Fast TreeSHAP: Accelerating SHAP Value Computation for Trees

Anonymous Author(s)
Affiliation
Address
email

Abstract

SHAP (SHapley Additive exPlanation) values are one of the leading tools for interpreting machine learning models, with strong theoretical guarantees (consistency, local accuracy) and a wide availability of implementations and use cases. Even though computing SHAP values takes exponential time in general, TreeSHAP takes polynomial time on tree-based models. While the speedup is significant, TreeSHAP can still dominate the computation time of industry-level machine learning solutions on datasets with millions or more entries, causing delays in post-hoc model diagnosis and interpretation service. In this paper we present two new algorithms, Fast TreeSHAP v1 and v2, designed to improve the computational efficiency of TreeSHAP for large datasets. We empirically find that Fast TreeSHAP v1 is 1.5x faster than TreeSHAP while keeping the memory cost unchanged. Similarly, Fast TreeSHAP v2 is 2.5x faster than TreeSHAP, at the cost of a slightly higher memory usage, thanks to the pre-computation of expensive TreeSHAP steps. We also show that Fast TreeSHAP v2 is well-suited for multi-time model interpretations, resulting in as high as 3x faster explanation of newly incoming samples.

1 Introduction

Predictive machine learning models are almost everywhere in industry, and complex models such as random forest, gradient boosted trees, and deep neural networks are being widely used due to their high prediction accuracy. However, interpreting predictions from these complex models remains an important challenge, and many times the interpretations at individual sample level are of the most interest [27]. There exist several state-of-the-art sample-level model interpretation approaches, e.g., SHAP [19], LIME [27], and Integrated Gradient [30]. Among them, SHAP (SHapley Additive exPlanation) calculates SHAP values quantifying the contribution of each feature to the model prediction, by incorporating concepts from game theory and local explanations. In contrast to other approaches, SHAP has been justified as the only consistent feature attribution approach with several unique properties which agree with human intuition. Due to its solid theoretical guarantees, SHAP becomes one of the top recommendations of model interpretation approaches in industry [20, 33, 2, 3].

There exist several variants of SHAP. The general version is KernelSHAP [19], which is model-agnostic, and generally takes exponential time to compute the exact SHAP values. Its variants include TreeSHAP [18, 17] and DeepSHAP [19], which are designed specifically for tree-based models (e.g., decision tree, random forest, gradient boosted trees) and neural-network models. In these variants, the special model structures may lead to potential improvements in computational efficiency. For example, the tree structure enables a polynomial time complexity for TreeSHAP. In this paper, we choose TreeSHAP for further exploration, as tree-based models are widespread in industry.

After looking into many TreeSHAP use cases, we find out that despite its algorithmic complexity improvement, computing SHAP values for large sample size (e.g., tens of millions of samples) or

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

large model size (e.g., maximum tree depth ≥ 10) still remains a computational concern in practice. For example, explaining 20 million samples for a random forest model with 400 trees and maximum tree depth 12 can take as long as 15 hours even on a 100-core server (more details in Appendix A.9). In fact, explaining (at least) tens of millions of samples widely exist in industry-level predictive models, e.g., feed ranking model, ads targeting model, and subscription propensity model. Spending tens of hours in model interpretation becomes a significant bottleneck in these modeling pipelines, causing huge delays in post-hoc model diagnosis via important feature analysis, as well as long waiting time in preparing feature reasoning for model end users.

In this paper, we conduct a thorough inspection into the TreeSHAP algorithm, with the focus on improving its computational efficiency for large size of samples to be explained. In this, we take the number of samples into consideration in our time complexity analysis of TreeSHAP algorithm. We propose two new algorithms - Fast TreeSHAP v1 and Fast TreeSHAP v2. Fast TreeSHAP v1 is built based on TreeSHAP by redesigning the ways of calculating certain computationally-expensive components. In practice, Fast TreeSHAP v1 is able to consistently boost the computational speed by $\sim$1.5x with the same memory cost as in TreeSHAP. Fast TreeSHAP v2 is built based on Fast TreeSHAP v1 by further pre-computing certain computationally-expensive components which are independent with the samples to be explained. It is able to largely reduce the time complexity when calculating SHAP values, leading to 2-3x faster computational speed than TreeSHAP in practice with just a small sacrifice on memory. The pre-computation step in Fast TreeSHAP v2 enables its suitability in multi-time model interpretations, where the model is pre-trained and remains unchanged, and new scoring data are coming on a regular basis.

2 Related Works

Since the introductions of SHAP [19] and TreeSHAP [18, 17], many related works have been done in this area. Most of them focus on the application side in a variety of areas, including medical science [20, 33, 2], social science [4], finance [3] and sports [28]. There also exist a lot of exciting papers focusing on the designs of SHAP/TreeSHAP implementations [31, 26, 11, 32], as well as the theoretical justification of SHAP values [10, 16, 29, 1, 6, 22, 5].

Besides these works, only a few of them focus on the computational efficiency of SHAP/TreeSHAP: The authors of [7] developed a Python package `fastshap` to approximate SHAP values for arbitrary models by calling scoring function as few times as possible. In [8], a model-agnostic version of SHAP value approximation was implemented on Spark. Both works were not specifically designed for tree-based models, and thus the advanced polynomial time complexity may not be leveraged. The authors of [21, 15] built R packages `shapper` and `treeshap` as R wrappers of SHAP Python library, which achieved comparable speed. However, no algorithmic improvements have been made. In [23, 24], MASHAP was proposed to compute SHAP values for arbitrary models in an efficient way, where an arbitrary model is first approximated by a surrogate XGBoost model, and TreeSHAP is then applied on this surrogate model to calculate SHAP values. The most related work of improving computational efficiency in TreeSHAP as far as we know is [25], where the authors presented GPUTreeShap as a GPU implementation of TreeSHAP algorithm. Our work is different from GPUTreeShap as our work focuses on improving the computational complexity of the algorithm itself, while the parallel mechanism of GPU rather than the improvement of algorithm complexity has led to the speedup of GPUTreeShap. In our future work, it is possible to further combine Fast TreeSHAP v1 and v2 with GPUTreeShap.

3 Background

In this section, we review the definition of SHAP values [19], and then walk through the derivation of TreeSHAP algorithm [18, 17] which directly leads to the development of Fast TreeSHAP v1 and v2.

3.1 SHAP Values

Let f be the predictive model to be explained, and N be the set of all input features. f maps an input feature vector $x \in \mathbb{R}^{|N|}$ to an output $f(x)$ where $f(x) \in \mathbb{R}$ in regression and $f(x) \in [0, 1]$ in (binary) classification. SHAP values are defined as the coefficients of the additive surrogate explanation model: $g(z') = \phi_0 + \sum_{i \in N} \phi_i z'_i$. Here $z' \in \{0, 1\}^{|N|}$ represent a feature being observed ($z'_i = 1$) or

unknown ($z'_i = 0$), and $\phi_i \in \mathbb{R}$ are the feature attribution values. Note that the surrogate explanation model $g(z')$ is a local approximation to the model prediction $f(x)$ given an input feature vector x.

As described in [19], SHAP values are the single unique solution of ϕ_i's in the class of additive surrogate explanation models $g(z')$ that satisfies three desirable properties: local accuracy, missingness, and consistency. To compute SHAP values we denote $f_S(x)$ as the model output restricted to the feature subset $S \subset N$, and the SHAP values are then computed based on the classic Shapley values:

$$\phi_i = \sum_{S \subset N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} (f_{S \cup \{i\}}(x) - f_S(x)). \tag{1}$$

It is still remaining to define $f_S(x)$. In recent literature, there exist two main options for defining $f_S(x)$: the conditional expectation $f_S(x) = E[f(x)|x_S] = E_{x_{\bar{S}}|x_S}[f(x)]$, and the marginal expectation $f_S(x) = E_{x_{\bar{S}}}[f(x)]$, where x_S is the sub-vector of x restricted to the feature subset S, and $\bar{S} = N \setminus S$. While it is still debating which option is more appropriate [10, 16, 29, 1, 6, 22, 5], both options need exponential time in computation as Equation 1 considers all possible subsets in N.

3.2 SHAP Values for Trees

For a tree-based model f, we note that it is sufficient to investigate the ways to calculate SHAP values on a single tree, since the SHAP values of tree ensembles equal the sums of SHAP values of its individual trees according to the additivity property of SHAP values [18, 17]. The authors of [18, 17] define a conditional expectation $f_S(x) = E[f(x)|x_S]$ for tree-based models. The basic idea is to calculate $f_S(x)$ by recursively following the decision path for x if the split feature in the decision path is in S, and taking the weighted average of both branches if the split feature is not in S. We use the proportions of training data that flow down the left and right branches as the weights.

Algorithm 1 proposed in [18, 17] (Appendix A.2) provides the details to calculate $f_S(x)$ for tree-based models. A tree is specified as a tuple of six vectors $\{v, a, b, t, r, d\}$: v contains the values of each leaf node. a and b represent the left and right node indexes for each internal node. t contains the thresholds for each internal node, d contains the feature indexes used for splitting in internal nodes, and r represents the cover of each node (i.e., how many training samples fall in that node). The time complexity of calculating $f_S(x)$ in Algorithm 1 is $O(L)$, where L is the number of leaves in a tree, since we need to loop over each node in the tree. This leads to an exponential complexity of $O(MTL2^{|N|})$ for computing SHAP values for M samples with a total of T trees. We next show how TreeSHAP can help reduce this time complexity from exponential to polynomial.

