TE2Rules: Explaining Tree Ensembles using Rules

G Roshan Lal, Xiaotong (Elaine) Chen, Varun Mithal

LinkedIn AI

rlal@linkedin.com, elachen@linkedin.com, vamithal@linkedin.com

arXiv:2206.14359v4 [cs.LG] 25 Aug 2023

Abstract

Tree Ensemble (TE) models (like Gradient Boosted Trees) often provide higher prediction performance compared to single decision trees. However, TE models generally lack transparency and interpretability, as humans have difficulty understanding their decision logic. This paper presents a novel approach to convert a TE trained for a binary classification task, to a rule list (RL) that closely approximates the TE and is interpretable for a human. This RL can effectively explain the model even on the minority class predicted by the model. Experiments on benchmark datasets demonstrate that, (i) predictions from the RL generated by TE2Rules have higher fidelity (with respect to the original TE) compared to state-of-the-art methods, (ii) the run-time of TE2Rules is comparable to that of some other similar baselines and (iii) the run-time of TE2Rules algorithm can be traded off at the cost of a slightly lower fidelity.

Introduction

In recent years, many decision support systems have been constructed as black box models using machine learning such as Tree Ensembles (TE) and Deep Neural Networks. Lack of understanding of the internal logic of decision systems constitutes both a practical and an ethical issue, especially for critical tasks that directly affect the lives of people like health care, credit approval, criminal justice etc. In such use cases, there is a possibility of making wrong decisions, learned from spurious correlations in the training data. The cost of making wrong decisions in these domains is very high. Hence, having some explanations (like what part of the input is the model focusing on, or under what conditions satisfied by the input, does the model behave similarly) is important for building trust on these decision systems. Moreover, recent legal regulations like General Data Protection Regulation (GDPR, 2018) enables all individuals to obtain "meaningful explanations of the logic involved" when automated decision making takes place.

In this work, we focus our attention on explaining tree ensemble models which are popular in many use cases involving tabular data (Grinsztajn, Oyallon, and Varoquaux 2022; Qin et al. 2021). Further, we only work with binary classification problems since it is the common scenario in many trust and safety critical decision making systems like disease diagnosis, spam detection etc., In binary classification setting, explaining one of the classes may be more important than the other. For example in domains like healthcare or fraud detection, it is important to explain why a model flagged an input data point as positive (say detected tumor or predicted scammer), while in some other domains like credit approval it maybe more important to explain rejections. In this work, we attempt at explaining the minority class in the data, which often happens to be the positive class.

In a tree ensemble, every input data point traverses though the decision trees to reach a corresponding leaf node. Each leaf node gives a certain score to the input data point. When the sum of these leaf node scores exceeds a bias term, the input data gets classified into the positive class. Figure 1 shows the decision paths of two input data points, x_1 and x_2 that are classified as positive by a tree ensemble of 3 trees.

A typical tree ensemble consists of many (hundreds of) decision trees. Tree based models are not differentiable and cannot be explained using gradient based methods developed for deep networks (Sundararajan, Taly, and Yan 2017; Selvaraju et al. 2019). However, when the decision trees are not too deep, it is easy to understand each of them individually. Each leaf node can be explained using a rule consisting of the decisions made by the tree internally to reach that leaf node. But, adopting a similar approach for tree ensembles has a number of challenges. 1) In a tree ensemble, it is computationally hard to find which leaf node combinations (with one leaf node per tree) would gather enough score for the tree ensemble to classify an input data point as positive. For a tree ensemble with n trees of max depth d, the number of such possible leaf node combinations grows exponentially: $O(2^{dn})$. 2) Many of these leaf node combinations would be impossible to satisfy. For example, a rule formed from n node combinations containing the nodes: "age > 40 and gender = male" and "age ≤ 50 and gender = female", can never be satisfied by any input data point. 3) Even if a n node combination is known, such that there exists data points that satisfy it and most of those data points are classified as positive by the model, the corresponding rule formed by the conjunction of n (hundreds of) leaf node rules would be too long for a human to comprehend.

TE2Rules presents an efficient method to search across n node combinations from the tree ensemble to mine rules for

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

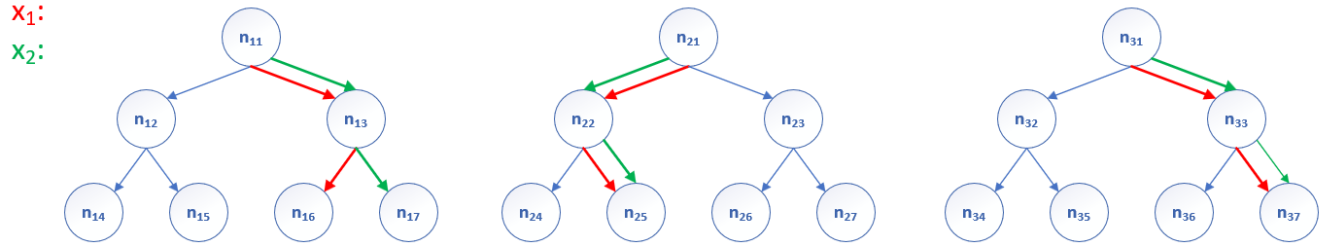

Figure 1: A tree ensemble showing decision paths of two inputs data points that are both classified as positive

the positive class such that the rules are short, with decent support (number of data points satisfying the rule) and have a high precision (fraction of data points satisfying the rule that are classified as positive by the model). TE2Rules takes a trained tree ensemble model and training data to mine a list of if-then rules which closely approximate the TE model predictions. Once these rules are generated, any data point classified as positive by the model can be explained by running the if-then rules on the same input. TE2Rules uses the following insights to overcome the challenges mentioned above.

