Fast TreeSHAP: Accelerating SHAP Value Computation for Trees

Jilei Yang
LinkedIn Corporation
jlyang@linkedin.com

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. The link to the code repository to replicate the results in this paper is https://drive.google.com/drive/u/4/folders/1P0V0Vj42KO4QcTSwmbMW4Y6joycB7ZQ.

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 [28]. There exist several state-of-the-art sample-level model interpretation approaches, e.g., SHAP [20], LIME [28], and Integrated Gradient [31]. 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 [21, 34, 2, 3].

There exist several variants of SHAP. The general version is KernelSHAP [20], which is model-agnostic, and generally takes exponential time to compute the exact SHAP values. Its variants include TreeSHAP [19, 18] and DeepSHAP [20], 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.

1st Workshop on eXplainable AI approaches for debugging and diagnosis (XAI4Debugging@NeurIPS2021).
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 large model size (e.g., maximum tree depth $\geq 10$) 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.11).

In fact, explaining (at least) tens of millions of samples widely exist in member-level predictive models in industry, 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: On one hand, it is likely to cause huge delays in post-hoc model diagnosis via important feature analysis, increasing the risks of incorrect model implementations and untimely model iterations. On the other hand, it can also lead to long waiting time in preparing actionable items for model end users (e.g., marketing team in subscription propensity model) based on feature reasoning \[33\], and as a result, the end users may not take appropriate actions in a timely manner, which can negatively impact the company’s revenue.

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 Work

Since the introductions of SHAP \[20\] and TreeSHAP \[19, 18\], 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 \[21, 34, 2\], social science \[4\], finance \[3\] and sports \[29\]. There also exist a lot of exciting papers focusing on the designs of SHAP/TreeSHAP implementations \[32, 27, 12, 33\], as well as the theoretical justification of SHAP values \[11, 17, 30, 11, 23, 5\].

Besides these works, only a few of them focus on the computational efficiency of SHAP/TreeSHAP: The authors of \[8\] developed a Python package fastshap to approximate SHAP values for arbitrary models by calling scoring function as few times as possible. In \[9\], 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 \[22, 16\] built R packages shapper and treeexpl as R wrappers of SHAP Python library, which achieved comparable speed. However, no algorithmic improvements have been made. In \[24, 25\], 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 \[26\], 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 \[20\], and then walk through the derivation of TreeSHAP algorithm \[19, 18\] 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: \( g(z') = \phi_0 + \sum_{i=1}^{|N|} \phi_i z'_i \). 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 [20], SHAP values are the single unique solution of \( \phi_i \)'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 \( f_S(x) = E[f(x)|x_S] = E_{x_S|x_S}[f(x)] \), and the marginal expectation \( f_S(x) = E_{x_S}[f(x)] \), 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 [11][17][20][22][5], both options need exponential time in computation as Equation \( \text{1} \) 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 [19][18]. The authors of [19][18] 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 [19][18] (Appendix A.3) 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 \( O(M|TLD^2|N) \) 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 [19][18] 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 [19][18] 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 \( v_k \) to \( v_{k+1} \), 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 \( j \)th 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 \( T_{kj} = (-\infty, t_{kj}] \).
We now further simplify Equation 2 for an even faster computational speed. Consider a subset $S$ for subset size 1 where

$$\text{Theorem 2.}$$

The proof of Theorem 1 can be found in Appendix A.3. For convenience, we call simplifies Equation 2 (proof in Appendix A.4):

can be interpreted as the sum of the "proportion"s of all subsets that flow down into a leaf node $v_k$, then

$$f_S(x) = \sum_{k=1}^{K} \prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{k_j}\} \cdot \prod_{j \in P_k, d_j \not\in S} R_{k_j} \cdot v_k,$$

where $\mathbb{1}\{\cdot\}$ is an indicator function. Let $W_{k,S} := \prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{k_j}\} \cdot \prod_{j \in P_k, d_j \not\in S} R_{k_j}$, i.e., $W_{k,S}$ is the "proportion" of subset $S$ that flows down into the leaf node $v_k$, then

$$f_S(x) = \sum_{k=1}^{K} \frac{W_{k,S} \cdot v_k}{|N|}.$$

Plugging it into Equation 1 leads to the SHAP value

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \sum_{k=1}^{K} \sum_{|S| = m} \frac{W_{k,S}}{|N|} \sum_{S \subseteq D_k \setminus \{i\}} \prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{k_j}\} \prod_{j \in P_k, d_j \not\in S} R_{k_j} \left(\prod_{j \in P_k, d_j = i} \mathbb{1}\{x_{d_j} \in T_{k_j}\} - \prod_{j \in P_k, d_j = i} R_{k_j}\right) v_k,$$

(2)

The proof of Theorem 1 can be found in Appendix A.3. For convenience, we call 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 algorithm 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, · · · up to the current depth. At the leaf node, EXTEND reaches the sequence of values

$$\frac{m(|D_k|-m)!}{(m+1)!} \sum_{S \subseteq D_k, |S|=m} \left(\prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{k_j}\} \prod_{j \in P_k, d_j \not\in S} R_{k_j}\right)$$