3.3 TreeSHAP

TreeSHAP proposed in [18, 17] runs in $O(MTLD^2)$ time and $O(D^2 + |N|)$ memory, where M is the total number of samples to be explained, $|N|$ is the number of features, T is the number of trees, D is the maximum depth of any tree, and L is the maximum number of leaves in any tree. The intuition of the polynomial time algorithm is to recursively keep track of the proportion of all possible feature subsets that flow down into each leaf node of the tree, which is similar to running Algorithm 1 simultaneously for all $2^{|N|}$ feature subsets. We recommend readers to check [18, 17] for the algorithm details. In this paper, we present our understanding of the derivation of the TreeSHAP algorithm, which also leads to our proposed Fast TreeSHAP algorithms with computational advantage.

We introduce some additional notations. Assume in a given tree $\{v, a, b, t, r, d\}$, there are K leaves with values v_1 to v_K, corresponding to K paths P_1 to P_K, where path P_k is the set of internal nodes starting from the root node and ending at the leaf node v_k (leaf node v_k is not included in the path). We use $j \in P_k$ to denote the jth internal node in path P_k hereafter. We also use $D_k = \{d_j : j \in P_k\}$ to denote the feature set used for splitting in internal nodes in path P_k. Moreover, we denote the feature subspaces restricted by the thresholds along the path P_k as $\{T_{kj} : j \in P_k\}$, where $T_{kj} = (-\infty, t_{kj}]$ if node j connects to its left child in path P_k or $T_{kj} = (t_{kj}, \infty)$ if node j connects to its right child in path P_k ($\{t_{kj} : j \in P_k\}$ are the thresholds for internal nodes in path P_k), and we also denote the covering ratios along the path P_k as $\{R_{kj} : j \in P_k\}$, where $R_{kj} = r_{k,j+1}/r_{kj}$ ($\{r_{kj} : j \in P_k\}$ are the covers of internal nodes in path P_k). The formula of $f_S(x)$ in Algorithm 1 can be simplified as:

$$f_S(x) = \sum_{k=1}^{K} \prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} \cdot \prod_{j \in P_k, d_j \notin S} R_{kj} \cdot v_k,$$

where $\mathbb{1}\{\cdot\}$ is an indicator function. Let $W_{k,S} := \prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} \cdot \prod_{j \in P_k, d_j \notin S} R_{kj}$, i.e., $W_{k,S}$ is the "proportion" of subset S that flows down into the leaf node v_k, then $f_S(x) = \sum_{k=1}^{K} W_{k,S} v_k$. Plugging it into Equation 1 leads to the SHAP value $\phi_i = \sum_{S \subset N \setminus \{i\}} \frac{|S|!(|N|-|S|-1)!}{|N|!} \sum_{k=1}^{K} (W_{k,S \cup \{i\}} - W_{k,S}) v_k$.

Theorem 1. *ϕ_i can be computed by only considering the paths which contain feature i and all the subsets within these paths (i.e., instead of considering all $2^{|N|}$ subsets). Specifically,*

$$\phi_i = \sum_{k: i \in D_k} [\sum_{m=0}^{|D_k|-1} \frac{m!(|D_k|-m-1)!}{|D_k|!} \sum_{S \subset D_k \setminus \{i\}, |S|=m} (\prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} \cdot \prod_{j \in P_k, d_j \in D_k \setminus (S \cup \{i\})} R_{kj})] \cdot (\prod_{j \in P_k, d_j = i} \mathbb{1}\{x_{d_j} \in T_{kj}\} - \prod_{j \in P_k, d_j = i} R_{kj}) v_k, \tag{2}$$

The proof of Theorem 1 can be found in Appendix A.3. For convenience, we call $\frac{m!(|D_k|-m)!}{(|D_k|+1)!}$ the "Shapley weight" for subset size m and path P_k. We point out that the TreeSHAP algorithm is exactly built based on Theorem 1. Specifically, the EXTEND method in TreeSHAP alogithm keeps track of the sum of the "proportion"s of all subsets that flow down into a certain leaf node weighted by its Shapley weight for each possible subset size. When descending a path, EXTEND is called repeatedly to take a new feature in the path and add its contribution to the sum of the "proportion"s of all feature subsets of size $1, 2, \cdots$ up to the current depth. At the leaf node, EXTEND reaches the sequence of values $\frac{m!(|D_k|-m)!}{(|D_k|+1)!} \sum_{S \subset D_k, |S|=m} (\prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} \prod_{j \in P_k, d_j \in D_k \setminus S} R_{kj})$ for each possible subset size $m = 1, \cdots, |D_k|$. The UNWIND method in TreeSHAP algorithm is used to undo a previous call to EXTEND, i.e., to remove the contribution of a feature previously added via EXTEND, and is indeed commutative with EXTEND. UNWIND can be used when duplicated features are encountered in the path, or at the leaf node when calculating the contribution of each feature in the path to SHAP values according to Equation 2.

4 Fast TreeSHAP

We now further simplify Equation 2 for an even faster computational speed. Consider a subset $S_k \subset D_k$ which consists of all the features in D_k not satisfying the thresholds along the path P_k, i.e., $S_k = \{d_j : j \in P_k, x_{d_j} \notin T_{kj}\}$. Also define $U_{m,D_k,C} := \frac{m!(|D_k|-m)!}{(|D_k|+1)!} \sum_{S \subset C, |S|=m} \prod_{j \in P_k, d_j \in C \setminus S} R_{kj}$ for $m = 0, \cdots, |C|$, where $C \subset D_k$ is a subset of D_k. We see that when $C = D_k \setminus S_k$, $U_{m,D_k,D_k \setminus S_k}$ can be interpreted as the sum of the "proportion"s of all subsets that flow down into a leaf node v_k (each feature in the subsets must satisfy the thresholds along the path P_k) weighted by its Shapley weight for subset size m. Finally, we define $U_{D_k,C} := \sum_{m=0}^{|C|} \frac{|D_k|+1}{|D_k|-m} U_{m,D_k,C}$. Theorem 2 further simplifies Equation 2 (proof in Appendix A.4):

Theorem 2. *ϕ_i can be computed by only considering the subsets within the paths where each feature in the subsets satisfies the thresholds along the path. Specifically,*

$$\phi_i = \begin{cases} -\sum_{k: i \in D_k} \left[U_{D_k, D_k \setminus S_k} \prod_{j \in P_k, d_j \in S_k} R_{kj} \right] v_k & if\ i \in S_k, \\ \sum_{k: i \in D_k} \left[U_{D_k, D_k \setminus (\{i\} \cup S_k)} \prod_{j \in P_k, d_j \in S_k} R_{kj} (1 - \prod_{j \in P_k, d_j = i} R_{kj}) \right] v_k & if\ i \notin S_k. \end{cases} \tag{3}$$

4.1 Fast TreeSHAP v1 Algorithm

We propose Fast TreeSHAP v1 algorithm based on Theorem 2 which runs in $O(MTLD^2)$ time and $O(D^2 + |N|)$ memory. This computational complexity looks the same as in the original TreeSHAP. However, we will show in section 4.1.1 and 5 that the average running time can be largely reduced.

In Fast TreeSHAP v1 algorithm (Algorithm 2 in Appendix A.5), we follow the similar algorithm setting as in the original TreeSHAP algorithm, where both EXTEND and UNWIND methods are being used. The EXTEND method is used to keep track of $U_{m,D_k,D_k \setminus S_k}$ for $m = 0, \cdots, |D_k| - |S_k|$. Remind that $U_{m,D_k,D_k \setminus S_k}$ is the sum of the "proportion"s of all subsets that flow down into a leaf node v_k **(each feature in the subsets must satisfy the thresholds along the path** P_k**)** weighted by

its Shapley weight for subset size m. Compared with the EXTEND method in the original TreeSHAP algorithm, the main difference is the constraint applied on these subsets (highlighted in bold), which largely reduces the number of subset sizes to be considered. Specifically, in Fast TreeSHAP v1, when descending a path, EXTEND is called only when a new feature in the path satisfies the threshold, and then its contribution to the sum of "proportion"s of all feature subsets of size $1, 2, \cdots$ up to the number of features satisfying the thresholds until the current depth is added. When reaching the leaf node, the number of possible subset sizes considered by EXTEND is $|D_k| - |S_k|$ in Fast TreeSHAP v1 rather than $|D_k|$ in the original TreeSHAP. The UNWIND method is still used to undo a previous call to EXTEND. Specifically, it is used when duplicated features are encountered in the path or when calculating $U_{m,D_k,D_k\setminus(\{i\}\cup S_k)}$ for $i \in D_k\setminus S_k$, $m = 0, \cdots, |D_k| - |S_k| - 1$ at the leaf node. Besides EXTEND and UNWIND, we also keep track of the product of covering ratios of all features not satisfying the thresholds along the path, i.e., $\prod_{j\in P_k, d_j\in S_k} R_{kj}$ in Equation 3, which is trivial.

4.1.1 Complexity Analysis

In the original TreeSHAP, the complexity of EXTEND and UNWIND is bounded by $O(D)$, since both of them need to loop over the number of possible subset sizes, which equals $|D_k|$ in path P_k. At each internal node, EXTEND is called once, while at each leaf node, UNWIND is called D times to update SHAP values for each of the D features in the path. This leads to a complexity of $O(LD^2)$ for the entire tree because the work done at the leaves dominates the work at the internal nodes.

In Fast TreeSHAP v1, both EXTEND and UNWIND need to loop over the number of possible subset sizes under the constraint on subset (highlighted in bold), which is $|D_k| - |S_k|$ in path P_k. Thus, although the complexity of EXTEND and UNWIND is still bounded by $O(D)$, the average running time can be reduced to 50%, which equals the average ratio between the number of possible subset sizes under constraint and the number of all possible subset sizes, i.e., $E[(|D_k| - |S_k|)/|D_k|] \approx \sum_{i=0}^{D} C_D^i (D-i)/(2^D D) \approx 50\%$. Moreover, according to Equation 3, at the leaf node of Path P_k, UNWIND is called $|D_k| - |S_k|$ times for each of the $|D_k| - |S_k|$ features satisfying the thresholds in the path, and only once for all other features in the path. Therefore, although the number of times UNWIND being called is still bounded by $O(D)$, the actual number can also be lowered by 50% on average. As a result, although we still have the complexity of $O(LD^2)$ for the entire tree, the average running time can be reduced to $50\% \times 50\% = 25\%$ compared with the original TreeSHAP. Finally, the complexity is $O(MTLD^2)$ for the entire ensemble of T trees and M samples to be explained, with the running time reduced to 25% on average compared with the original TreeSHAP.