Exploring internal nodes: In Figure 1, the data given to us consists of x_1 and x_2 among other data points. Let us assume that the only data points that satisfy the leaf node combinations of $n_{16} \cap n_{25} \cap n_{37}$ and $n_{17} \cap n_{25} \cap n_{37}$ are $\{x_1\}$ and $\{x_2\}$ respectively. Since, x_1 and x_2 are positives, both these rules have a precision of 100%. However, if we were to allow internal nodes in the n node combination, we find that the combination of $n_{13} \cap n_{25} \cap n_{37}$ has a even bigger support of $\{x_1, x_2\}$ with the same precision of 100%. Since, the decision path to reach an internal node like n_{13} is shorter than the leaf nodes (like n_{16}, n_{17}), the resulting rule of conjunctions is also shorter. An extreme example is that of using root node in the n node combination. Since all decision paths start at the root node, the corresponding rule to reach the root node is an empty rule. Thus, by searching across internal node combinations along with leaf nodes, we benefit from finding shorter rules. Including internal nodes does not drastically increase our search space, since a decision tree has only about as many internal nodes ($2^d - 1$) as leaf nodes (2^d).

Mining shorter rules before longer rules: For each n node combination that is explored, the support and precision of the corresponding rule can be computed from data. It is desirable to have short rules with high precision, i.e, for any high precision rule, modifying it by dropping any node (or replacing it with the corresponding root node) in its node combination should not result in another high precision rule. If such a case arises, the modified rule would be shorter and more desirable. Hence, to mine shorter rules early on, TE2Rules runs the search in n stages, where in the k^{th} stage we consider combinations of k non-root nodes from k different trees and $n - k$ root nodes from the rest of the trees. Thus, in stage 1, rules are mined using one non-root node from some tree and root nodes from other trees; in stage 2, rules are mined from non-root node combinations from any 2 trees (the rest being root nodes from the remaining $n - 2$ trees), and so on.

Pruning the search space: With every stage, the rules tend to grow longer and their support in data tends to decrease. In order theory, this is called *Anti-Monotone property*. This property occurs in many rule mining problems (Agrawal, Srikant et al. 1994). Consider a rule in stage k formed from k non-root nodes. If it has 0 support in data, any rule in stage $k + 1$ formed using the same k non-root nodes along with a new non-root node in the place of the root node would also have 0 support. Such rules are not worth exploring. TE2Rules leverages the Anti-Monotone property to generate stage k candidates efficiently, without having to go through a lot of node combinations of 0 support that are not worth exploring.

Reducing the number of rules: It is not necessary to search through all stages from 1 through n. Once all the positives in the dataset are covered by some high precision rules, further stages need not be explored. After mining rules for all positives, it is possible for multiple rules to cover similar sets of positives. We run a simplification step on the mined rules to select a small subset of rules which still cover all the positives in the data.

Once the rules are mined, we evaluate the rule list using *fidelity*, defined as the accuracy of the rule list's prediction with respect to the predictions of the trained model. A good explainer needs to agree with the model or have a high fidelity on test data. High overall fidelity in test data does not always guarantee high fidelity on positives especially when the positives happen to be the minority class in the dataset (which is often true in most datasets). We evaluate all explainers on their overall fidelity, fidelity on positives (minority class).

In this work, we show that 1) many existing state of the art rule based explainers for TE models have poor fidelity on positives (the minority class) though they may have good overall fidelity. 2) To solve this problem, we propose a novel method, TE2Rules, that can generate rules corresponding to a single class of interest (say positive class) by merging decision paths from multiple trees in the tree ensemble. Since all the rules are mined for a single class, there is no conflict among the labels predicted by different rules. 3) We show that the resulting rules have high overall fidelity and high fidelity on positives (the class of interest) even if the positives happen to be a minority class in the dataset. TE2Rules can achieve such high performance at comparable number of rules relative to existing baselines, at the cost of slightly higher run time. 4) By stopping the algorithm in the early stages, we can tradeoff fidelity with runtime.

Related Work

In the past, several methods have been proposed to explain a tree ensemble model using rules or decision trees that can

be better understood by a human. Some of these approaches work at a global level by approximating the tree ensemble with a set of rules or decision tree. While some other approaches work at a local (instance-level) by finding a rule or a decision tree that best explains the decisions of the tree ensemble for data points sampled from the neighborhood of that instance.

Rule-based explainers: inTrees(Deng 2019) generate rules from decisions made from individual trees in the tree ensemble and selects high precision rules among them. Another closely related method, ruleFit (Friedman and Popescu 2008) runs a sparse linear regression on rules generated from individual trees to select the most important rules. However, a sparse ensemble of rules is not as interpretable as a list of if-then rules. Both inTrees and ruleFit generate rules from nodes of individual trees and do not consider node combinations from multiple trees. Hence, their search space of rules is limited to rules from individual trees, resulting in low fidelity. deFragTrees (Hara and Hayashi 2018) identifies fragmented regions in the input space defined by the splits made by the tree ensemble and tries to simplify them into a short set of rules that are almost equivalent to the tree ensemble using bayesian inference. deFragTrees works on simplifying rules obtained from all possible node combinations from multiple trees in the ensemble and can achieve higher fidelity than methods like inTrees. However, most rule-based explainers are not targeted to explain any one single class. They often end up mining a lot of rules for the majority class and miss out on the minority class. In such cases, they are not very effective in explaining the minority class prediction.

Tree-based explainers: BATrees (Vidal, Pacheco, and Schiffer 2020), uses the tree ensemble to generate more labeled data points that can be used to fit a decision tree on the data. However, trying to obtain a single tree decision tree to represent a tree ensemble can result in a very deep tree, making it harder to interpret. Node Harvest (Meinshausen 2010) selects a few nodes from the shallow parts of the trees in the ensemble and creates an ensemble of shallow trees. The simplified model is easier to interpret than the original model. But, it is still not as easy to interpret as a decision tree or rule list.

