on explaining random forests, categorizing these approaches into heuristic methods that approximate model behavior, logic-based methods that apply formal reasoning, and compilation-based techniques that transform models into interpretable forms.

2.1 Heuristic Explainability Approaches

Heuristic approaches like LIME (Ribeiro et al., 2016), SHAP (Lundberg, 2017), and Anchor (Ribeiro et al., 2018) aim to explain the predictions of black-box models by approximating their behavior locally or globally. While these methods are model-agnostic and can be applied to any machine learning model, they often provide approximate explanations that may not capture the full complexity of the model.

2.2 Logic-Based Explainability Approaches

In contrast to heuristic methods, logic-based approaches aim to provide formal and rigorous explanations for model predictions by taking into account the internal structure of the model. One type of logic-based explanation that has become popular is abductive explanations (Ignatiev et al., 2019a). Abductive explanations aim to identify minimal subsets of feature assignments that are sufficient to yield a specific prediction from a model. In the context of random forests, this involves finding the smallest set of input features that, when fixed, guarantee the model's output for a particular instance. These explanations provide insights into which features are essential for the prediction.

To generate abductive explanations, algorithms often utilize formal reasoning tools such as SAT, SMT, or MaxSAT solvers. These solvers are used to encode the model's decision logic into logical constraints and then compute minimal satisfying assignments that correspond to the explanations. For example, a SAT-based approach to generate abductive explanations for random forests is presented in (Izza and Marques-Silva, 2021), while (Ignatiev et al., 2022) introduces a MaxSAT solver-based method for boosted trees. An optimization over the latter approach, aimed at improving the efficiency of the transformation process, is discussed in (Audemard et al., 2023).

Another type of logic-based explanation is inflated explanations which have been introduced in (Izza et al., 2023). Unlike abductive explanations that only identify a minimal set of features that are necessary

for a prediction, inflated explanations also require an interval for each feature that cannot be increased in either direction without changing the prediction. In (Izza et al., 2023), the authors present an SMT-based approach to generate inflated explanations for random forests.

2.3 Compilation-Based Explainability Approaches

Compilation-based approaches focus on translating the decision process of a random forest or other complex models into an interpretable form, often using logical formulas or some other compact representation. Such representations might allow for more efficient computation of explanations or reasoning about the model’s behavior. However, these techniques are often computationally intensive and may not scale well to very large models (Marques-Silva, 2024).

For example, (Shih et al., 2019) presents a compilation-based approach to transform bayesian network classifiers into decision graphs, which enables reasoning about the model’s behavior. Similarly, (Shi et al., 2020) proposes a method to compile binary neural networks into binary decision diagrams and sentential decision diagrams.

Our approach also falls into the category of compilation-based methods. In (Gossen and Steffen, 2021; Murtovi et al., 2025b), the authors show how to compile Random Forests into Algebraic Decision Diagrams (Bahar et al., 1997) (ADDs), while (Murtovi et al., 2025b) proposes several optimizations to improve the efficiency of the transformation process. In (Murtovi et al., 2025a), the authors present a compilation-based approach to transform boosted trees into ADDs and how to compute abductive and inflated explanations from these ADDs. Building upon these compilation-based methods, our approach aims to improve efficiency of the transformation process by transforming random forests into semantically equivalent decision trees.

3 BACKGROUND

In this section, we introduce the relevant concepts.

3.1 Classification Problems

Classification is a supervised learning task where the objective is to assign input instances to predefined categories or classes. Let $\mathcal{F} = \{1, \dots, m\}$ denote the set of features and $\mathcal{C} = \{1, \dots, K\}$ the set of classes.

The value of a feature $j \in \mathcal{F}$ is denoted as x_j where $x_j \in \mathbb{R}$. The feature space is defined as $\mathbb{F} := \mathbb{R}^m$. We use $\vec{x} = (x_1, \dots, x_m) \in \mathbb{F}$, for an arbitrary point in the feature space. A classifier implements a total classification function $\tau: \mathbb{F} \rightarrow \mathcal{C}$, that maps each instance in the feature space to one of the predefined classes.

3.2 Decision Trees

Decision trees (Quinlan, 1986) are a widely used class of models for classification and regression tasks due to their simplicity and interpretability. A decision tree T consists of internal nodes and leaf nodes. Each internal node represents a decision based on a feature x_j of the form $x_j < t$ where $t \in \mathbb{R}$ is a threshold value¹. Based on the outcome of the decision, the tree branches to one of two child nodes: the true child if the condition is satisfied, and the false child otherwise. Leaf nodes are associated with class labels and represent the final prediction of the tree for instances that reach them. While decision trees are easy to interpret and visualize, they have a tendency to overfit the training data, leading to high variance and reduced generalization performance on unseen data.

3.3 Random Forests

Random forests (Breiman, 2001) address the limitations of individual decision trees by constructing an ensemble of trees. A random forest consists of n decision trees T_1, \dots, T_n , where each tree is trained on a different subset of the training data. By combining the predictions of multiple trees, random forests reduce the variance of the model and improve the predictive power.

Given an input instance \vec{x} , each decision tree T_i in the random forest produces a prediction $T_i(\vec{x}) \in \mathcal{C}$, where $i = 1, \dots, n$. The final prediction \hat{y} is then obtained by aggregating these individual predictions through majority voting:

$$\hat{y} = \arg \max_{c \in \mathcal{C}} \sum_{i=1}^n \mathbb{I}(T_i(\vec{x}) = c),$$

where $\mathbb{I}(\cdot)$ is the indicator function that returns 1 if the condition is true and 0 otherwise.