4.2 Fast TreeSHAP v2 Algorithm

We propose Fast TreeSHAP v2 algorithm that runs in $O(TL2^D D + MTLD)$ time and $O(L2^D)$ memory. For balanced trees it becomes $O(TL^2 D + MTLD)$ time and $O(L^2)$ memory. Compared with $O(MTLD^2)$ time and $O(D^2 + |N|)$ memory in original TreeSHAP and Fast TreeSHAP v1, Fast TreeSHAP v2 outperforms in computational speed when the number of samples M exceeds $2^{D+1}/D$, where D is the maximum depth of any tree (more details in the next paragraph). Fast TreeSHAP v2 has a stricter restriction on tree size due to memory concerns. In practice, it works well for trees with maximum depth as large as 16 in an ordinary laptop, which covers most of the use cases of tree-based models. We discuss the memory concerns in detail in Sections 4.3 and 5.

The design of Fast TreeSHAP v2 algorithm is inspired by Fast TreeSHAP v1 algorithm. Recall that the loops of UNWIND at the leaves dominate the complexity of Fast TreeSHAP v1 algorithm, where the length of each loop is $O(D)$, and each call to UNWIND also takes $O(D)$ time, resulting in $O(D^2)$ time complexity at each leaf node. While looping over each of the D features in the path is inevitable in updating SHAP values at the leaf node (i.e., the loop of length $O(D)$ is necessary), our question is: Is it possible to get rid of calling UNWIND? From Equation 3 we see that, the ultimate goal of calling UNWIND is to calculate $U_{D_k,D_k\setminus S_k}$ and $U_{D_k,D_k\setminus(\{i\}\cup S_k)}$ for $i \in D_k\setminus S_k$. We also note that for different samples to be explained, although S_k may vary from sample to sample, all the possible values of $U_{D_k,D_k\setminus S_k}$ and $U_{D_k,D_k\setminus(\{i\}\cup S_k)}$ for $i \in D_k\setminus S_k$ fall in the set $\{U_{D_k,C} : C \subset D_k\}$ with size 2^{D_k}. Therefore, a natural idea to reduce the computational complexity is, instead of calling UNWIND to calculate $U_{D_k,D_k\setminus S_k}$ and $U_{D_k,D_k\setminus(\{i\}\cup S_k)}$ for $i \in D_k\setminus S_k$ every time we explain a sample, we can pre-compute all the values in the set $\{U_{D_k,C} : C \subset D_k\}$ which only depend on the tree itself, and then extract the corresponding value when looping over features at leaf nodes to calculate ϕ_i for each specific sample to be explained. In fact, what we just proposed is to trade space

complexity for time complexity. This should significantly save computational efforts when there exist redundant computations of $U_{D_k,C}$ across samples, which generally happens when $M > 2^{D+1}/D$ (For each sample, around $D/2$ $U_{D_k,C}$'s should be calculated for each path, thus on average $MD/2$ calculations should be taken for M samples). This commonly occurs in a moderate-sized dataset, e.g., $M > 22$ when $D = 6$, $M > 205$ when $D = 10$, and $M > 2341$ when $D = 14$. We show the appropriateness of trading space complexity for time complexity in practice in Section 5.

We split Fast TreeSHAP v2 algorithm into two parts: **Fast TreeSHAP Prep** and **Fast TreeSHAP Score**. Fast TreeSHAP Prep (Algorithm 3 in Appendix A.6) calculates the sets $\{U_{D_k,C} : C \subset D_k\}$ for all D_k's in the tree, and Fast TreeSHAP Score (Algorithm 4 in Appendix A.6) calculates ϕ_i's for all samples to be explained based on the pre-computed $\{U_{D_k,C} : C \subset D_k\}$. The main output of Fast TreeSHAP Prep is S, an $L \times 2^D$ matrix where each row records the values in $\{U_{D_k,C} : C \subset D_k\}$ for one path P_k. To calculate S, similar to the original TreeSHAP and Fast TreeSHAP v1, both EXTEND and UNWIND methods are used. The EXTEND method keeps track of $\{U_{m,D_k,C} : m = 0, \cdots, |C|\}$ for all possible subsets $C \subset D_k$ simultaneously, and the UNWIND method undoes a previous call to EXTEND when duplicated features are encountered in the path. At the leaf node, $U_{D_k,C}$ is obtained by summing up $\{U_{m,D_k,C}\}$ across m for all possible subsets $C \subset D_k$ simultaneously. In Fast TreeSHAP Score, given a feature vector x, we need to find out its corresponding S_k, i.e., the feature subset within path P_k where each feature satisfies the thresholds along the path, and then extract the corresponding value of $U_{D_k,D_k \setminus S_k}$ and $U_{D_k,D_k \setminus (\{i\} \cup S_k)}$ for $i \in D_k \setminus S_k$ from pre-computed S.

4.2.1 Complexity Analysis

In Fast TreeSHAP Prep, the time complexities of both EXTEND at the internal node and S calculation at the leaf node are bounded by $O(2^D D)$, where $O(2^D)$ comes from the number of possible subsets within each path, and $O(D)$ comes from the number of possible subset sizes. Thus the time complexity is $O(TL2^D D)$ for the entire ensemble of T trees. Note that this time complexity is independent with the number of samples to be explained, thus this entire part can be pre-computed, and matrix S can be stored together with other tree properties to facilitate future SHAP value calculation. The space complexity is dominated by S, which is $O(L2^D)$. Note that this complexity is for one tree. In practice, there are two ways to achieve this complexity for ensemble of T trees: i). Sequentially calculate S for each tree, and update SHAP values for all samples immediately after one S is calculated. ii). Pre-calculate S for all trees and store them in the local disk, and sequentially read each S into memory and update SHAP values for all samples accordingly.

In Fast TreeSHAP Score, it takes $O(1)$ time at each internal node to figure out S_k, and $O(D)$ time at each leaf node to loop over each of the D features in the path to extract its corresponding value from S (It takes $O(1)$ time to look up in S). Therefore, the loops at the leaves dominate the complexity of Fast TreeSHAP Score, which is $O(D)$. Finally, the complexity is $O(MTLD)$ for the entire ensemble of T trees and M samples to be explained. Compared with $O(MTLD^2)$ complexity in the original TreeSHAP and Fast TreeSHAP v1, this is a D-time improvement in computational complexity.

4.3 Fast TreeSHAP Summary

Table 1 summarizes the time and space complexities of each variant of TreeSHAP algorithm (M is the number of samples to be explained, $|N|$ is the number of features, T is the number of trees, L is the maximum number of leaves in any tree, and D is the maximum depth of any tree).

Table 1: Summary of computational complexities of TreeSHAP algorithms.

TreeSHAP Version	**Time Complexity**	**Space Complexity**
Original TreeSHAP	$O(MTLD^2)$	$O(D^2 + \|N\|)$
Fast TreeSHAP v1	$O(MTLD^2)$[1]	$O(D^2 + \|N\|)$
Fast TreeSHAP v2 (general case)	$O(TL2^D D + MTLD)$	$O(L2^D)$
Fast TreeSHAP v2 (balanced trees)	$O(TL^2 D + MTLD)$	$O(L^2)$

[1]Average running time is reduced to 25% of original TreeSHAP.

Fast TreeSHAP v1 strictly outperforms original TreeSHAP in average running time and performs comparably with original TreeSHAP in space allocation. Thus we recommend to at least replace original TreeSHAP with Fast TreeSHAP v1 in any tree-based model interpretation use cases.

We consider two scenarios in model interpretation use cases to compare Fast TreeSHAP v1 and v2:

- One-time usage: We explain all the samples for once, which usually occurs in ad-hoc model diagnosis. In this case, as mentioned in Section 4.2, Fast TreeSHAP v2 is preferred when $M > 2^{D+1}/D$ (commonly occurs in a moderate-sized dataset, as most tree-based models produce trees with depth ≤ 16). Also, Fast TreeSHAP v2 is under a stricter memory constraint: $O(L2^D) < memory\, tolerance$. For reference, for double type $L \times 2^D$ matrix S (assume in complete balanced trees, i.e., $L = 2^D$), its space allocation is 32KB for $D = 6$, 8MB for $D = 10$, and 2GB for $D = 14$. In practice, when D becomes larger, it becomes harder to build a complete balanced tree, i.e., L will be much smaller than 2^D, leading to a much smaller memory allocation than the theoretical upbound. We will see this in Section 5.
- Multi-time usage: We have a stable model in the backend, and we receive new data to be scored on a regular basis. This happens in most of the use cases of predictive modeling in industry, where the model is trained in a monthly/yearly frequency but the scoring data are generated in a daily/weekly frequency. One advantage of Fast TreeSHAP v2 is that it is well-suited for this multi-time usage scenario. In Fast TreeSHAP v2, we only need to calculate S once and store it in the local disk, and read it when new samples are coming, which leads to D-time computational speedup over Fast TreeSHAP v1.