Local instance-level explainers: Anchors (Ribeiro, Singh, and Guestrin 2018) finds high precision, if-then rules satisfied by the instance, using multi-armed bandit and beam search algorithms. LoRE (Local Rule-based Explanations) (Guidotti et al. 2018) constructs interpretable models (decision trees) based on local samples. These explainers are model agnostic and build rules that are locally correct at an instance level. They perform d a local search for every instance to be explained as it arrives. Hence, they can be prohibitively expensive if the objective is to explain a very large set of data points for a single model.

Interpretable models: Instead of explaining existing machine learning models, an alternate approach has been to build interpretable models like falling rule lists (Wang and Rudin 2015) directly from training data. Some of these approaches like SkopeRules (Gautier, Jafre, and Ndiaye 2020) learn rule lists by first fitting a tree ensemble model internally on the data and then extracting rules from it. SkopeRules generates rules directly from data and does not allow the user to specify a tree ensemble model to be explained. In this work, we only focus on explainers that take a tree ensemble model along with some data as input and generate human underdtandable explanations as output. Learning interpretable models from data is beyond the scope of our work.

In this work, we propose TE2Rules, a global, rule-based explainer for tree ensemble models trained on binary classification tasks. TE2Rules generates a rule list to explain positive class prediction by the model. We choose inTrees and deFragTrees as our baselines since they are also global explainers that generate a rule list from the tree ensemble model.

Method

For a tree ensemble (TE) model with n trees with max depth d, TE2Rules mines rules for the positive class (similar approach can be adopted for the negative class). It runs in two phases: rule generation and rule simplification. Here is a rough sketch of the two phases:

Rule generation: In the rule generation phase, the algorithm goes through the node combinations in n stages. In the k^{th} stage, we consider k (non-root) nodes from k different trees. Each node can be represented using a rule formed out of decision paths to reach the node from its root node. Along the same lines, the k node combination can be represented using conjunction of rules from each of those nodes. If such a rule from k nodes has support above a user defined threshold (default of 0), then we call it a *candidate* rule for stage k. Further, we assign positive class as the label for the candidate rule and check if the rule can be used to identify positive class prediction by the model with high precision. If the rule's precision is above a user defined threshold (default of 0.95), then we call it a *solution* rule. A default precision threshold of 0.95 (as against 1.00) helps keep the mined rules shorter (with less terms) and not overfit on the training data. We continue this process in stages till all positives in the training data are covered by some solution rule.

To generate the candidate rules for stage 1, we consider (non-root) nodes from each tree, which have non-zero support. Given the candidate rules from stage k, we can generate a stage $k+1$ candidate from a pair of stage k candidates that have $k-1$ nodes in common. By merging nodes from such a pair of k node combinations, we get a combination of $k+k-(k-1) = k+1$ nodes. If the resulting $k+1$ node combination has non-zero support, it can be a candidate rule for stage $k+1$.

Generating stage $k+1$ candidates in this way, we make sure that we do not explore $k+1$ node combinations which can be proven to have 0 support using the node combinations already explored in stage k. Let us say there exists a $k+1$ node combination that can not be generated by the above process. Then, consider the k node combinations formed out of the first k nodes and the last k nodes from the $k+1$ nodes. These two k node combinations share $k-1$ common nodes. If both these node combinations were present among stage k candidates, then they could be merged to form the $k+1$ nodes in stage $k+1$, by the above process. Hence, at least one of these two k node combinations do not exist among