Random forests mitigate high variance associated with single decision trees by averaging multiple trees, thereby enhancing the model’s robustness and predictive accuracy. They are also less prone to overfitting compared to individual trees, especially when a large number of trees are used in the ensemble.

¹The techniques discussed also extend to other types of comparisons or categorical features.

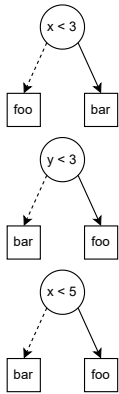


Figure 1: A random forest consisting of three decision trees.

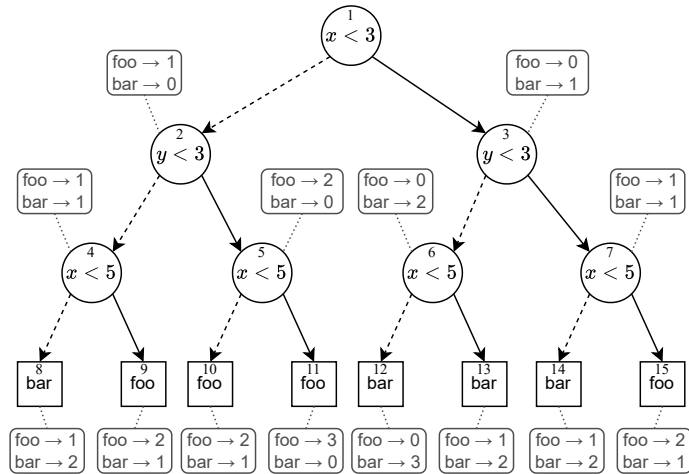


Figure 2: A single decision tree resulting from our transformation process.

4 TRANSFORMING RANDOM FORESTS TO DECISION TREES

In this section, we present our approach to aggregate the decision trees of a Random Forest into an individual, semantically equivalent decision tree. The advantage of the single decision tree representation is that it eliminates the need to keep track of the votes for each class in the ensemble. Instead of computing the majority class at prediction time, we just need to traverse the decision tree to reach the leaf node with the majority class. The core idea is to iteratively join the decision trees of the ensemble by appending each tree at the leaves of the current aggregate, effectively unrolling the ensemble into a single decision tree that represents the entire model.

For deciding what prediction is made at the leaves of the aggregated tree, we must keep track of the votes of the decision trees of the ensemble. For that, we accumulate votes for each class by tracking the class labels at the leaf nodes of each tree. When we reach a leaf node in the last tree of the ensemble, we determine the majority class based on the accumulated votes and assign it to the corresponding leaf node in the transformed decision tree. This ensures that for any given input, the transformed decision tree produces the same prediction as the original ensemble, accurately mirroring its decision-making process.

Alg. 1 outlines the transformation process. The algorithm takes as input a random forest with decision trees T_1, \dots, T_n , the current node t_i in tree T_i , and an array *votes* to keep track of the cumulative votes for each class. The function `RF2DT` is initially called with

the root node of the first tree ($t_i = T_1.root$), and the *votes* array initialized to zeros.

The algorithm recursively processes each tree in the ensemble, starting with the first tree. When a leaf node is reached, the algorithm increases the vote count for the class associated with that leaf (Line 4). If the current tree is the last in the ensemble ($i = n$), it creates a leaf node in the transformed decision tree with the class that has the most votes (Lines 5–6). If it is not the last tree, the algorithm proceeds by recursively calling `RF2DT` with the root node of the next tree ($T_{i+1}.root$) and the updated *votes* array. After the recursive call, the algorithm decrements the vote count for the current class label to backtrack correctly. For internal nodes, the algorithm copies the decision criteria (feature and threshold) from the current node to a new node in the transformed tree (Line 11–12). It then recursively processes the true and false child nodes (Lines 13–14). The results of these recursive calls are assigned to the corresponding child pointers of the new node in the transformed tree. This process ensures that all possible paths through the ensemble are explored and represented in the single decision tree.

Example 1. We illustrate the transformation process with a simple example. Consider a random forest consisting of three decision trees,² as shown in Fig. 1. Applying our transformation process to this random forest results in a single decision tree, as shown in Fig. 2, which given an input directly returns the majority vote class. Consider the rightmost path in the decision tree in Fig. 2. When we follow the true edge

²Dotted edges represent false edges.

Algorithm 1: RF2DT: Transforming a random forest to a single decision tree.

Input : Random Forest T_1, \dots, T_n , Current node t_i in tree T_i , Array $votes$

Output: A node in the transformed decision tree.

```

1 Function RF2DT( $T_1, \dots, T_n, t_i, votes$ ):
2   Create new node  $newNode$ ;
3   if  $t_i$  is a leaf then
4     // Increase vote count for the
      class in the current leaf
       $votes[t_i.class] \leftarrow votes[t_i.class] + 1$ ;
5     if  $i = n$  then
6       // If processing the last
        tree, create a leaf with
        the majority class
         $newNode.class \leftarrow \arg \max_{class} votes$ ;
7     else
8       // Recursively call RF2DT for
        the next tree in the list
         $newNode \leftarrow$ 
        RF2DT( $T_1, \dots, T_n, t_{i+1}.root, votes$ );
9       // Backtrack the vote count
        after recursion
         $votes[t_i.class] \leftarrow votes[t_i.class] - 1$ ;
10    else
11       $newNode.feature \leftarrow t_i.feature$ ;
12       $newNode.threshold \leftarrow t_i.threshold$ ;
13       $newNode.true \leftarrow$ 
        RF2DT( $T_1, \dots, T_n, t_i.true, votes$ );
14       $newNode.false \leftarrow$ 
        RF2DT( $T_1, \dots, T_n, t_i.false, votes$ );
15    return  $newNode$ ;
```

of $x < 3$ in the first tree, we reach the leaf node with class foo, so the vote count for foo is increased by one resulting in the voting vector (1,0). We now continue with the second tree, where we follow the true edge of $y < 3$ and reach the leaf node with class bar, so the vote count for bar is increased by one to (1,1). Finally, we reach the last tree, where we follow the true edge of $x < 5$ and reach the leaf node with class foo, so the vote count for foo is increased by one to (2,1). Since we are processing the last tree, we create a leaf node with the majority class, which is foo in this case.