5 Evaluation

We train different sizes of random forest models for evaluation on a list of datasets in Table 4 in Appendix A.7, with the goal of evaluating a wide range of tree ensembles representative of different real-world settings. While the first three datasets Adult [14], Superconductor [9], and Crop [12, 13] in Table 4 are publicly available, we also include one LinkedIn internal dataset "Upsell" to better illustrate the TreeSHAP implementation in industry. The Upsell dataset is used to predict how likely each LinkedIn customer is to purchase more Recruiters products by using features including product usage, recruiter activity, and company attributes. For each dataset, we fix the number of trees to be 100, and we train a small, medium, large, and extra-large model variant by setting the maximum depth of trees to be 4, 8, 12, and 16 respectively. Other hyperparameters in the random forest are left as default. Summary statistics for each model variant is listed in Table 5 in Appendix A.7.

We compare the execution times of Fast TreeSHAP v1 and v2 against the existing TreeSHAP implementation in the open source SHAP package (`https://github.com/slundberg/shap`). For fair comparison, we directly modify the C file `treeshap.h` in SHAP package to incorporate both Fast TreeSHAP v1 and v2. All the evaluations were run on a single core in Azure Virtual Machine with size Standard_D8_v3 (8 cores and 32GB memory). We ran each evaluation on 10,000 samples. In Table 2, results are averaged over 5 runs and standard deviations are also presented. To justify the correctness of Fast TreeSHAP v1 and v2, in each run we also compare the calculated SHAP values from Fast TreeSHAP v1 and v2 with SHAP values from the original TreeSHAP, and the maximal element-wise difference we observed during the entire evaluation process is $\sim 10^{-13}$, which is most likely the numerical error. We conduct pairwise comparisons between these three algorithms:

- Original TreeSHAP vs Fast TreeSHAP v1: For medium, large, and extra-large models, we observe speedups consistently around 1.5x. We observe lower speedup (around 1.2x) for small models probably due to the insufficient computation in computationally-expensive parts. These speedups also seem much lower than the theoretical upper bound (~ 4) discussed in Section 4.1, which is probably due to the existence of other tasks with slightly lower computational complexity in the algorithm.
- Original TreeSHAP vs Fast TreeSHAP v2: For medium and large models, we observe speedups around 2.5-3x, while the speedups drop to around 2x for extra-large models. This is because the first step Fast TreeSHAP Prep in Fast TreeSHAP v2 takes much longer time for larger models, and the execution time of Fast TreeSHAP v2 listed in Table 2 is a combination of its two steps. Later in this section, we will examine the execution times of Fast TreeSHAP Prep and Fast TreeSHAP Score separately.
- Fast TreeSHAP v1 vs Fast TreeSHAP v2: The speedups of Fast TreeSHAP v2 are consistently higher than the speedups of Fast TreeSHAP v1 except for small models, showing the effectiveness of Fast TreeSHAP v2 in improving the computational complexity. Their comparable performance for small models is also due to the insufficient computation.

Table 2: TreeSHAP vs Fast TreeSHAP v1 vs Fast TreeSHAP v2 - 10,000 samples (s.d. in parathesis).

Model	**Original TreeSHAP (s)**	**Fast TreeSHAP v1 (s)**	**Speedup**	**Fast TreeSHAP v2 (s)**	**Speedup**
Adult-Small	2.40 (0.03)	2.11 (0.02)	**1.14**	1.30 (0.04)	**1.85**
Adult-Med	61.04 (0.61)	44.09 (0.61)	**1.38**	26.62 (1.08)	**2.29**
Adult-Large	480.33 (3.60)	333.94 (4.20)	**1.44**	161.43 (3.95)	**2.98**
Adult-xLarge	1805.54 (13.75)	1225.20 (8.97)	**1.47**	827.62 (16.17)	**2.18**
Super-Small	2.50 (0.03)	2.04 (0.02)	**1.23**	1.28 (0.08)	**1.95**
Super-Med	89.93 (3.58)	60.04 (3.56)	**1.50**	35.65 (2.06)	**2.52**
Super-Large	1067.18 (10.52)	663.02 (5.79)	**1.61**	384.14 (4.78)	**2.78**
Super-xLarge	3776.44 (28.77)	2342.44 (35.23)	**1.61**	1988.48 (15.19)	**1.90**
Crop-Small	3.53 (0.07)	2.90 (0.04)	**1.22**	3.15 (0.02)	**1.12**
Crop-Med	69.88 (0.71)	50.13 (0.91)	**1.39**	34.57 (1.49)	**2.02**
Crop-Large	315.27 (6.37)	216.05 (8.64)	**1.46**	130.66 (3.80)	**2.41**
Crop-xLarge	552.23 (10.37)	385.51 (8.48)	**1.43**	290.49 (3.19)	**1.90**
Upsell-Small	2.80 (0.04)	2.23 (0.05)	**1.26**	2.20 (0.06)	**1.27**
Upsell-Med	90.64 (4.59)	63.34 (1.82)	**1.43**	34.02 (0.93)	**2.66**
Upsell-Large	790.83 (5.79)	515.16 (1.66)	**1.54**	282.98 (4.89)	**2.79**
Upsell-xLarge	2265.82 (17.44)	1476.56 (4.20)	**1.53**	1166.98 (15.02)	**1.94**

Table 3: Fast TreeSHAP v2: Fast TreeSHAP Prep & Fast TreeSHAP Score - 10,000 samples (s.d. in parathesis).

Model	**Original TreeSHAP (s)**	**Fast TreeSHAP Prep (s)**	**Fast TreeSHAP Score (s)**	**Speedup (Large M)**	**Space Allocation of S**
Adult-Small	2.40 (0.03)	<0.01 (<0.01)	1.30 (0.04)	**1.85**	2KB
Adult-Med	61.04 (0.61)	0.20 (0.01)	26.42 (1.07)	**2.31**	368KB
Adult-Large	480.33 (3.60)	11.32 (0.14)	150.11 (3.81)	**3.20**	24.9MB
Adult-xLarge	1805.54 (13.75)	268.90 (8.29)	558.72 (7.88)	**3.23**	955MB
Super-Small	2.50 (0.03)	<0.01 (<0.01)	1.28 (0.08)	**1.95**	2KB
Super-Med	89.93 (3.58)	0.36 (0.01)	35.29 (2.05)	**2.55**	462KB
Super-Large	1067.18 (10.52)	30.30 (0.34)	353.84 (4.34)	**3.02**	45.2MB
Super-xLarge	3776.44 (28.77)	673.04 (8.35)	1315.44 (6.84)	**2.87**	1.76GB
Crop-Small	3.53 (0.07)	<0.01 (<0.01)	3.15 (0.02)	**1.12**	2KB
Crop-Med	69.88 (0.71)	0.23 (0.01)	34.34 (1.48)	**2.03**	370KB
Crop-Large	315.27 (6.37)	8.08 (0.09)	122.58 (3.71)	**2.57**	15.1MB
Crop-xLarge	552.23 (10.37)	75.28 (2.34)	215.21 (2.02)	**2.57**	323MB
Upsell-Small	2.80 (0.04)	<0.01 (<0.01)	2.20 (0.06)	**1.27**	2KB
Upsell-Med	90.64 (4.59)	0.33 (0.01)	33.69 (0.92)	**2.69**	452KB
Upsell-Large	790.83 (5.79)	24.59 (0.36)	258.39 (4.53)	**3.06**	33.7MB
Upsell-xLarge	2265.82 (17.44)	442.74 (14.26)	724.24 (7.89)	**3.13**	996MB

Table 3 shows the execution times of Fast TreeSHAP Prep and Fast TreeSHAP Score in Fast TreeSHAP v2. We see that the execution time of Fast TreeSHAP Prep is almost negligible for small models, but increases dramatically when the model size increases. This coincides with our discussions in Section 4.3 that for Fast TreeSHAP v2, a larger model needs a larger set of samples to offset the computational cost in Fast TreeSHAP Prep. The column "Speedup" shows the ratios between the execution times of Fast TreeSHAP Score and original TreeSHAP. For one-time usage scenarios, this column approximates the speedup when sample size M is sufficiently large (i.e., Fast TreeSHAP Score dominates the execution time of Fast TreeSHAP v2). For multi-time usage scenarios, this column reflects the exact speedup when matrix S is pre-computed, and newly incoming samples are being explained. We observe that the speedup increases as the model size increases, which exactly reflects the D-time improvement in computational complexity between Fast TreeSHAP Score and

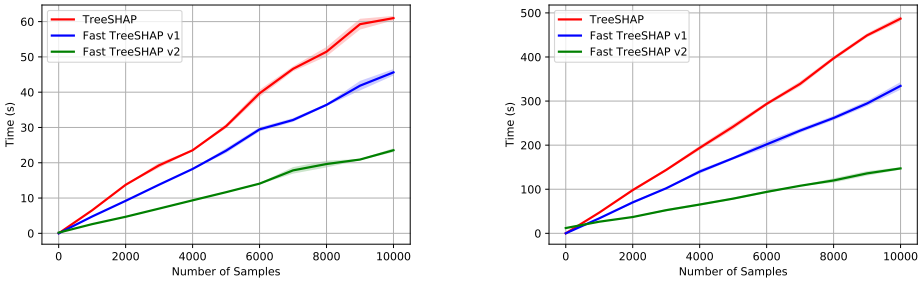

Figure 1: Execution time v.s. number of samples. Left: Adult-Med, Right: Adult-Large.

original TreeSHAP. Finally, the last column shows the space allocation of matrix S which dominates the memory usage of Fast TreeSHAP v2.[2] We can see that, although Fast TreeSHAP v2 costs more memory than the other two algorithms in theory, in practice, the memory constraint is quite loose as all the space allocations in Table 3 are not causing memory issues even in an ordinary laptop. Actually, the maximum depth of trees in most tree-based models in industry do not exceed 16.