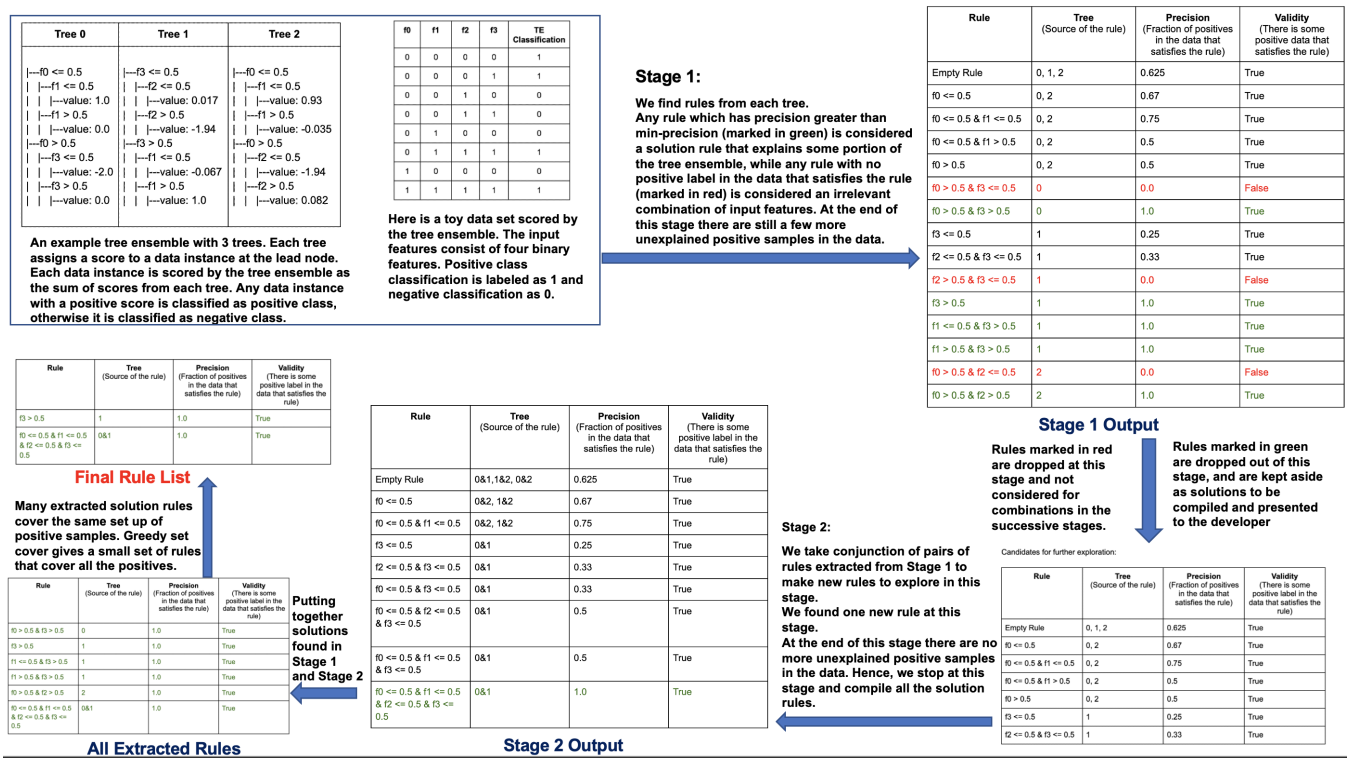

Figure 2: An example run of TE2Rules algorithm on a TE with 3 trees. The algorithm is run with min-precision of 1.0, extracting rules in stages and converging after explaining all the positive class predictions in the data. The final RL is simplified to have only a few rules that cover all the positive class predictions in the data.

stage k candidate rules. This implies that at least one of the two k node combinations has 0 support. Thus, any $k+1$ node combinations containing the k nodes would also have 0 support and is not worth considering in stage $k+1$. An example run of the TE2Rules algorithm on a tree ensemble with 3 trees can be found in Figure 2. For the pseudocode, refer Appendix.

The above method of generating stage $k+1$ candidates from stage k is inspired from Apriori Algorithm (Agrawal, Srikant et al. 1994) for associative rule mining. Apriori Algorithm exploits the Anti-Monotone Property: As a rule grows with more number of clauses, it's support in data decreases. Hence, for a rule with 0 support, it does not make sense to grow it any further by adding any additional clauses to it. Highly scalable implementations of Apriori have been discussed in the past using prefix and suffix hash tables to find the appropriate pairs of node combinations to merge in every stage.

To further speed up the candidate rule generation, we make a key observation: a lot of low precision candidate rules from stage k, can be very similar to each other. For example consider the rules: "age $>$ 35 and capital gain $>$ 2500 and marital status = married" and "age $>$ 36 and capital gain $>$ 2400 and marital status = married". These two rules end up covering almost the same set of data points in the training data. Such rules can occur because of some important feature occurring at the top of a lot of trees in the tree ensemble, or due to feature correlations which are hard for a human to detect but easy for the ML model to learn. When two such similar rules from stage k are combined together, it generates a similar rule (with similar support) in stage $k+1$. Since these rules did not meet the precision threshold in stage k, they are unlikely to meet the same in stage $k+1$. It is a wasteful exercise to combine such similar rules while generating "new" candidate rules for stage $k+1$. Moreover, combining them at every stage makes the number of such similar rules grow exponentially with stages.

To avoid combining similar rules, we use a set similarity metric (Mann, Augsten, and Bouros 2016) on the support set of the candidate rules to identify pairs which are highly similar to each other. We use Jaccard Score (defined as Jaccard(A, B) = $|A \cap B|/|A \cup B|$). We combine two candidate rules only when the jaccard score of the two rules is less than a small threshold (default of 0.2).

These insights help scale TE2Rules algorithm to run on tree ensemble models with hundreds of trees in a reasonable time relative to existing state of the art methods.

Rule simplification: After the generating the solution rules, the algorithm simplifies them. The solution rule list might have multiple rules covering the same data points. In the simplification phase, the algorithm attempts to find a minimum subset of rules from the solution rules that covers all the positives covered by the solution rules. This is the well known "set-cover" problem, and we use a greedy approach to solve it. We start by selecting the rule that covers the most number of positives. Then, we remove these positives from

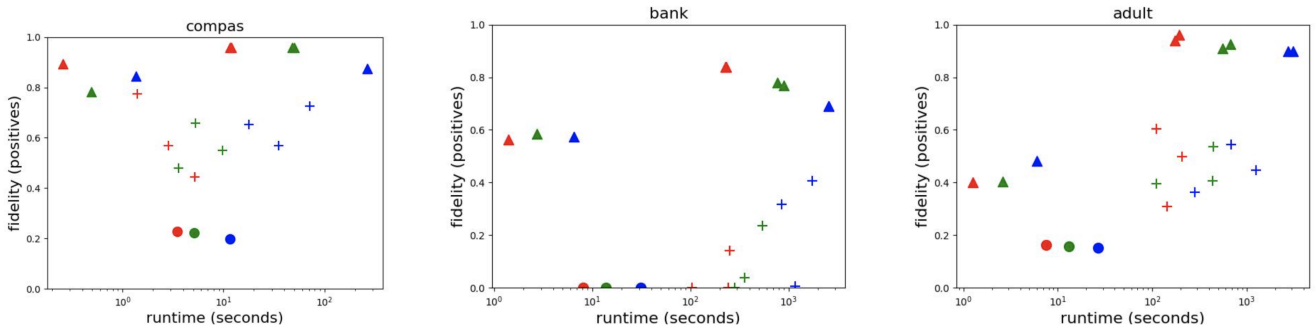

Figure 3: Comparison of fidelity (positives) on test data versus runtime for different explainers: TE2Rules (△), inTrees (○), deFragTrees (+). All explainers are run on TE models with 100 (red), 200 (green), 500 (blue) trees of depth 5.

the dataset and again the select the rule that has the highest coverage of positives that are left in the dataset. We repeat this process till no more positives are left in the dataset. We present the selected list of rules as an explanation to the user.

Example: Here is a short example of rule lists extracted by TE2Rules from a tree ensemble model trained with 50 trees, depth 3 on the *compas* dataset. The *compas* dataset consists of the results from a commercial algorithm used to assess a convicted criminal's likelihood of reoffending. Positive label denotes that the convicted criminal is likely to reoffend.

The TE2Rules algorithm was run until stage 3, achieving a fidelity of 94.3%, positive fidelity of 100%, negative fidelity of 90.4%. The rules mined by TE2Rules are shown in Table 1, in decreasing order of support. These rules summarize how the TE model classifies a convicted criminal as likely to reoffend.