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., $S_k = \{d_j : j \in P_k, x_{d_j} \not\in T_{k_j}\}$. Also define $U_{m,D_k,C} := \frac{m(|D_k|-m)!}{(m+1)!} \sum_{S \subseteq C, |S|=m} \prod_{j \in P_k, d_j \in S} R_{k_j}$ for $m = 0, \cdots, |C|$, where $C \subset D_k$ is a subset of $D_k$. We see that when $C = D_k \setminus S_k$, $U_{m,D_k,C,k}$ 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 $U_{D_k,C} := \sum_{m=0}^{|C|} \frac{m(|D_k|-m)!}{(m+1)!} U_{m,D_k,C}$. Theorem 2 further simplifies Equation 2 (proof in Appendix A.4).

$$\text{Theorem 2.}$$

We see that when $C = D_k \setminus S_k$, $U_{m,D_k,C,k}$ 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 $U_{D_k,C} := \sum_{m=0}^{|C|} \frac{m(|D_k|-m)!}{(m+1)!} U_{m,D_k,C}$. Theorem 2 further simplifies Equation 2 (proof in Appendix A.4).

$$\phi_i = \begin{cases} -\frac{U_{D_k,D_k \setminus S_k}}{U_{D_k,D_k \setminus S_k}} \sum_{j \in P_k, d_j \in S_k} R_{k_j} v_k & \text{if } i \in S_k, \\ \sum_{k: i \in D_k} \left[U_{D_k,D_k \setminus \{i\} \cup S_k} \prod_{j \in P_k, d_j \in S_k} R_{k_j} (1 - \prod_{j \in P_k, d_j = i} R_{k_j}) \right] v_k & \text{if } i \not\in S_k. \end{cases}$$

(3)
4.1 Fast TreeSHAP v1 Algorithm

We propose Fast TreeSHAP v1 algorithm based on Theorem 2 which runs in $O(M TLD^2)$ time and $O(D^2 + |N|)$ memory. This computational complexity looks the same as in the original TreeSHAP. However, we will show in sections 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, ⋯ 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 $U_{m,d_k,d_k \setminus (i) \setminus S_k}$ 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 5, 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 (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., $E[|D_k| - |S_k|]/|D_k| \approx \sum_{i=0}^{D} C_i^D (D - i)/(2^D D) \approx 50\%$. 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% × 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 $O(TL^2D + MTLD)$ time and $O(L^2D)$ memory. For balanced trees it becomes $O(TL^2D + MTLD)$ time and $O(L^2D)$ 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 \(\{U_{D_k,C} : C \subseteq D_k\}\) 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 \(\{U_{D_k,C} : C \subseteq D_k\}\) which only depend on the tree itself, and then extract the corresponding value when looping over features at leaf nodes to calculate \(\phi_i\) 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 \(M > 2^{D_k+1}/D\) (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 \(\{U_{D_k,C} : C \subseteq D_k\}\) for all \(D_k\)’s in the tree, and Fast TreeSHAP Score (Algorithm 4 in Appendix A.6) calculates \(\phi_i\’s\) for all samples to be explained based on the pre-computed \(\{U_{D_k,C} : C \subseteq D_k\}\). The main output of Fast TreeSHAP Prep is \(S\), an \(L \times 2^D\) matrix where each row records the values in \(\{U_{D_k,C} : C \subseteq D_k\}\) 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 \(C \subseteq D_k\) 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 \(C \subseteq D_k\) 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 \(O(MTLD^2)\) 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 I 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.

<table>
<thead>
<tr>
<th>TreeSHAP Version</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original TreeSHAP</td>
<td>(O(MLD^2))</td>
<td>(O(D^2 +</td>
</tr>
<tr>
<td>Fast TreeSHAP v1</td>
<td>(O(MLD^2))</td>
<td>(O(D^2 +</td>
</tr>
<tr>
<td>Fast TreeSHAP v2 (general case)</td>
<td>(O(TL^2D + MLD))</td>
<td>(O(L^2))</td>
</tr>
<tr>
<td>Fast TreeSHAP v2 (balanced trees)</td>
<td>(O(TL^2D + MLD))</td>
<td>(O(L^2))</td>
</tr>
</tbody>
</table>

\(^1\)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 \(M > \frac{2^{D+1}}{D}\) (commonly occurs in a moderate-sized dataset, as most tree-based models produce trees with depth \(\leq 16\)). Also, Fast TreeSHAP v2 is under a stricter memory constraint: \(O(L^2D) < \text{memory tolerance}\). 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 upperbound. 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 2 with the goal of evaluating a wide range of tree ensembles representative of different real-world settings. While the first three datasets Adult [15], Superconductor [10], and Crop [13, 14] in Table 2 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.

Table 2: Datasets.