While the presented transformation already simplifies the prediction process, it results in an exponential growth in the size of the resulting decision tree. Consider a random forest consisting of n decision trees, each with m leaf nodes. The resulting decision tree will have m^n leaf nodes, which is infeasible

for large values of n and m . To avoid this exponential blowup, we present further optimizations that not only improve the prediction time but also reduce the time required for the transformation process.

5 OPTIMIZATIONS

When we take a closer look at our transformation, we can identify several inefficiencies that can be optimized.

5.1 Deduplication of Isomorphic Subtrees

The decision tree resulting from the transformation process can become quite large, especially for random forests with many trees. To optimize memory usage during the recursive construction of the decision tree, we implement a deduplication strategy that avoids the creation of identical subtrees. Fig. 2 shows that the subtrees beginning at the leftmost and rightmost $x < 5$ nodes are identical. Instead of creating two separate instances of the same subtree, we can reuse the existing subtree. This idea is similar to the concept of deduplication in Binary Decision Diagrams (BDDs) (Bryant, 1986) and Algebraic Decision Diagrams (ADDs) (Bahar et al., 1997), though they require a variable ordering to identify isomorphic subgraphs.

We implement this through the use of a hash map that serves as a memoization cache for subtrees. In the recursive process, when a new subtree is generated, we compute a hash key that represents the subtree's structure and content. Before returning the new subtree, we check if the key is already present in the hash map. If it is, we return the subtree from the hash map instead of the newly created one. Otherwise, we add the new subtree to the hash map and return it. This optimization significantly reduces the memory usage during the transformation process.

5.2 Redundant Predicate Elimination

One way to optimize the transformation process is to eliminate redundancies resulting from the predicates seen along the path (Gossen and Steffen, 2021; Murtovi et al., 2023). Consider the rightmost path in Fig. 2. When we reach $x < 5$, we already know that $x < 3$ is true since we followed the true edge of $x < 3$ in the first tree. If $x < 3$ is true, $x < 5$ must also be true, so instead of creating a new node for $x < 5$, we can directly continue with its true successor node. This

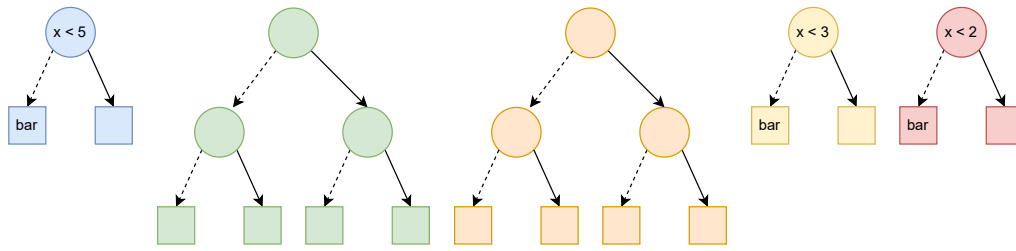


Figure 3: A random forest consisting of five decision trees.

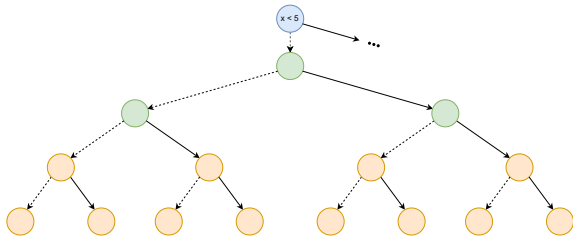


Figure 4: An illustration of the transformation process of the random forest in Fig. 3.

way, we avoid having to process the false subtree of $x < 5$, which reduces the size of the resulting decision tree, and thus the time required for the transformation process. At prediction time, we also benefit from this optimization since we can avoid unnecessary comparisons.

5.3 Early Stopping

The decision tree resulting from our transformation only returns the majority vote class, so the exact class votes are not needed. We can make use of this fact in the following way: During the transformation, when we reach a leaf node in a tree, we increase the vote count for the class associated with that leaf. If the accumulated votes for this class exceed the maximum possible votes that any other class could obtain from the remaining trees, we can stop the transformation early and create a leaf node with this class. This way, we can avoid processing the remaining trees in the ensemble. In Fig. 1 this can be observed on two paths, i.e., when foo receives two votes and bar zero, and when bar receives two votes and foo zero. It is also clear that the internal node $x < 5$ is redundant in this case since both successor nodes lead to the same class. This optimization has already been applied in (Murtovi et al., 2025b) for random forests. Although this approach incurs some overhead due to computing the top two classes by votes, it can significantly reduce the overall transformation time.

5.4 Abstract Interpretation

One limitation of early stopping is that it can only be applied after at least half of the trees have been processed. In (Murtovi et al., 2025b) the authors propose a non-semantic preserving approach to apply early stopping before processing half of the trees. While this approach is faster, it is not semantically preserving. In this paper, we present an approach based on abstract interpretation that allows us to apply early stopping before processing half of the trees while still preserving semantics.