We note that: (i) TE2Rules could achieve a positive fidelity of 100% in as little as 3 stages out of 50 (ii) the RL consists of 14 interpretable rules, all with 1-4 predicates; (iii) variables such as *age* and *priors_count* shows up frequently in the RL, which makes intuitive sense; (iv) some of the rules hint potential discrimination by the TE model. Example: One of the rules at the very top (high support rule) stating that "$age \leq 22.5$ & $priors\_count \leq 1.5$ & $race\_African\_American > 0.5$ & $sex\_Female \leq 0.5$" shows that the TE model considers young African_American males with just 1 or no priors_counts as likely to reoffend. This is discriminatory behavior against African_Americans, a known artifact in the compas dataset, that was learnt by the TE model. Some other age discriminatory rules (with low support) can be found at the bottom of the RL.

Results

Datasets: We demonstrate the effectiveness of TE2Rules using 3 datasets from domains (like finance, legal sector) where transparency in decision making process is crucial: *compas*, *bank*, *adult* (Larson et al. 2016; Dua and Graff 2017). The *compas* dataset consists of the results from a commercial algorithm used to assess a convicted criminal's likelihood of reoffending. The *bank* dataset consists of results from a marketing campaign by a banking institution on whether a client will subscribe to their term deposit. The *adult* dataset consists of census data on whether a person has income over 50K$. All these datasets contain demographic attributes of participants like age, gender, race, etc.

Rule	Label
IF age $\leq$ 36.5 & priors_count $>$ 2.5	positive
IF age $\leq$ 59.5 & priors_count $>$ 5.5	positive
IF age $\leq$ 21.5 & days_b_screening_arrest $\leq$ 17.5 & priors_count $\leq$ 12.5 & sex_Female $\leq$ 0.5	positive
IF age $\leq$ 27.5 & priors_count $>$ 1.5	positive
IF age $\leq$ 22.5 & priors_count $\leq$ 1.5 & race_African_American $>$ 0.5 & sex_Female $\leq$ 0.5	positive
IF age $\leq$ 23.5 & days_b_screening_arrest $>$ -2.5 & priors_count $\leq$ 8.5 & priors_count $>$ 0.5	positive
IF days_b_screening_arrest $>$ -1.5 & priors_count $>$ 5.5	positive
IF age $\leq$ 31.5 & c_charge_degree_F $>$ 0.5 & days_b_screening_arrest $>$ -4.5 & priors_count $>$ 1.5	positive
IF age $\leq$ 54.0 & days_b_screening_arrest $>$ 0.5 & race_Hispanic $\leq$ 0.5	positive
IF age $\leq$ 20.5 & c_charge_degree_F $\leq$ 0.5 & days_b_screening_arrest $\leq$ 0.5	positive
IF age $\leq$ 51.5 & priors_count $>$ 4.5	positive
IF days_b_screening_arrest $>$ 17.5 & priors_count $\leq$ 12.5	positive
IF age $\leq$ 19.5	positive
IF age $>$ 89.5 & priors_count $\leq$ 3.5	positive
ELSE	negative

Table 1: Rules generated by TE2Rules on a TE with 50 trees, depth 3, trained on compas dataset

Baselines: We compared TE2Rules with two popular baselines: inTrees and deFragTrees.

inTrees goes through each node in the tree ensemble and extracts rules from each node using the decision path to reach the node from its respective root node. For each rule extracted from a node, it assigns the majority label from the support of the rule. Further, it selects a small set of high precision rules and presents it in a falling rule list format.

deFragTrees identifies all possible rules that can be formed out of different node combinations from the tree ensemble. It then simplifies these rules by inducing a probability distribution over the rules and finding the simplest representation of this distribution. In this process, it finds a short falling rule list to represent the tree ensemble.

These algorithms operate at different ends of the spectrum. inTrees mines rules from individual nodes and completely discounts the effects of node combinations from multiple