Figure 1 plots the execution time versus number of samples for models Adult-Med and Adult-Large. 95% confidence interval of the execution time is also indicated by the shaded area. Here we consider the one-time usage scenarios for a better performance showcase of the two steps in Fast TreeSHAP v2. For Adult-Med, Fast TreeSHAP v2 almost starts at the same place as the other two algorithms, since the first step Fast TreeSHAP Prep takes only 0.2s. For Adult-Large, Fast TreeSHAP Prep takes much longer time due to the larger model size, resulting in higher starting point of the green curve. However, the green curve immediately crosses the other two curves when the number of samples exceeds 500, which coincides with our previous discussions on the sample size requirement of Fast TreeSHAP v2. In both plots, Fast TreeSHAP v2 consistently performs the best while the original TreeSHAP consistently performs the worst when the number of samples exceeds a certain threshold.

The above evaluations are all based on 100 trees, 10,000 samples, and 1 core for fast and fair comparisons. In real life scenarios, the number of trees can be as large as several thousands, hundreds of millions of samples can be encountered in model scoring, and multi-core machines can be used to conduct parallel computing. As parallel computing is one of our future works, we just briefly discuss the potential ways to parallelize Fast TreeSHAP v1 and v2 in Appendix A.8. Based on the proposed ways of parallel computing, we can reasonably expect that both Fast TreeSHAP v1 and v2 are able to significantly improve the computational efficiency in real life scenarios (e.g., reducing the execution time of explaining 20 million samples in a 100-core server from 15h to 5h, and reducing the execution time of explaining 320 thousand samples in a 4-core laptop from 3.5h to 1.4h). More details of the analysis of real life scenarios can be found in Appendix A.9.

6 Conclusion

TreeSHAP has been widely used for explaining tree-based models due to its desirable theoretical properties and polynomial computational complexity. In this paper, we presented Fast TreeSHAP v1 and Fast TreeSHAP v2, two new algorithms to further improve the computational efficiency of TreeSHAP, with the emphasis on explaining samples with a large size. Specifically, Fast TreeSHAP v1 shrinks the computational scope for the features along each path of the tree, which is able to consistently improve the computational speed while maintaining the low memory cost. Fast TreeSHAP v2 further splits part of computationally-expensive components into pre-computation step, which significantly reduces the time complexity from $O(MTLD^2)$ to $O(TL2^D D + MTLD)$ with a small sacrifice on memory cost, and is well-suited for multi-time model explanation scenarios.

As one of our future works, we are currently working on implementing the parallel computation in Fast TreeSHAP v1 and v2. Another future direction of Fast TreeSHAP is to implement it in Spark which naturally fits the environment of Hadoop clusters and the datasets stored in HDFS.

[2] Space allocation of S is calculated by $\#maximum leaves \cdot 2^{\#maximum\ depth} \cdot 8B$ for double type entries.

References

[1] Kjersti Aas, Martin Jullum, and Anders Løland. Explaining individual predictions when features are dependent: More accurate approximations to shapley values. *arXiv preprint arXiv:1903.10464*, 2019.

[2] Rudi Agius, Christian Brieghel, Michael A Andersen, Alexander T Pearson, Bruno Ledergerber, Alessandro Cozzi-Lepri, Yoram Louzoun, Christen L Andersen, Jacob Bergstedt, Jakob H von Stemann, et al. Machine learning can identify newly diagnosed patients with cll at high risk of infection. *Nature communications*, 11(1):1–17, 2020.

[3] Miller Janny Ariza-Garzón, Javier Arroyo, Antonio Caparrini, and Maria-Jesus Segovia-Vargas. Explainability of a machine learning granting scoring model in peer-to-peer lending. *Ieee Access*, 8:64873–64890, 2020.

[4] Kumar Ayush, Burak Uzkent, Marshall Burke, David Lobell, and Stefano Ermon. Generating interpretable poverty maps using object detection in satellite images. *arXiv preprint arXiv:2002.01612*, 2020.

[5] Hugh Chen, Joseph Janizek, Scott Lundberg, and Su-In Lee. True to the model or true to the data? *arXiv preprint arXiv:2006.16234*, 2020.

[6] Christopher Frye, Damien de Mijolla, Tom Begley, Laurence Cowton, Megan Stanley, and Ilya Feige. Shapley explainability on the data manifold. *arXiv preprint arXiv:2006.01272*, 2020.

[7] Brandon Greenwell. fastshap. `https://github.com/bgreenwell/fastshap`, 2019.

[8] Niloy Gupta and Isaac Joseph. Shparkley: Scaling shapley values with spark. `https://github.com/Affirm/shparkley`, 2020.

[9] Kam Hamidieh. A data-driven statistical model for predicting the critical temperature of a superconductor. *Computational Materials Science*, 154:346–354, 2018.

[10] Dominik Janzing, Lenon Minorics, and Patrick Blöbaum. Feature relevance quantification in explainable ai: A causal problem. In *International Conference on Artificial Intelligence and Statistics*, pages 2907–2916. PMLR, 2020.

[11] Harmanpreet Kaur, Harsha Nori, Samuel Jenkins, Rich Caruana, Hanna Wallach, and Jennifer Wortman Vaughan. Interpreting interpretability: Understanding data scientists' use of interpretability tools for machine learning. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2020.

[12] Iman Khosravi and Seyed Kazem Alavipanah. A random forest-based framework for crop mapping using temporal, spectral, textural and polarimetric observations. *International Journal of Remote Sensing*, 40(18):7221–7251, 2019.

[13] Iman Khosravi, Abdolreza Safari, and Saeid Homayouni. Msmd: maximum separability and minimum dependency feature selection for cropland classification from optical and radar data. *International Journal of Remote Sensing*, 39(8):2159–2176, 2018.

[14] Ron Kohavi et al. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *Kdd*, volume 96, pages 202–207, 1996.

[15] Konrad Komisarczyk, Pawel Kozminski, Szymon Maksymiuk, and Przemyslaw Biecek. treeshap. `https://github.com/ModelOriented/treeshap`, 2020.

[16] I Elizabeth Kumar, Suresh Venkatasubramanian, Carlos Scheidegger, and Sorelle Friedler. Problems with shapley-value-based explanations as feature importance measures. In *International Conference on Machine Learning*, pages 5491–5500. PMLR, 2020.

[17] Scott M Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. Explainable ai for trees: From local explanations to global understanding. *arXiv preprint arXiv:1905.04610*, 2019.

[18] Scott M Lundberg, Gabriel G Erion, and Su-In Lee. Consistent individualized feature attribution for tree ensembles. *arXiv preprint arXiv:1802.03888*, 2018.

[19] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Proceedings of the 31st international conference on neural information processing systems*, pages 4768–4777, 2017.

[20] Scott M Lundberg, Bala Nair, Monica S Vavilala, Mayumi Horibe, Michael J Eisses, Trevor Adams, David E Liston, Daniel King-Wai Low, Shu-Fang Newman, Jerry Kim, et al. Explainable machine-learning predictions for the prevention of hypoxaemia during surgery. *Nature biomedical engineering*, 2(10):749–760, 2018.

[21] Szymon Maksymiuk, Alicja Gosiewska, Przemyslaw Biecek, Mateusz Staniak, and Michal Burdukiewicz. shapper. `https://github.com/ModelOriented/shapper`, 2020.

[22] Luke Merrick and Ankur Taly. The explanation game: Explaining machine learning models using shapley values. In *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*, pages 17–38. Springer, 2020.

[23] Andreas Messalas, Christos Aridas, and Yannis Kanellopoulos. Evaluating mashap as a faster alternative to lime for model-agnostic machine learning interpretability. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 5777–5779. IEEE, 2020.

[24] Andreas Messalas, Yiannis Kanellopoulos, and Christos Makris. Model-agnostic interpretability with shapley values. In *2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA)*, pages 1–7. IEEE, 2019.

[25] Rory Mitchell, Eibe Frank, and Geoffrey Holmes. Gputreeshap: Fast parallel tree interpretability. *arXiv preprint arXiv:2010.13972*, 2020.

[26] W James Murdoch, Chandan Singh, Karl Kumbier, Reza Abbasi-Asl, and Bin Yu. Definitions, methods, and applications in interpretable machine learning. *Proceedings of the National Academy of Sciences*, 116(44):22071–22080, 2019.

[27] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " 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, 2016.

[28] Nikki Rommers, Roland Rössler, Evert Verhagen, Florian Vandecasteele, Steven Verstockt, Roel Vaeyens, Matthieu Lenoir, Eva D'Hondt, and Erik Witvrouw. A machine learning approach to assess injury risk in elite youth football players. *Medicine and science in sports and exercise*, 52(8):1745–1751, 2020.

[29] Mukund Sundararajan and Amir Najmi. The many shapley values for model explanation. In *International Conference on Machine Learning*, pages 9269–9278. PMLR, 2020.

[30] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*, pages 3319–3328. PMLR, 2017.

[31] Danding Wang, Qian Yang, Ashraf Abdul, and Brian Y Lim. Designing theory-driven user-centric explainable ai. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–15, 2019.

[32] Jilei Yang, Diana Negoescu, and Parvez Ahammad. Intellige: A user-facing model explainer for narrative explanations. *arXiv preprint arXiv:2105.12941*, 2021.

[33] Kang Zhang, Xiaohong Liu, Jun Shen, Zhihuan Li, Ye Sang, Xingwang Wu, Yunfei Zha, Wenhua Liang, Chengdi Wang, Ke Wang, et al. Clinically applicable ai system for accurate diagnosis, quantitative measurements, and prognosis of covid-19 pneumonia using computed tomography. *Cell*, 181(6):1423–1433, 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [N/A]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]
6. **Question from eXplainable AI approaches for debugging and diagnosis Workshop @ NeurIPS2021...**
   (a) **Would you like an extended version of the submission to be considered for publication in a journal special issue?** [Yes]

A Appendix

A.1 Link to Code Repository

Here is the link to an anonymized repository containing the code to replicate the results in Section 5: `https://drive.google.com/drive/folders/1Xfu7hGnhXGPwpMoFSpBNIoQhaPLeIYyh?usp=sharing`.