Consider the random forest in Fig. 3 consisting of 5 decision trees. When we follow the false successor of $x < 5$, we reach the leaf node with class *bar*. Our transformation process would now proceed with the next tree in the ensemble, and when a leaf node is reached, proceed with the third tree and so on. When the fourth tree is reached, we know that $x < 3$ must be false since we followed the false edge of $x < 5$ in the first tree, so we land in the leaf node with class *bar*. The same applies to the fifth tree, where we know that $x < 2$ must be false, so we also land in the leaf node with class *bar*. We have now reached a leaf in the last tree of the ensemble, so we create a leaf node with the majority class, which is *bar* in this case. Fig. 4 sketches how the resulting tree would look like following the false edges of the first tree, and processing just the next 2 trees in the ensemble. If we had first processed trees 4 and 5, instead of 2 and 3, we could have applied early stopping, and we would have created a leaf node with class *bar* after processing the third tree. In the end, the false edge of the first tree would directly lead to the leaf node with class *bar*. This would have saved us from processing the remaining trees in the ensemble. We make use of this observation to apply early stopping before processing half of the trees in the ensemble.

Alg. 2 outlines an abstract interpretation approach to apply the early stopping before processing half of the trees. The algorithm takes as input a node t in the decision tree and a path condition pc , which represents the accumulated predicates along the path from the root to the current node. When a leaf node is

reached, the algorithm returns the class label of the leaf node (Line 3). If the path condition implies that the predicate of the current node is always true or false, the algorithm directly continues with the true (Line 5) or false successor node (Line 7), respectively. Otherwise, both successor nodes are processed. If the class labels of the true and false successor subtrees are the same, the algorithm returns this class label (Line 12). If the class labels are different, the algorithm returns “unsure” (Line 14), indicating that depending on the input, the class label could be different. Essentially, the algorithm returns a class if, given the path condition, this is the only class that can be reached, and “unsure” otherwise.

Algorithm 2: Abstract Interpretation of a Decision Tree.

Input : Node t , Path condition pc
Output: Class label or “unsure”.

```

1 Function AbsIntDT ( $t, pc$ ) :
2   if  $t$  is a leaf then
3     return  $t.class$ ;
4   if  $pc \implies (t.feature < t.threshold)$  then
5     return AbsIntDT ( $t.true, pc$ );
6   if  $pc \implies \neg(t.feature < t.threshold)$  then
7     return AbsIntDT ( $t.false, pc$ );
8   else
9      $trueClass \leftarrow$  AbsIntDT ( $t.true, pc$ );
10     $falseClass \leftarrow$  AbsIntDT ( $t.false, pc$ );
11    if  $trueClass = falseClass$  then
12      return  $trueClass$ ;
13    else
14      return “unsure”;

```

Alg. 3 shows how the abstract interpretation can be used to apply the abstract early stopping optimization. In the algorithm, the array *safeVotes* keeps track of the number of votes each class is guaranteed to receive from the unprocessed trees, based on the current path condition. The variable *freeVotes* counts the number of votes that can be assigned to any class from the unprocessed trees. The algorithm calls the abstract interpretation on each unprocessed tree with the current path condition (Line 5). The result is either a specific class label, indicating that the tree will always predict that class given the path condition, or “unsure”, indicating multiple classes are possible. If the abstract interpretation returns a class label, the algorithm increases the vote count for this class (Line 7). If the abstract interpretation returns “unsure”, the algorithm increases the number of free votes (Line 9). After processing all unprocessed trees, the algorithm calculates $totalVotes = votes[c] + safeVotes[c]$ for

each class c , combining the votes accumulated so far with the guaranteed votes from the unprocessed trees. It then identifies the class with the highest total votes (*max*), the second-highest (*secondMax*), and the idx of the class with the highest votes (*idx*).

The algorithm checks whether the leading class has more votes than the sum of the second-highest votes and the number of free votes (Line 18). If this condition holds, it means that even if all free votes went to the runner-up class, it still would not surpass the leading class. In this case, the algorithm safely returns the leading class as the winner (Line 19). Otherwise, it returns “unsure” (Line 21), indicating that early stopping cannot be applied yet.

Algorithm 3: Abstract Early Stopping.

Input : Random Forest T_1, \dots, T_n , Tree T_i , Array *votes*, Path condition pc
Output: Winning class or “unsure”

```

1 Function AbsES ( $T_1, \dots, T_n, T_i, votes, pc$ ) :
2   Initialize: Array safeVotes[1 ..  $K$ ] of
   integers with 0;
3    $freeVotes \leftarrow$  0;
4   for  $j \leftarrow i + 1$  to  $n$  do
5      $class \leftarrow$  AbsIntDT ( $T_j.root, pc$ );
6     if  $cls \neq$  “unsure” then
7        $safeVotes[cls] \leftarrow safeVotes[cls] + 1$ ;
8     else
9        $freeVotes \leftarrow freeVotes + 1$ ;
10   $max, secondMax, idx \leftarrow -\infty, -\infty, -1$ ;
11  for  $c \leftarrow 1$  to  $K$  do
12     $totalVotes \leftarrow votes[c] + safeVotes[c]$ ;
13    if  $totalVotes > max$  then
14       $secondMax, max \leftarrow max, totalVotes$ ;
15       $idx \leftarrow c$ ;
16    else if  $totalVotes > secondMax$  then
17       $secondMax \leftarrow totalVotes$ ;
18  if  $max > secondMax + freeVotes$  then
19    return  $idx$ ;
20  else
21    return “unsure”;