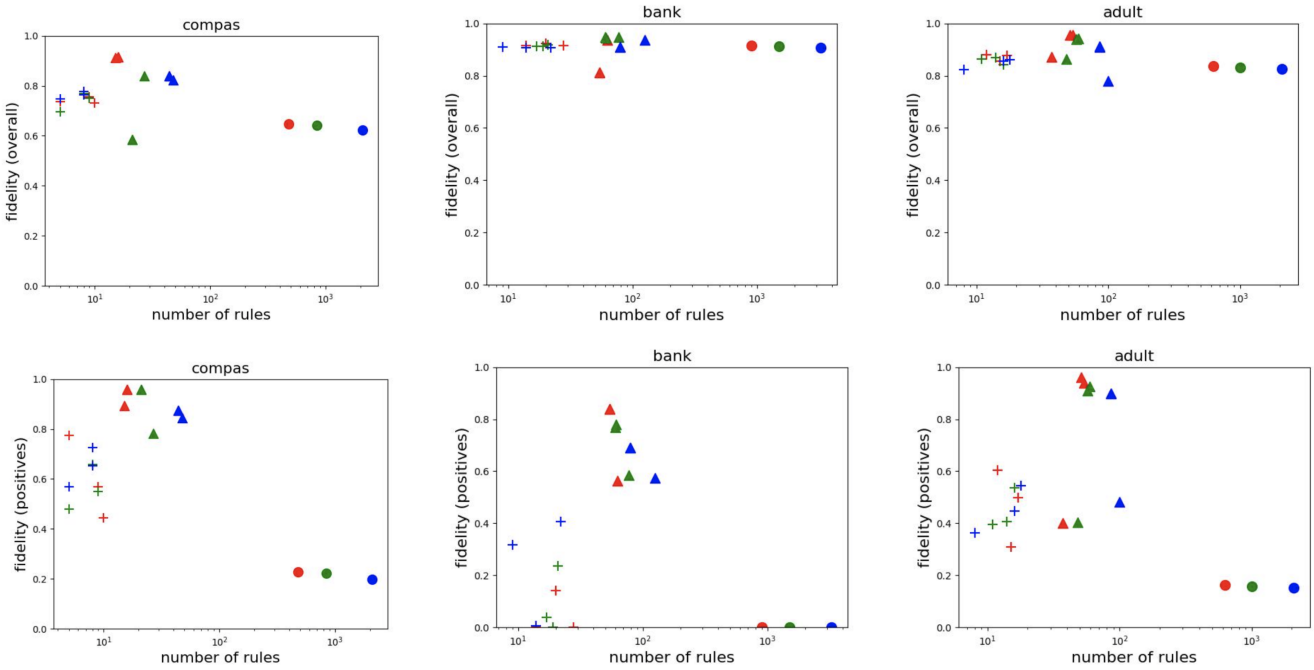

Figure 4: Comparison of fidelity on test dataset (overall and on positives) versus number of rules mined for different explainers: TE2Rules (△), inTrees (○), deFragTrees (+). All explainers are run on TE models with 100 (red), 200 (green), 500 (blue) trees of depth 5.

trees. deFragTrees mines rules by simplifying rules from all possible n node combinations from multiple trees. TE2Rules provides a middle ground of exploring rules in stages of k node combinations, with $k = 1, 2, 3, \ldots n$.

We report results of TE2Rules run with stages 1, 2 and 3. deFragTrees requires a parameter to specify the maximum number of rules to mine. We run deFragTrees with 1x, 5x and 10x times the number of rules mined by TE2Rules (stage-3). Both the baselines (and TE2Rules) take the trained model and a sample of training data (10%) as input to mine rules to explain the model. All explainers are run using the same sampled training dataset, trained model and evaluated using the same test dataset.

Implementation: TE2Rules and deFragTrees are implemented in python while inTrees is implemented in R. We trained our xgboost models in python using scikit-learn and exported them in a format that can be ingested in R. Our code to reproduce the results can be found in Appendix. All experiments were conducted on a 64-bit Ubuntu OS 20.04, with an Intel Xeon 2.4 GHz CPU and 32 GB RAM.

Models: We trained gradient boosted tree ensemble (TE) models with 100, 200, 500 trees with depth 3, 5 for binary classification in python scikit-learn. In all our results, we use red, green, blue colors to denote TE models with 100, 200, 500 trees, respectively. We explain these models using inTrees, deFragTrees and TE2Rules. We report the number of extracted rules and time taken to extract the rules. We evaluate the performance of the rules using fidelity: accuracy of the rules with respect to the model predictions. We report the fidelity of the rules on the test data (overall fidelity) and on the portion of test data on which the model labels it as positive class (positive fidelity). In all these datasets, positive class happens to be the minority class.

In Figure 3, for adult dataset, for each model (color), there are 3 "triangles" representing TE2Rules run till stages 1, 2 and 3 respectively. Among the 3 triangles of same color, the triangle with the least fidelity on positives corresponds to stage-1. The fidelity on positives improves with stages. However, in all our experiments, most of the positives in the training data sample could be covered by the rules in stage-2. Hence, stage-3 provides little improvement in fidelity on positives. Thus, the triangles for stage-3 often overlaps with that of stage-2. Example: compas, bank dataset in Figure 3.

Though triangles with higher fidelity on positives always represent later stages, the same cannot be said about number of rules. The number of rules always increase with stages in the "generation" phase. However, the number of rules in the final output of TE2Rules, includes both generation and simplification phase. Thus, it is possible for "stage-1 + simplification" to have slightly more number of rules than "stage-1 + stage-2 + simplification". This effect can be seen in Figure 4 (bottom 3 plots). Example: Among the blue triangles in bank dataset, the one with higher positive fidelity (stage-2) has lesser number of rules than the one with lower positive fidelity (stage-1).

Performance

Figure 4 shows the fidelity (on overall test set and test set with positive model prediction) of the explainers on xgboost models with 100, 200, 500 trees with depth 5. Each model is shown with a different color and it only makes sense to compare explainers (+, △, ○) within the same color.

In the top 3 plots of Figure 4, we note that all explainers achieve very high fidelity on the overall test set with