A.2 Algorithm Details of Estimating $f_S(x)$

Algorithm 1 proposed in [18, 17] provides the details to calculate $f_S(x)$ for tree-based models. Here a tree is specified as $\{v, a, b, t, r, d\}$, where v is a vector of node values, which takes the value $internal$ for internal nodes.[3] The vectors a and b represent the left and right node indexes for each internal node. The vector t contains the thresholds for each internal node, and d is a vector of indexes of the features used for splitting in internal nodes. The vector r represents the cover of each node (i.e., how many data samples fall in that node from its parent).

Algorithm 1 Estimating $f_S(x)$

```
procedure EXPVALUE(x, S, tree = {v, a, b, t, r, d})
    return G(1)                                          ▷ Start recursion at the root node
end procedure
procedure G(j)
    if v_j ≠ internal then                               ▷ Check if node j is a leaf
        return v_j                                       ▷ Return the leaf's value
    else
        if d_j ∈ S then                                  ▷ Check if the split feature is in S
            return G(a_j) if x_{d_j} ≤ t_j else G(b_j)   ▷ Use value of child on the decision path
        else
            return G(a_j) r_{a_j}/r_j + G(b_j) r_{b_j}/r_j   ▷ Weight values of both children by coverage
        end if
    end if
end procedure
```

A.3 Proof of Theorem 1

Proof. Plugging in $f_S(x) = \sum_{k=1}^{K} W_{k,S} v_k$ into Equation 1 leads to the SHAP value

$$\phi_i = \sum_{S \subset N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} \sum_{k=1}^{K} (W_{k,S \cup \{i\}} - W_{k,S}) v_k.$$

It is easy to see that $W_{k,S\cup\{i\}} = W_{k,S}$ if $i \notin D_k$ (i.e., $\forall j \in P_k$, $d_j \neq i$). Therefore, the above equation can be simplified as

$$\begin{aligned}\phi_i &= \sum_{S \subset N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} \left[\sum_{k: i \in D_k} (W_{k,S \cup \{i\}} - W_{k,S}) v_k \right] \\ &= \sum_{k: i \in D_k} \left[\sum_{S \subset N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} (W_{k,S \cup \{i\}} - W_{k,S}) \right] v_k.\end{aligned}$$

[3] A node in a tree can be either an internal node or a leaf node.

Similarly, $\forall l \notin D_k$ where $l \neq i$, $\forall S \subset N \backslash \{i, l\}$, we have $W_{k,S\cup\{i,l\}} = W_{k,S\cup\{i\}}$ and $W_{k,S\cup\{l\}} = W_{k,S}$, thus $W_{k,S\cup\{i,l\}} - W_{k,S\cup\{l\}} = W_{k,S\cup\{i\}} - W_{k,S}$. Therefore, $\forall l \notin D_k$,

$$
\begin{aligned}
\phi_i = & \sum_{k:i\in D_k} [\sum_{S\subset N\backslash\{i,l\}} \frac{|S|!(|N|-|S|-1)!}{|N|!}(W_{k,S\cup\{i\}} - W_{k,S}) \\
& + \sum_{S\subset N\backslash\{i,l\}} \frac{(|S|+1)!(|N|-|S|)!}{|N|!}(W_{k,S\cup\{i,l\}} - W_{k,S\cup\{l\}})]v_k \\
= & \sum_{k:i\in D_k} \left[\sum_{S\subset N\backslash\{i,l\}} \left(\frac{|S|!(|N|-|S|-1)!}{|N|!} + \frac{(|S|+1)!(|N|-|S|)!}{|N|!} \right) (W_{k,S\cup\{i\}} - W_{k,S}) \right] v_k \\
= & \sum_{k:i\in D_k} \left[\sum_{S\subset N\backslash\{i,l\}} \frac{|S|!(|N|-|S|-2)!}{(|N|-1)!}(W_{k,S\cup\{i\}} - W_{k,S}) \right] v_k.
\end{aligned}
$$

We repeat the above process $|N| - |D_k|$ times for each k, each time on a feature $l \notin D_k$, and we finally have

$$
\phi_i = \sum_{k:i\in D_k} \left[\sum_{S\subset D_k\backslash\{i\}} \frac{|S|!(|D_k|-|S|-1)!}{|D_k|!}(W_{k,S\cup\{i\}} - W_{k,S}) \right] v_k.
$$

Plugging in $W_{k,S} = \prod_{j\in P_k, d_j\in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} \cdot \prod_{j\in P_k, d_j\notin S} R_{kj}$ into the above equation, we have

$$
\begin{aligned}
\phi_i = & \sum_{k:i\in D_k} [\sum_{S\subset D_k\backslash\{i\}} \frac{|S|!(|D_k|-|S|-1)!}{|D_k|!} (\prod_{j\in P_k, d_j\in S\cup\{i\}} \mathbb{1}\{x_{d_j} \in T_{kj}\} \prod_{j\in P_k, d_j\in D_k\backslash(S\cup\{i\})} R_{kj} \\
& - \prod_{j\in P_k, d_j\in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} \prod_{j\in P_k, d_j\in D_k\backslash S} R_{kj})]v_k \\
= & \sum_{k:i\in D_k} [\sum_{S\subset D_k\backslash\{i\}} \frac{|S|!(|D_k|-|S|-1)!}{|D_k|!} (\prod_{j\in P_k, d_j\in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} \prod_{j\in P_k, d_j\in D_k\backslash(S\cup\{i\})} R_{kj}) \\
& (\prod_{j\in P_k, d_j=i} \mathbb{1}\{x_{d_j} \in T_{kj}\} - \prod_{j\in P_k, d_j=i} R_{kj})]v_k \\
= & \sum_{k:i\in D_k} \left[\sum_{m=0}^{|D_k|-1} \frac{m!(|D_k|-m-1)!}{|D_k|!} \sum_{S\subset D_k\backslash\{i\}, |S|=m} \left(\prod_{j\in P_k, d_j\in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} \prod_{j\in P_k, d_j\in D_k\backslash(S\cup\{i\})} R_{kj} \right) \right] \\
& \cdot \left(\prod_{j\in P_k, d_j=i} \mathbb{1}\{x_{d_j} \in T_{kj}\} - \prod_{j\in P_k, d_j=i} R_{kj} \right) v_k.
\end{aligned}
$$

□

A.4 Proof of Theorem 2

Proof. Let $S_{ki} := \{d_j : j \in P_k, x_{d_j} \notin T_{kj}, d_j \neq i\}$, i.e., S_{ki} is the subset of $D_k\backslash\{i\}$ where the features in S_{ki} do not satisfy the thresholds along the path P_k. Note that $\prod_{j\in P_k, d_j\in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} = 0$ if $S \cap S_{ki} \neq \emptyset$ and $\prod_{j\in P_k, d_j\in S} \mathbb{1}\{x_{d_j} \in T_{kj}\} = 1$ otherwise, Equation 2 can then be simplified as

$$
\begin{aligned}
\phi_i = & \sum_{k:i\in D_k} \left[\sum_{m=0}^{|D_k|-1-|S_{ki}|} \frac{m!(|D_k|-m-1)!}{|D_k|!} \left(\sum_{S\subset D_k\backslash(\{i\}\cup S_{ki}), |S|=m} \prod_{j\in P_k, d_j\in D_k\backslash(S\cup\{i\}\cup S_{ki})} R_{kj} \right) \right] \\
& \cdot \prod_{j\in P_k, d_j\in S_{ki}} R_{kj} \cdot \left(\prod_{j\in P_k, d_j=i} \mathbb{1}\{x_{d_j} \in T_{kj}\} - \prod_{j\in P_k, d_j=i} R_{kj} \right) v_k.
\end{aligned}
$$

I.e., only subsets $S \subset D_k \backslash (\{i\} \cup S_{ki})$ are necessary to be included in the equation.

Remind that $S_k = \{d_j : j \in P_k, x_{d_j} \notin T_{kj}\}$, i.e., S_k is the subset of D_k where the features in S_k do not satisfy the thresholds along the path P_k. Also remind that $U_{m,D_k,C} := \frac{m!(|D_k|-m)!}{(|D_k|+1)!} \sum_{S \subset C, |S|=m} \prod_{j \in P_k, d_j \in C \backslash S} R_{kj}$, $m = 0, \cdots, |C|$, and $U_{D_k,C} := \sum_{m=0}^{|C|} \frac{|D_k|+1}{|D_k|-m} U_{m,D_k,C}$, where $C \subset D_k$ is a subset of D_k.