```

5.5 Heuristic Optimizations

While our approach based on abstract interpretation allows us to apply early stopping before processing half of the trees, it is computationally more expensive than the early stopping approach proposed in (Murtovi et al., 2025b). Each time we reach a leaf node in a tree, we need to apply the abstract interpretation

on each tree that has not been processed yet. For a tree consisting of k nodes, the time complexity of the abstract interpretation is $O(k)$, so if n trees still have to be processed each with a maximum of k nodes, the time complexity is $O(n \cdot k)$. To mitigate this computational overhead, we implemented a simple heuristic that predicts when our abstract interpretation will confirm that a particular class is guaranteed to win. Our heuristic consists of two parts:

- If we have already processed half of the trees, instead of applying the abstract interpretation, we directly apply the early stopping optimization.
- Otherwise, we only try our abstract interpretation approach if the class with the most votes has received at least 70% of the votes it could have received so far.

This heuristic tries to avoid the computational overhead of the abstract interpretation when it will not provide any benefit.

Another heuristic we implemented is based on the order of the trees in the ensemble. In general, the trees in the ensemble can be processed in any order, and the semantics of the resulting decision tree will be the same. However, the order in which the trees are processed can have an impact on the time required for the transformation process, and the size of the resulting decision tree. Our heuristic uses abstract interpretation to always select the next tree with the lowest number of reachable leaf nodes. By prioritizing trees that are simpler under the current path conditions, we can reduce the computational effort required for the transformation.

5.6 Optimized Transformation Algorithm

Alg. 4 enhances Alg. 1 with the optimizations and heuristics discussed in this section. The key differences are the following:

- The algorithm now maintains a path condition pc that represents the predicates seen along the path to the current node³. This allows the algorithm to apply the redundant predicate elimination optimization which is described in Sec. 5.2. If the path condition implies that the predicate of the current node is always true (Line 15), the algorithm can skip the creation of the current node and directly continue with the true successor node

³The actual implementation keeps track of an interval for each feature which allows one to update the path condition and perform the checks in Line 15 and 17 in $O(1)$ time.

(Line 16). Similarly, if the predicate is always false (Line 17), the algorithm proceeds with the false successor node (Line 18).

- A hash map *uniqueMap* is used to store and reuse identical subtrees. The algorithm implements the deduplication optimization described in Sec. 5.1 in lines 26–30.
- When a leaf node is reached in a tree, which is not the last tree in the ensemble, the algorithm checks if early stopping can be applied. This can either be the early stopping optimization described in (Murtovi et al., 2025b) or the abstract interpretation based early stopping.

For reasons of space, we do not provide a full algorithm for the heuristic that chooses the next tree to process based on the number of reachable leaf nodes, but the general idea is when the algorithm reaches a leaf node and has to decide which tree to process next, it uses the abstract interpretation to compute the number of reachable leaf nodes for each tree that has not been processed yet and choose the tree with the lowest number of reachable leaf nodes. This heuristic has some overhead, because the algorithm always needs to keep track of the decision trees that have not been processed yet, whereas previously we could just process the next tree in the list.

6 EVALUATION

This section presents an evaluation of our approach by applying our transformation to random forests trained on several datasets from the UCI Machine Learning Repository (Asuncion et al., 2007). We compare our approach to the state-of-the-art approach presented in (Gossen and Steffen, 2021) and evaluate the impact of our optimizations and heuristics on the transformation time and the size of the resulting decision tree.

6.1 Experimental Setup

We evaluated our approach on a machine with an Intel(R) Xeon(R) Gold 6152 CPU 2.10GHz with 502GB of RAM. We implemented our approach in Java and used exactly the same random forests as in (Murtovi et al., 2025b) to ensure a fair comparison. These random forests were trained on datasets from the UCI Machine Learning Repository (Asuncion et al., 2007) as described in Table 1.

We perform measurements with the following configurations:

- *ADD*: The state-of-the-art approach presented in (Gossen and Steffen, 2021) with the early stop-

Algorithm 4: Transform Random Forest to Decision Tree.

Input : Random Forest T_1, \dots, T_n , Current node t_i in Tree T_i , Array $votes$, Path condition pc , Hash map $uniqueMap$

Output: A decision tree that represents the random forest

```

1 Function RF2DTopt ( $T_1, \dots, T_n, t_i, votes, pc, uniqueMap$ ):
2   Create new node  $newNode$ ;
3   if  $t_i$  is a leaf then
4     // Increase vote count for the class in the current leaf
5      $votes[t_i.class] \leftarrow votes[t_i.class] + 1$ ;
6     if  $i = n$  then
7       // If processing leaf in the last tree, create leaf with the majority class
8        $newNode.class \leftarrow \arg \max_{class} votes$ ;
9     else
10      // Recursively call RF2DTopt for the next tree in the list
11       $safeWinner \leftarrow earlyStop(T_1, \dots, T_n, t_i, votes, pc)$ ;
12      if  $safeWinner \neq "unsure"$  then
13         $newNode.class \leftarrow safeWinner$ ;
14      else
15         $newNode \leftarrow RF2DTopt(T_1, \dots, T_n, t_{i+1}, votes, pc, uniqueMap)$ ;
16      // After recursive call, decrease the vote count
17       $votes[t_i.class] \leftarrow votes[t_i.class] - 1$ ;
18   else
19     if  $pc \implies (t_i.feature < t_i.threshold)$  then
20        $newNode \leftarrow RF2DTopt(T_1, \dots, T_n, t_i.true, votes, pc, uniqueMap)$ ;
21     else if  $pc \implies \neg(t_i.feature < t_i.threshold)$  then
22        $newNode \leftarrow RF2DTopt(T_1, \dots, T_n, t_i.false, votes, pc, uniqueMap)$ ;
23     else
24        $newNode.feature \leftarrow t_i.feature$ ;
25        $newNode.threshold \leftarrow t_i.threshold$ ;
26        $newNode.true \leftarrow$ 
27          $RF2DTopt(T_1, \dots, T_n, t_i.true, votes, pc \wedge (t_i.feature < t_i.threshold), uniqueMap)$ ;
28        $newNode.false \leftarrow$ 
29          $RF2DTopt(T_1, \dots, T_n, t_i.false, votes, pc \wedge \neg(t_i.feature < t_i.threshold), uniqueMap)$ ;
30       if  $newNode.true == newNode.false$  then
31          $newNode \leftarrow newNode.true$ ;
32   if  $uniqueMap.contains(newNode)$  then
33     return  $uniqueMap.get(newNode)$ ;
34   else
35      $uniqueMap.put(newNode, newNode)$ ;
36   return  $newNode$ ;