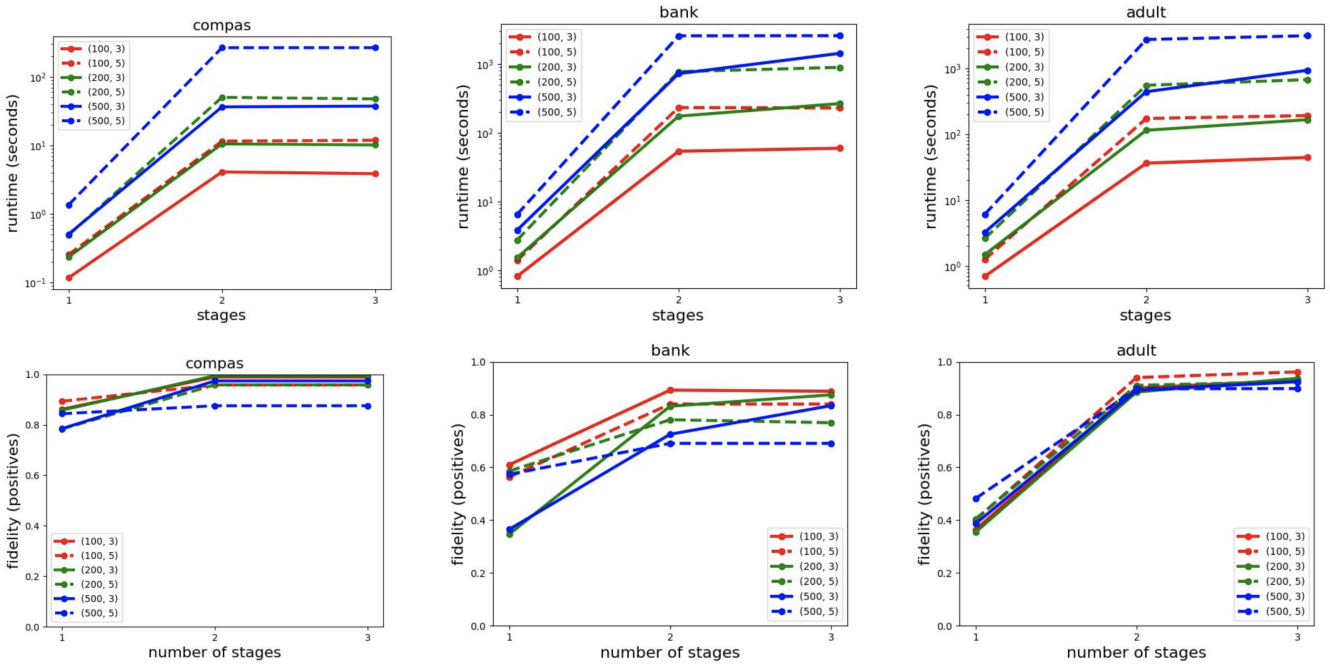

Figure 5: Comparison of runtime and fidelity (positives) of TE2Rules with more number of stages. TE2Rules is run on TE models with 100 (red), 200 (green), 500 (blue) trees of depth 3 (solid line), 5 (dashed line).

deFragTrees achieving it with the least number of rules and inTrees with the most number of rules. In the bottom 3 plots, we note that the baselines have poor fidelity on the portion of the test set with positive model predictions, with deFragTrees performing better than inTrees.

TE2Rules is able to achieve high fidelity on the positive model predictions because it combines rules from multiple trees. inTrees only fetches rules from individual trees. deFragTrees performs better since it tries to mine rules from the global model prediction boundaries. Stage-1 of TE2Rules is closer to inTrees in principle, since it also mines rules from individual nodes. However, TE2Rules outperforms inTrees since the falling rule list mined by inTrees does not mine enough rules for the minority class (positives). Most rules mined by inTrees only explain the majority class (negatives). This leads to poor performance in fidelity on positives. deFragTrees also suffers from this same problem. Inspite of mining 10 times as many rules as TE2Rules, deFragTrees could not explain the minority class effectively compared to TE2Rules stage-3.

Scalability

Figure 3 shows the run time of the explainers for the 3 xgboost models: 100, 200, 500 trees with depth 5. We can notice two clusters of $\triangle$ points corresponding to TE2Rules: one on the left (lower runtime) and the other on the top right corner (higher runtime, higher fidelity on positives) of the plot. The first cluster corresponds to TE2Rules stage-1 while the other corresponds to TE2Rules stage-2 and 3. Triangles corresponding to stage-2 and 3 might overlap, as explained in previous subsection. We observe that in stage-1, TE2Rules already beats inTrees and has comparable performance to deFragTrees with a relatively small run time. As we run more stages (2 or 3) with TE2Rules, its performance beats deFragTrees with runtimes comparable to deFragTrees.

Fidelity-Runtime tradeoff

Figure 5 shows the run time and positive fidelity of TE2Rules with stages for 6 different xgboost models with 100, 200, 500 trees and depth 3, 5. We note that there is marginal improvement in fidelity beyond stage-2. So for all our use cases, running TE2Rules for 3 stages was sufficient. A brute force search across node combinations would have meant exploring exponentially more nodes in every stage. But due to the smart way of generating stage-3 candidates from stage-2, very few (almost no) candidates with non-zero support are generated in stage-3. This reduces the runtime of stage-3 significantly. Since, most of the rules are mined in early stages, TE2Rules can be stopped early (2 to 3 stages) without loosing much fidelity on positives.

Conclusion

We presented a novel approach, TE2Rules to explain a binary tree ensemble classifier using rules mined specially for a class of interest. We showed that our explainer is faithful (with high fidelity) to the model on both the overall test data and specifically on the minority class. It achieves such high performance in runtimes that are comparable to the state of the art baselines. Further, we show that stopping the algorithm in early stages can tradeoff runtime without loosing much fidelity on positives.

References

Agrawal, R.; Srikant, R.; et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, 487–499. Citeseer.

Deng, H. 2019. Interpreting tree ensembles with intrees. *International Journal of Data Science and Analytics*, 7(4): 277–287.

Dua, D.; and Graff, C. 2017. UCI Machine Learning Repository.

Friedman, J. H.; and Popescu, B. E. 2008. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3): 916 – 954.

Gautier, R.; Jafre, G.; and Ndiaye, B. 2020. scikit-learn-contrib/skope-rules. https://github.com/scikit-learn-contrib/skope-rules. V1.0.1.

Grinsztajn, L.; Oyallon, E.; and Varoquaux, G. 2022. Why do tree-based models still outperform deep learning on tabular data? arXiv:2207.08815.

Guidotti, R.; Monreale, A.; Ruggieri, S.; Pedreschi, D.; Turini, F.; and Giannotti, F. 2018. Local Rule-Based Explanations of Black Box Decision Systems. *CoRR*, abs/1805.10820.

Hara, S.; and Hayashi, K. 2018. Making tree ensembles interpretable: A bayesian model selection approach. In *International conference on artificial intelligence and statistics*, 77–85. PMLR.

Larson, J.; Mattu, S.; Kirchner, L.; and Angwin, J. 2016. How we analyzed the COMPAS recidivism algorithm. ProPublica.

Mann, W.; Augsten, N.; and Bouros, P. 2016. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment*, 9(9): 636–647.

Meinshausen, N. 2010. Node harvest. *The Annals of Applied Statistics*, 4(4).

Qin, Z.; Yan, L.; Zhuang, H.; Tay, Y.; Pasumarthi, R. K.; Wang, X.; Bendersky, M.; and Najork, M. 2021. Are Neural Rankers still Outperformed by Gradient Boosted Decision Trees? In *International Conference on Learning Representations*.