Then it is easy to see that when $\prod_{j \in P_k, d_j = i} \mathbb{1}\{x_{d_j} \in T_{kj}\} = 0$ (i.e., $i \in S_k$), we have $S_{ki} \cup \{i\} = S_k$, and

$$
\begin{aligned}
\phi_i &= \sum_{k:i \in D_k} \left[\sum_{m=0}^{|D_k|-|S_k|} \frac{m!(|D_k|-m-1)!}{|D_k|!} \left(\sum_{S \subset D_k \backslash S_k, |S|=m} \prod_{j \in P_k, d_j \in D_k \backslash (S \cup S_k)} R_{kj} \right) \right] \\
&\quad \cdot \prod_{j \in P_k, d_j \in S_{ki}} R_{kj} \left(- \prod_{j \in P_k, d_j = i} R_{kj} \right) v_k \\
&= - \sum_{k:i \in D_k} \left[\sum_{m=0}^{|D_k|-|S_k|} \frac{|D_k|+1}{|D_k|-m} U_{m,D_k,D_k \backslash S_k} \right] \prod_{j \in P_k, d_j \in S_k} R_{kj} v_k \\
&= - \sum_{k:i \in D_k} \left[U_{D_k,D_k \backslash S_k} \prod_{j \in P_k, d_j \in S_k} R_{kj} \right] v_k.
\end{aligned}
$$

When $\prod_{j \in P_k, d_j = i} \mathbb{1}\{x_{d_j} \in T_{kj}\} = 1$ (i.e., $i \notin S_k$), we have $S_{ki} = S_k$, and

$$
\begin{aligned}
\phi_i &= \sum_{k:i \in D_k} \left[\sum_{m=0}^{|D_k|-1-|S_k|} \frac{m!(|D_k|-m-1)!}{|D_k|!} \left(\sum_{S \subset D_k \backslash (\{i\} \cup S_k), |S|=m} \prod_{j \in P_k, d_j \in D_k \backslash (S \cup \{i\} \cup S_k)} R_{kj} \right) \right] \\
&\quad \cdot \prod_{j \in P_k, d_j \in S_k} R_{kj} \left(1 - \prod_{j \in P_k, d_j = i} R_{kj} \right) v_k \\
&= \sum_{k:i \in D_k} \left[\sum_{m=0}^{|D_k|-1-|S_k|} \frac{|D_k|+1}{|D_k|-m} U_{m,D_k,D_k \backslash (\{i\} \cup S_k)} \right] \prod_{j \in P_k, d_j \in S_k} R_{kj} \left(1 - \prod_{j \in P_k, d_j = i} R_{kj} \right) v_k \\
&= \sum_{k:i \in D_k} \left[U_{D_k,D_k \backslash (\{i\} \cup S_k)} \prod_{j \in P_k, d_j \in S_k} R_{kj} (1 - \prod_{j \in P_k, d_j = i} R_{kj}) \right] v_k.
\end{aligned}
$$

□

A.5 Fast TreeSHAP v1 Algorithm Details

In Algorithm 2, m is the path of unique features we have split on so far, and contains three attributes (We use the dot notation to access member values)[4]: i) $m_{j+1}.d$, the feature index for the jth internal node (i.e., d_j), ii) $m_{j+1}.z$, the covering ratio for the jth internal node (i.e., R_{kj}), iii) $m_{j+1}.o$, the threshold condition for the jth internal node (i.e., $\mathbb{1}\{x_{d_j} \in T_{kj}\}$) ($m_0.d$, $m_0.z$, $m_0.o$ record the information of a dummy node). Moreover, at the jth internal node, w is used to record $U_{m,D_{kj},D_{kj} \backslash S_{kj}}$ ($m = 0, \cdots, j - |S_{kj}|$) where D_{kj} and S_{kj} are subsets of D_k and S_k up to the jth internal node, and q is used to record $\prod_{l \in P_{kj}, d_l \in S_{kj}} R_{kl}$ where P_{kj} is the subpath of P_k up to the jth internal node. When reaching the leaf node, m will reach length $|D_k|+1$, w will reach length $|D_k|-|S_k|+1$ and q is always a scalar. The values p_z, p_o, and p_i represent the covering ratio, the threshold condition, and the feature index of the last split. One thing to note is that the functionality of EXTEND in Algorithm 2 is slightly different from what we described in Section 4.1, where in addition to its functionality described in Section 4.1 when we encounter a feature not in S_k, EXTEND is also called when we encounter a feature in S_k to simply update the Shapley weights.

[4]We use the same notation as in the original TreeSHAP algorithm for easy reference.

Algorithm 2 Fast TreeSHAP v1

```
procedure TREESHAP(x, tree = {v, a, b, t, r, d})
    φ = array of len(x) zeros
    RECURSE(0, [], [], 1, 1, 1, −1)                    ▷ Start recursion at the root node
    return φ
end procedure
procedure RECURSE(j, m, w, q, p_z, p_o, p_i)
    m, w, q = EXTEND(m, w, q, p_z, p_o, p_i)           ▷ Update m, w, q for the feature of the last split
    if v_j ≠ internal then                             ▷ Check if we are at a leaf node
        if len(m) > len(w) then
            s_0 = −sum(UNWIND(m, w, q, −1)[1])         ▷ Pre-calculate U_{D_k,D_k\S_k} for d_i's in S_k
        end if
        for i ← 1 to len(m) − 1 do
            if m_i.o = 0 then
                φ_{m_i.d} = φ_{m_i.d} + s_0 · q · v_j  ▷ Update contribution for feature d_i ∈ S_k
            else
                s = sum(UNWIND(m, w, q, i)[1])         ▷ Calculate U_{D_k,D_k\({d_i}∪S_k)} for d_i ∉ S_k
                φ_{m_i.d} = φ_{m_i.d} + s · q · (1 − m_i.z) · v_j ▷ Update contribution for feature d_i ∉ S_k
            end if
        end for
    else
        h, c = (a_j, b_j) if x_{d_j} ≤ t_j else (b_j, a_j)   ▷ Determine hot and cold children
        i_z = i_o = 1
        k = FINDFIRST(m.d, d_j)
        if k ≠ nothing then   ▷ Undo previous update for feature d_k if we see this feature again
            i_z, i_o = m_k.z, m_k.o
            m, w, q = UNWIND(m, w, q, k)
        end if
        RECURSE(h, m, w, q, i_z r_h/r_j, i_o, d_j)
        RECURSE(c, m, w, q, i_z r_c/r_j, 0, d_j)
    end if
end procedure
procedure EXTEND(m, w, q, p_z, p_o, p_i)               ▷ Update m, w, q for the feature of the last split
    l, l_w = len(m), len(w)
    m, w = copy(m), copy(w)
    m_l.(d, z, o) = (p_i, p_z, p_o)                    ▷ Extend m given the feature of the last split
    if p_o = 0 then
        q = q · p_z
        for i ← l_w − 1 to 0 do
            w_i = w_i · (l − i)/(l + 1)
        end for
    else
        w_{l_w} = 1 if l_w = 0 else 0                  ▷ Extend w if the feature of the last split is not in S_k
        for i ← l_w − 1 to 0 do
            w_{i+1} = w_{i+1} + w_i · (i + 1)/(l + 1)
            w_i = p_z · w_i · (l − i)/(l + 1)
        end for
    end if
    return m, w, q
end procedure
procedure UNWIND(m, w, q, i)                           ▷ Undo previous update for feature d_i
    l, l_w = len(m) − 1, len(w) − 1
    m = copy(m_{0...l−1})
    if i < 0 then
        w = copy(w)
        for j ← l_w to 0 do
            w_j = w_j · (l + 1)/(l − j)
```

```
        end for
    else
        if m_i.o = 0 then
            w = copy(w)
            for j ← l_w to 0 do
                w_j = w_j · (l + 1)/(l − j)
            end for
            q = q/m_i.z
        else
            n = w_{l_w}
            w = copy(w_{0···l_w−1})                    ▷ Shrink w if feature d_i ∉ S_k
            for j ← l_w − 1 to 0 do
                t = w_j
                w_j = n · (l + 1)/(j + 1)
                n = t − w_j · m_i.z · (l − j)/(l + 1)
            end for
        end if
        for j ← i to l − 1 do
            m_j.(d, z, o) = m_{j+1}.(d, z, o)          ▷ Shrink m given feature d_i
        end for
    end if
    return m, w, q
end procedure
```

A.6 Fast TreeSHAP v2 Algorithm Details

In Algorithm 3 and 4, S is a matrix of size $L \times 2^D$ where each row records the values in the set $\{U_{D_k,C} : C \subset D_k\}$ for one path P_k, and the subsets C in each row are ordered according to the reverse ordering of binary digits.[5] W is a matrix of size $2^D \times (D+1)$, where each row corresponds to a subset $C \subset D_k$ ordered according to the reverse ordering of binary digits. At the jth internal node along path P_k, each row of W records $\{U_{m,D_{kj},C} : m = 0, \cdots, |C|\}$ for a subset $C \subset D_{kj}$ where D_{kj} is the subset of D_k up to the jth internal node. It is easy to see that at the leaf node of path P_k, summing up each row of W exactly matches one row in S corresponding to path P_k. l_w is a vector of size 2^D which records the number of nonzero elements in each row of W. e is a vector of the same size as d, which records the node indices of duplicated features along the path. c is an integer which records the number of paths explored so far.

Algorithm 3 Fast TreeSHAP v2: Prep

```
procedure TREESHAPPREP(tree = {v, a, b, t, r, d})
    S = matrix of L × 2^D zeros
    e = array of len(d) negative ones
    W = matrix of 2^D × (D + 1) zeros
    l_w = array of 2^D zeros
    c = 0
    RECURSE(0, [], W, l_w, 1, −1)                      ▷ Start recursion at the root node
    return S, e
end procedure
procedure RECURSE(j, m, W, l_w, p_z, p_i)
    m, W, l_w = EXTEND(m, W, l_w, p_z, p_i)    ▷ Update m, W, l_w for the feature of the last split
    if v_j ≠ internal then                                  ▷ Check if we are at a leaf node
        l = len(m) − 1
        for t ← 0 to 2^l − 1 do          ▷ Calculate {U_{D_k,C} : C ⊂ D_k} for the current path
            for i ← l_w[t] to 0 do
                S[c, t] = S[c, t] + W[t, i] · (l + 1)/(l − i)
            end for
        end for
```