```

ping optimization presented in (Murtovi et al., 2025b).

- *DT*: Our basic approach as described in Alg. 1.
- *ES*: Our approach with the early stopping optimization from (Murtovi et al., 2025b).
- *AbsES*: Our approach with the abstract early stopping optimization (Alg. 3).

- *HEUR*: Our approach with the heuristic optimization that decides when to apply the abstract early stopping optimization.

- *ORD*: Configuration *HEUR* extended with the heuristic optimization that processes the trees in the order such that the tree with the lowest number of reachable leaf nodes is processed first.

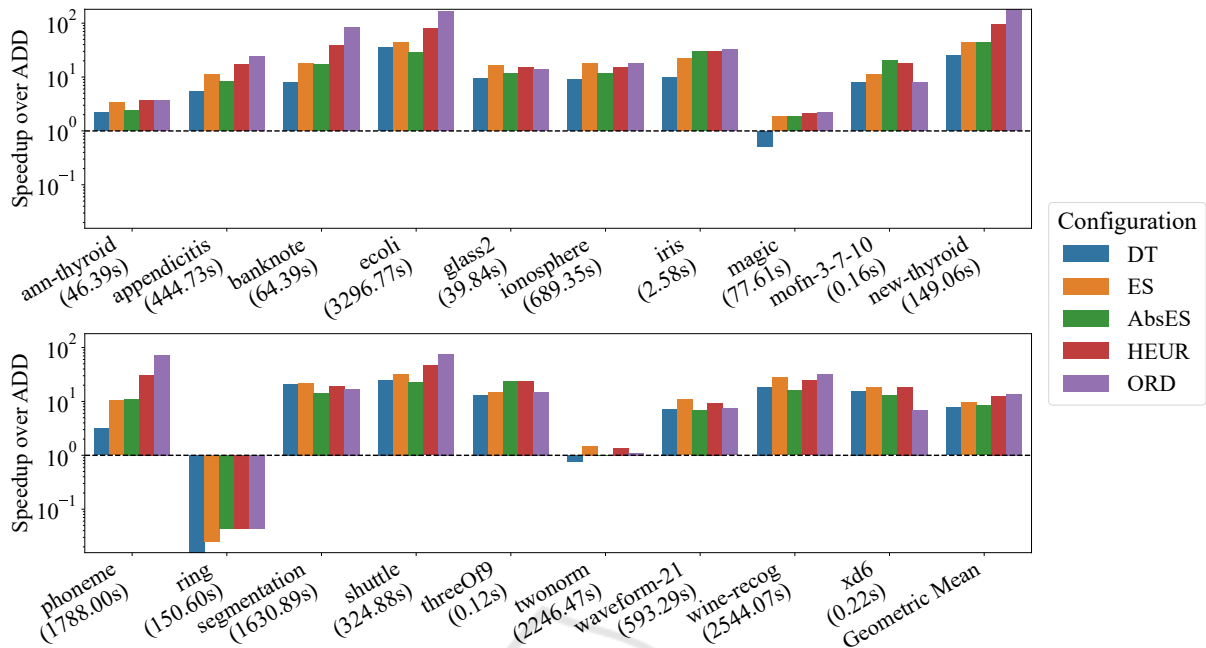


Figure 5: Speedups over *ADD* for the transformation of random forests into decision trees. Baseline times (in seconds) are shown in parentheses next to dataset names.

Table 1: Overview of datasets and the learned random forests (Murtovi et al., 2025b) (#F = Number of features, #I = Number of test instances, #C = Number of classes, #T = Number of trees, #N = Number of nodes in the Random Forest, #P = Number of unique predicates, D = Maximum depth, %A = Accuracy of the Random Forest on test set).

Dataset	#F	#I	#C	#T	#N	#P	D	%A
ann-thyroid	21	1426	3	25	555	146	4	98
appendicitis	7	22	2	50	722	207	4	90
banknote	4	270	2	100	1998	614	4	97
ecoli	7	66	5	100	2532	379	4	90
glass2	9	33	2	25	445	159	4	87
ionosphere	34	70	2	15	247	101	4	87
iris	4	30	2	100	1200	94	4	93
magic	10	3781	2	25	747	349	4	82
mofn-3-7-10	10	205	2	100	2904	10	4	85
new-thyroid	3	43	3	100	1452	237	4	100
phoneme	5	43	2	100	2836	957	4	78
ring	20	1480	2	25	625	287	4	83
segmentation	19	42	7	15	329	148	4	92
shuttle	9	11600	7	50	1296	205	4	99
threeOf9	9	103	2	100	1364	9	4	100
twonorm	29	1480	2	15	465	225	4	90
waveform-21	21	1000	3	15	465	214	4	80
wine-recog	13	36	3	25	399	152	4	97
xd6	9	103	2	100	2904	9	4	90

All configurations of our approach include the deduplication and redundant predicate elimination optimizations described in Sections 5.1 and 5.2.