Ribeiro, M. T.; Singh, S.; and Guestrin, C. 2018. Anchors: High-Precision Model-Agnostic Explanations. In *AAAI*.

Selvaraju, R. R.; Cogswell, M.; Das, A.; Vedantam, R.; Parikh, D.; and Batra, D. 2019. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization. *International Journal of Computer Vision*, 128(2): 336–359.

Sundararajan, M.; Taly, A.; and Yan, Q. 2017. Axiomatic Attribution for Deep Networks. arXiv:1703.01365.

Vidal, T.; Pacheco, T.; and Schiffer, M. 2020. Born-Again Tree Ensembles. arXiv:2003.11132.

Wang, F.; and Rudin, C. 2015. Falling rule lists. In *Artificial intelligence and statistics*, 1013–1022. PMLR.

Appendix

Reproducibility

Here is our code implementing TE2Rules: https://github.com/linkedin/TE2Rules. The code has a Readme file with instructions on how to reproduce the plots presented in this paper.

Psudocode of TE2Rules

In this section we present a pseudocode of TE2Rules. Algorithm 1 shows the overall sketch of TE2Rules. It makes references to various helper methods. Pseudocode of the helper functions are also given below. TE2Rules algorithm runs in two phase: Rule Generation and Rule Simplification.

Algorithm 1: TE2Rules

$solutions \leftarrow []$

▷ Rule Generation

for $k \leftarrow 1, 2, 3, \ldots n$ **do**
  if $k = 1$ **then**
    $candidates \leftarrow getNodeRules(model)$
  else
    $candidates \leftarrow getNextStage(candidates, k)$
  end if

  for $r \leftarrow candidates$ **do**
    $p \leftarrow getPrecision(r \implies positiveLabel)$
    if $p > 1 - \delta$ **then**
      $candidates.remove(r)$
      $solutions.append(r)$
    end if
  end for
end for

▷ Rule Simplification

$solutions \leftarrow greedySetCover(solutions)$
return $solutions$

The Rule Generation phase generates candidate rules for each stage and mines solution rules out of the candidates. For each candidate rule, we check if the rule formed by $candidaterule \implies positive$ agrees with the model prediction with high precision. Any candidate rule that has precision above $1 - \delta$ is promoted to be a solution rule. However, the generation phase can produce a lot of such solution rules. The Simplification stage selects a small subset of the solution rules that are effective in explaining the TE model and are easy for a human to comprehend with less number of rules.

TE2Rules proceeds in stages and in stage-k, it explores all rules formed out of k-node combinations from the tree ensemble. For generating candidate rules, we use two helper methods: $getNodeRules()$ for the TE model for stage-1 candidates and $getNextStage()$ for any stage-k candidates (for all k ¿ 1). $getNodeRules()$ is described in Algorithm 2 and $getNextStage()$ is described in Algorithm 3. $getNodeRules()$ goes through every node in the tree ensemble and spits out the rule corresponding to the decision path to reach that node from its respective root node. Thus, it finds all individual nodes (or 1-node) combinations from the tree ensemble. It the filters out all rules which has very little

Algorithm 2: getNodeRules(model)

```
candidates ← []

for tree ← model do
    for node ← tree do
        r ← node.getDecisionPath()
        r.sourceNodes ← [node]
        if |r.support| > ϵ then
            candidates.append(r)
        end if
    end for
end for
return candidates
```

(less than $epsilon$) support in data. $getNextStage()$ uses the candidates found in the previous stage to efficiently find the next stage candidates. It considers all pairs of candidates from stage-$k-1$ such that they share $k-2$ nodes in common. When such a pair of rules is combined (using logical and), they get a total of $(k-1)+(k-1)-(k-2)=k$ nodes as their source. It then filters out all rules which has very little (less than $epsilon$) support in data.

Algorithm 3: getNextStage(candidates, k)

```
newCandidates ← []

for r1 ← candidates do
    for r2 ← candidates do
        nodes1 = r1.sourceNodes
        nodes2 = r2.sourceNodes
        if |nodes1 ∩ nodes2| = k − 2 then
            r ← r1&r2
            r.sourceNodes ← n1 ∪ n2
            if |r.support| > ϵ then
                candidates.append(r)
            end if
        end if
    end for
end for
return newCandidates
```

Generating stage k candidates in this way, we make sure that we do not explore k node combinations which can be proven to have little support ($< \epsilon$) using the node combinations already explored in stage $k-1$. Let us say there exists a k node combination that can not be generated by the above process. Then, consider the $k-1$ node combinations formed out of the first $k-1$ nodes and the last $k-1$ nodes from the k nodes. These two $k-1$ node combinations share $k-2$ common nodes. If both these node combinations were present among stage $k-1$ candidates, then they could be merged to form the k nodes in stage k, by the above process. Hence, at least one of these two $k-1$ node combinations do not exist among stage $k-1$ candidate rules. This implies that at least one of the two $k-1$ node combinations has support $< \epsilon$. Thus, any k node combinations containing the $k-1$ nodes would also have support $< \epsilon$ and is not worth considering in stage k.

Algorithm 4: greedySetCover(solutions)

```
newSolutions ← []
cover ← ϕ

while cover ≠ allPositives do
    r ← maxPositiveCoverage(solutions)
    solutions.remove(r)
    newSolutions.append(r)
    cover ← cover ∪ r.coveredPositives
end while
return newSolutions
```

The generation phase is run till we cover all positives in the data using some solution rule or we exhaust all stages $k = 1, 2, 3, \ldots, n$ (whichever is reached first). The simplifiaction phase uses a greedy set cover to find a small subset of solution rules that still explain all the positives in the data. The greedy set cover algorithm is described in Algorithm 4. We start by selecting the rule that covers the most number of positives. Then, we remove these positives from the dataset and again the select the rule that has the highest coverage of positives that are left in the dataset. We repeat this process till no more positives are left in the dataset. We present the selected list of rules as an explanation to the user.