[5] E.g., in a 3-element set, using 0/1 to indicate absence/presence of an element, the subsets in reverse ordering of binary digits are $\{(0,0,0),\ (1,0,0),\ (0,1,0),\ (1,1,0),\ (0,0,1),\ (1,0,1),\ (0,1,1),\ (1,1,1)\}$.

```
            c = c + 1
        else
            i_z = 1
            k = FINDFIRST(m.d, d_j)
            if k ≠ nothing then
                e_j = k                                  ▷ Record node index of duplicated feature
                i_z = m_k.z
                m, W, l_w = UNWIND(m, W, l_w, k)  ▷ Undo previous update for d_k if we see it again
            end if
            l_w = l_w + 1
            RECURSE(a_j, m, W, l_w, i_z r_{a_j}/r_j, d_j)
            RECURSE(b_j, m, W, l_w, i_z r_{b_j}/r_j, d_j)
        end if
    end procedure
    procedure EXTEND(m, W, l_w, p_z, p_i)        ▷ Update m, W, l_w for the feature of the last split
        l = len(m)
        m = copy(m)
        m_l.(d, z) = (p_i, p_z)                      ▷ Extend m given the feature of the last split
        if l = 0 then
            W[0, 0] = 1
        else                                         ▷ Extend W, l_w given the feature of the last split
            W, l_w = copy(W), copy(l_w)
            W[2^{l-1} : 2^l, :] = copy(W[: 2^{l-1}, :])
            l_w[2^{l-1} : 2^l] = copy(l_w[: 2^{l-1}])
            for t ← 0 to 2^{l-1} − 1 do
                for i ← l_w[t] − 1 to 0 do
                    W[t, i] = W[t, i] · (l − i)/(l + 1)
                end for
            end for
            for t ← 2^{l-1} to 2^l − 1 do
                for i ← l_w[t] − 1 to 0 do
                    W[t, i + 1] = W[t, i + 1] + W[t, i] · (i + 1)/(l + 1)
                    W[t, i] = p_z · W[t, i] · (l − i)/(l + 1)
                end for
            end for
            l_w[: 2^{l-1}] = l_w[: 2^{l-1}] − 1
        end if
        return m, W, l_w
    end procedure
    procedure UNWIND(m, W, l_w, i)                   ▷ Undo previous update for feature d_i
        l = len(m) − 1
        m, W, l_w = copy(m_{0...l-1}), copy(W), copy(l_w)
        ind = empty array
        for t_1 ← {0, 2^i, 2 · 2^i, 3 · 2^i, · · · , 2^l − 2^i} do
            for t_2 ← t_1 to t_1 + 2^{i-1} − 1 do
                ind.append(t_2)
            end for
        end for
        W[: 2^{l-1}, :] = W[ind, :]                  ▷ Shrink W, l_w given feature d_i
        l_w[: 2^{l-1}] = l_w[ind]
        for t ← 0 to 2^{l-1} − 1 do
            for i ← l_w[t] to 0 do
                W[t, i] = W[t, i] · (l + 1)/(l − i)
            end for
        end for
        for j ← i to l − 1 do
            m_j.(d, z) = m_{j+1}.(d, z)              ▷ Shrink m given feature d_i
        end for
```

```
    return m, W, l_w
end procedure
```

Algorithm 4 Fast TreeSHAP v2: Score

```
procedure TREESHAPSCORE(x, tree = {v, a, b, t, r, d, S, e})
    φ = array of len(x) zeros
    b = array of [0, 1, 2, 2^2, ··· , 2^D]
    c = 0
    RECURSE(0, [], 1, 1, 1, −1)                    ▷ Start recursion at the root node
    return φ
end procedure
procedure RECURSE(j, m, q, p_z, p_o, p_i)
    l = len(m)
    m = copy(m)
    m_l.(d, z, o) = (p_i, p_z, p_o)                ▷ Update m, q for the feature of the last split
    if p_o = 0 then
        q = q · p_z
    end if
    if v_j ≠ internal then                         ▷ Check if we are at a leaf node
        b_sum = 0
        for i ← 1 to l do          ▷ Use b_sum to search in {U_{D_k,C} : C ⊂ D_k} for the current path
            b_sum = b_sum + b_i if m_i.o ≠ 0
        end for
        for i ← 1 to l do
            if m_i.o = 0 then                      ▷ Update contribution for feature d_i ∈ S_k
                φ_{m_i.d} = φ_{m_i.d} − S[c, b_sum] · q · v_j
            else                                   ▷ Update contribution for feature d_i ∉ S_k
                φ_{m_i.d} = φ_{m_i.d} + S[c, b_sum − b_i] · q · (1 − m_i.z) · v_j
            end if
        end for
        c = c + 1
    else
        i_z = i_o = 1
        k = e_j
        if k ≠ −1 then         ▷ Undo previous update for feature d_k if we see this feature again
            i_z, i_o = m_k.z, m_k.o
            m = copy(m_{0···l−1})
            for j ← k to l − 1 do
                m_j.(d, z, o) = m_{j+1}.(d, z, o)
            end for
            if i_o = 0 then
                q = q/i_z
            end if
        end if
        RECURSE(a_j, m, q, i_z r_{a_j}/r_j, i_o · 𝟙(x_{d_j} ≤ t_j), d_j)
        RECURSE(b_j, m, q, i_z r_{b_j}/r_j, i_o · 𝟙(x_{d_j} > t_j), d_j)
    end if
end procedure
```

A.7 Additional Tables in Evaluation

Table 4 lists the basic information of each dataset used in the evaluation study.

Table 5 lists the summary statistics for each model variant used in the evaluation study. We fix the number of trees to be 100, since the computational time is approximately linear in terms of the number of trees, therefore without loss of generalizability, we simply set the number of trees to be a fixed number.

Table 4: Datasets.

Name	**# Instances**	**# Attributes (Original)**	**# Attributes (One-Hot)**	**Task**	**Classes**
Adult[14]	48,842	14	64	Classification	2
Superconductor[9]	21,263	81	81	Regression	-
Crop[12, 13]	325,834	174	174	Classification	7
Upsell	96,120	169	182	Classification	2

Table 5: Models.

Model	**# Trees**	**Max Depth**	**# Leaves**
Adult-Small	100	4	1,549
Adult-Med	100	8	14,004
Adult-Large	100	12	57,913
Adult-xLarge	100	16	140,653
Super-Small	100	4	1,558
Super-Med	100	8	19,691
Super-Large	100	12	124,311
Super-xLarge	100	16	314,763
Crop-Small	100	4	1,587
Crop-Med	100	8	13,977
Crop-Large	100	12	37,503
Crop-xLarge	100	16	51,195
Upsell-Small	100	4	1,600
Upsell-Med	100	8	20,247
Upsell-Large	100	12	92,101
Upsell-xLarge	100	16	181,670

A.8 Fast TreeSHAP Parallelization

Parallel computing is not the main topic of this paper and will be one of our future works. Here we just briefly discuss the potential ways to parallelize Fast TreeSHAP v1 and v2 to achieve an even higher computational speed. Fast TreeSHAP v1 can be parallelized per sample, similar to the parallelization setting in the original TreeSHAP. The parallelization setting for Fast TreeSHAP v2 is a bit complicated: For one-time usage, per-tree parallelization is recommended as long as the total memory cost across multiple threads (i.e., $\#thread \cdot O(L2^D)$) fits the entire memory, which works for most moderate-sized trees. For multi-time usage or when per-tree parallelization is not applicable, we can pre-compute S's for all trees and store them in the local disk, and read them when explaining the samples. In this case, per-sample parallelization can be implemented. Specifically, we can read these S's in batches with size B which leads to $O(BL2^D)$ memory cost. When the corresponding tree size is small, we can increase the batch size to potentially reduce the overheads in data loading. When the tree size is large, we can decrease the batch size until it fits the memory well. Once a batch of S's have been loaded into memory, we can implement per-sample parallelization to update SHAP values for all samples for this batch of trees.

A.9 Analysis of Real Life Scenarios

The evaluation studies in Section 5 are all based on 100 trees, 10,000 samples, and 1 core for fast and fair comparisons. In real life scenarios in industry, the number of trees is usually between 200 and 1000, hundreds of millions of samples can be encountered in model scoring, and multi-core machines can be used to conduct parallel computing. Based on the discussions in Section A.8 and the linear patterns in Figure 1, we can reasonably expect the linear scaling of the performance of the three TreeSHAP algorithms with respect to the number of trees, the number of samples, and the number of threads in parallel computing. Here we list two real life scenarios and estimate the improvement of Fast TreeSHAP v1 and v2 over the original TreeSHAP (The estimates may not be very accurate, and

they are mostly used to provide an intuitive view of the magnitude of problems and the improvement of proposed approaches):

- In the prediction of Premium subscriptions of LinkedIn members, 20 million LinkedIn members need to be scored every week. The model we have built is a random forest model which is refreshed every half a year. This model contains 400 trees with the maximum depth of 12. Assume we are using a server with 100 cores. We can reasonably approximate from Table 2 that the execution time of explaining the entire 20 million samples by using the original TreeSHAP is about $660 \times 4 \times 2000/100$s = 14.7h (we use the average execution time of the large models across 4 datasets) even with parallel computing on 100 cores. This computational cost is much higher than the model scoring itself! We propose using Fast TreeSHAP v2 as this is the multi-time usage scenario, where matrix S can be pre-computed and stored in the server. From Table 3, we can reasonably estimate that the matrix S occupies about 400×30MB=12GB in storage (store all 400 trees together), and 30MB in memory (read each tree sequentially) when running Fast TreeSHAP v2, which should well fit the industry-level server. We can also estimate from Table 3 that Fast TreeSHAP v2 provides $\sim$ 3x speedup compared with the original TreeSHAP, which reduces the execution time of model interpretation from 14.7h to just 4.9h.
- In the crop mapping prediction [12, 13], 320 thousand remote sensing images need to be scored on a 4-core laptop. The model we have built is similar to the model used in Section 5: a random forest model with 500 trees and maximum depth of 12. From Table 2, it is reasonable to estimate that the execution time of the original TreeSHAP is about $315 \times 32 \times 5/4$s = 3.5h when parallelizing on the entire 4 cores. Fast TreeSHAP v1 can help reduce the execution time to $216 \times 32 \times 5/4$s = 2.4h. Since this is the one-time usage scenario, per-tree parallelization can be implemented for Fast TreeSHAP v2, which costs around 15×4MB = 60MB in memory, and can further reduce the execution time to $130 \times 32 \times 5/4$s = 1.4h.