6.2 Experimentation Results

Here, we present the results of our evaluation in which we discuss the research questions presented in Sec. 1.

RQ1: Can our approach transform random forests into decision trees more efficiently than existing state-of-the-art methods? Fig. 5 shows the speedup achieved by each configuration compared to *ADD*. Our approach outperforms the state-of-the-art approach in terms of transformation time for all datasets except for the *ring* dataset. The geometric speedup across all datasets for the configurations *DT*, *ES*, *AbsES*, *HEUR*, and *ORD* are 7.7, 9.5, 8.3, 12.5, and 13.6, respectively.

The highest speedups are achieved for the *ecoli* and *new-thyroid* datasets, where *ORD* is 165 and 180 times faster than *ADD*, respectively. For these two datasets, *ADD* requires a transformation time of 3296 and 149 seconds while *ORD* completes the transformation in only 19.8 and 0.82 seconds, respectively.

Although *AbsES* is more precise than *ES*, and should therefore be able to apply early stopping earlier, it is slower than *ES* for many datasets. This is due to the computational overhead of the abstract interpretation. However, our heuristic *HEUR* which combines *ES* and *AbsES* and decides when to apply which optimization, is faster than both *ES* and *AbsES* for most datasets.

The only dataset where our approach is slower than *ADD* is the *ring* dataset. For this dataset, *ADD* requires a transformation time of 150 seconds, while *ORD* requires 3423 seconds, so *ADD* is 22 times faster than *ORD*. We believe this is primarily because *ADD*s are more compact in this case due to their

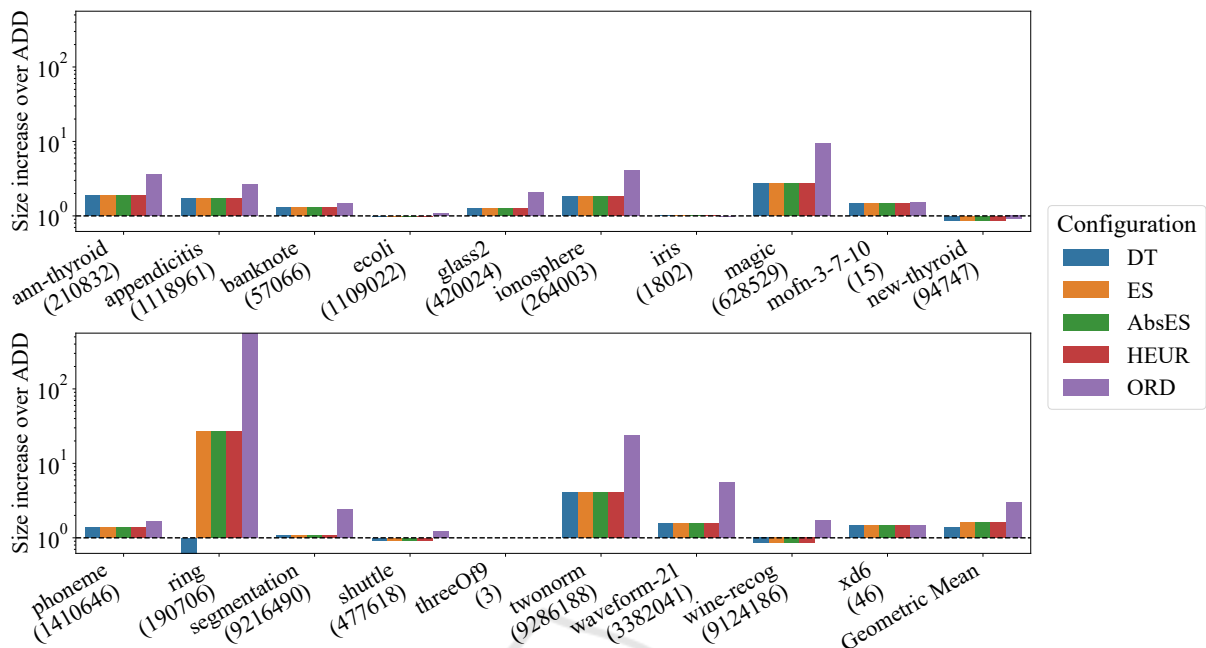


Figure 6: Size increase factor over *ADD* for the decision trees generated by our method. Baseline sizes (in number of nodes) are shown in parentheses next to dataset names.

variable ordering, since for the *ring* dataset the *ADD* is 561 times smaller than the decision tree generated by *ORD*.

Overall, our approach is more efficient than the state-of-the-art in terms of transformation time.

RQ2: How does the size of decision trees generated by our method compare to those produced by state-of-the-art techniques? Fig. 6 presents the size increase factor of the decision trees generated by each configuration compared to *ADD*. Our approach results in decision trees that are larger than those produced by *ADD* for most datasets. The geometric size increase factors for the configurations *DT*, *ES*, *AbsES*, *HEUR*, and *ORD* are 1.39, 1.62, 1.62, 1.62, and 3.03, respectively. The sizes resulting from *DT*, *ES*, *AbsES*, and *HEUR* are actually all the same, as they all result in the same decision tree. The difference in the geometric is due to *DT* timing out and not finishing within 3 hours for the *ring* dataset. The size increase factor for *ORD* is higher than for the other configurations, as at each leaf it decides which tree to process next based on the order of the trees. Processing the trees in a different order can result in less shared subtrees, which increases the size of the resulting decision tree.

The largest size increase factor can be observed for the *ring* dataset, where *ADD* generates an *ADD* with 190706 nodes while *ORD* generates a decision tree with 106994692 nodes, resulting in a size increase factor of 561. In general, *ADD*s are more compact than decision trees because they enforce a vari-

able ordering that allows for more sharing of nodes. However, for *ecoli*, *new-thyroid*, *shuttle*, and *wine-recog*, the size of the decision tree generated by our approaches (except for *ORD*) is smaller than the size of the *ADD* generated by *ADD*.

In summary, while our approach tends to produce larger decision trees than the state-of-the-art method for most datasets, the size increase is moderate in most cases.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented a novel approach to transform random forests into semantically equivalent decision trees. The primary motivation for this transformation is that representing the ensemble as a single decision tree enables more efficient computation of both abductive and inflated explanations, as demonstrated in (Murtovi et al., 2025a). Our approach is based on the idea of creating a single decision tree that represents the entire ensemble. We also introduced several optimizations and heuristics to improve the transformation process. Our evaluation showed that our approach outperforms the state-of-the-art approach in terms of transformation time by an order of magnitude on average.

In future work, we plan to investigate further optimizations and heuristics to improve the transforma-

tion process. We also plan to apply our approach to other types of tree ensembles, such as gradient boosted trees.

REFERENCES

- Asuncion, A., Newman, D., et al. (2007). Uci machine learning repository.
- Audemard, G., Lagniez, J.-M., Marquis, P., and Szczepanski, N. (2023). Computing abductive explanations for boosted trees. In *International Conference on Artificial Intelligence and Statistics*, pages 4699–4711. PMLR.
- Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., and Somenzi, F. (1997). Algebraic decision diagrams and their applications. *Formal methods in system design*, 10:171–206.
- Borisov, V., Leemann, T., Seßler, K., Haug, J., Pawelczyk, M., and Kasneci, G. (2022). Deep neural networks and tabular data: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21.
- Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.
- Bryant (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691.
- Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.
- Gossen, F. and Steffen, B. (2021). Algebraic aggregation of random forests: towards explainability and rapid evaluation. *International Journal on Software Tools for Technology Transfer*, pages 1–19.
- Grinsztajn, L., Oyallon, E., and Varoquaux, G. (2022). Why do tree-based models still outperform deep learning on typical tabular data? In *NeurIPS*.
- Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Gianotti, F., and Pedreschi, D. (2019). A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5):93:1–93:42.
- Ignatiev, A., Izza, Y., Stuckey, P. J., and Marques-Silva, J. (2022). Using maxsat for efficient explanations of tree ensembles. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3776–3785.
- Ignatiev, A., Narodytska, N., and Marques-Silva, J. (2019a). Abduction-based explanations for machine learning models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1511–1519.
- Ignatiev, A., Narodytska, N., and Marques-Silva, J. (2019b). On validating, repairing and refining heuristic ML explanations. *CoRR*, abs/1907.02509.
- Izza, Y., Ignatiev, A., Stuckey, P. J., and Marques-Silva, J. (2023). Delivering inflated explanations. *CoRR*, abs/2306.15272.
- Izza, Y. and Marques-Silva, J. (2021). On explaining random forests with SAT. In Zhou, Z., editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 2584–2591. ijcai.org.
- Lundberg, S. (2017). A unified approach to interpreting model predictions. *arXiv preprint arXiv:1705.07874*.
- Marques-Silva, J. (2024). Logic-based explainability: Past, present & future. *CoRR*, abs/2406.11873.
- Murtovi, A., Bainczyk, A., Nolte, G., Schlüter, M., and Steffen, B. (2023). Forest GUMP: a tool for verification and explanation. *Int. J. Softw. Tools Technol. Transf.*, 25(3):287–299.
- Murtovi, A., Schlüter, M., and Steffen, B. (2025a). Computing inflated explanations for boosted trees: A compilation-based approach. In Hinchey, M. and Steffen, B., editors, *The Combined Power of Research, Education, and Dissemination - Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday*, volume 15240 of *Lecture Notes in Computer Science*, pages 183–201. Springer.
- Murtovi, A., Schlüter, M., and Steffen, B. (2025b). Voting-based shortcuts through random forests for obtaining explainable models. In Graf, S., Pettersson, P., and Steffen, B., editors, *Real Time and Such - Essays Dedicated to Wang Yi to Celebrate His Scientific Career*, volume 15230 of *Lecture Notes in Computer Science*, pages 135–153. Springer.
- Quinlan, J. R. (1986). Induction of decision trees. *Mach. Learn.*, 1(1):81–106.
- Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144.
- Ribeiro, M. T., Singh, S., and Guestrin, C. (2018). Anchors: High-precision model-agnostic explanations. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32.
- Shi, W., Shih, A., Darwiche, A., and Choi, A. (2020). On tractable representations of binary neural networks. *arXiv preprint arXiv:2004.02082*.
- Shih, A., Choi, A., and Darwiche, A. (2019). Compiling bayesian network classifiers into decision graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7966–7974.
- Shwartz-Ziv, R. and Armon, A. (2022). Tabular data: Deep learning is not all you need. *Inf. Fusion*, 81:84–90.