<table>
<thead>
<tr>
<th>Name</th>
<th># Instances</th>
<th># Attributes (Original)</th>
<th># Attributes (One-Hot)</th>
<th>Task</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adult[15]</td>
<td>48,842</td>
<td>14</td>
<td>64</td>
<td>Classification</td>
<td>2</td>
</tr>
<tr>
<td>Superconductor[10]</td>
<td>21,263</td>
<td>81</td>
<td>81</td>
<td>Regression</td>
<td>-</td>
</tr>
<tr>
<td>Crop[13, 14]</td>
<td>325,834</td>
<td>174</td>
<td>174</td>
<td>Classification</td>
<td>7</td>
</tr>
<tr>
<td>Upsell</td>
<td>96,120</td>
<td>169</td>
<td>182</td>
<td>Classification</td>
<td>2</td>
</tr>
</tbody>
</table>

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
Table 3: TreeSHAP vs Fast TreeSHAP v1 vs Fast TreeSHAP v2 - 10,000 samples (s.d. in parenthesis).

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

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 3 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 ($\sim 4$) 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 3 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 4 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. Finally, the last column shows the space allocation of matrix $S$ which dominates
Table 4: Fast TreeSHAP v2: Fast TreeSHAP Prep & Fast TreeSHAP Score - 10,000 samples (s.d. in parathesis).

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

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

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 4 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.

We also compare the performance of original TreeSHAP, Fast TreeSHAP v1 and Fast TreeSHAP v2 on varying number of trees in Section A.8. We find that the execution time is approximately linear in

---

2Space allocation of $S$ is calculated by $\#\text{maximum leaves} \cdot \#\text{maximum depth} \cdot 8B$ for double type entries.
terms of the number of trees for all the three approaches, which justifies the linear term of number of trees in the theoretical complexity formulas in Table 1. We also find that the relative performance of the three approaches is consistent with the number of trees. In addition to this comparison, we also compare the performance of the three approaches on other types of tree-based models, e.g., XGBoost model in Section A.9. Overall we see that Fast TreeSHAP v1 and v2 can still achieve ~1.5x and ~2.5-3x speedups over the original TreeSHAP on XGBoost model.

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.10. 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.11.

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 $O(TL^2D + MTLD)$ with a small sacrifice on memory cost, and is well-suited for multi-time model explanation scenarios. We note that in practice, the computational advantage of Fast TreeSHAP v2 is achieved for sufficiently large value of $M$ (number of samples), and in the meantime $D$ (maximum depth of tree) should be upper-bounded according to the memory constraint.

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.

Acknowledgments and Disclosure of Funding

We would like to express our sincere thanks to Humberto Gonzalez, Diana Negoescu, and Wenrong Zeng for their helpful comments and feedback, and Parvez Ahammad for his support throughout this project.
References


A Appendix

A.1 Link to Code Repository

Here is the link to the code repository to replicate the results in Section 5: [https://drive.google.com/drive/u/4/folders/1P0V0Vj42K04QcT39wmbMM4Y6joyc67ZQ](https://drive.google.com/drive/u/4/folders/1P0V0Vj42K04QcT39wmbMM4Y6joyc67ZQ)

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

Algorithm 1 proposed in [19, 18] 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. 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 $f_S(x)$

```plaintext
procedure EXPVALUE(x, S, tree = \{v, a, b, t, r, d\})
    return G(1)  \quad \triangleright \text{Start recursion at the root node}
end procedure

procedure G(j)
    if v_j ≠ internal then
        return v_j  \quad \triangleright \text{Check if node j is a leaf}
    end if
    if d_j ∈ S then
        return G(a_j) if x_{d_j} ≤ t_j else G(b_j)  \quad \triangleright \text{Check if the split feature is in S}
    else
        return G(a_j)r_{a_j}/r_j + G(b_j)r_{b_j}/r_j  \quad \triangleright \text{Use value of child on the decision path}
    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

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

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

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

$^3$A node in a tree can be either an internal node or a leaf node.
Similarly, ∀l ∈ D_k where l ≠ i, ∀S ⊂ N \{i, l\}, we have W_{k,S∪\{i,l\}} = W_{k,S∪\{i\}} and W_{k,S∪\{l\}} = W_{k,S}, thus W_{k,S∪\{i,l\}} - W_{k,S∪\{i\}} = W_{k,S∪\{l\}} - W_{k,S}. Therefore, ∀l ∈ D_k,

\[
\phi_l = \sum_{k:i\in D_k} \sum_{S\subset N \{i,l\}} \frac{|S|!(|N|-|S|)!}{|N|!} (W_{k,S∪\{i\}} - W_{k,S})
\]

\[
+ \sum_{S\subset N \{i,l\}} \frac{(|S|+1)!(|N|-|S|)!}{|N|!} (W_{k,S∪\{i,l\}} - W_{k,S∪\{i\}}) v_k
\]

\[
= \sum_{k:i\in D_k} \left[ \sum_{S\subset N \{i,l\}} \left( \frac{|S|!(|N|-|S|)!}{|N|!} + \frac{(|S|+1)!(|N|-|S|)!}{|N|!} \right) (W_{k,S∪\{i\}} - W_{k,S}) \right] v_k
\]

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

\[
\phi_l = \sum_{k:i\in D_k} \left[ \sum_{S\subset D_k \{i\}} \frac{|S|!(|D_k|-|S|)!}{|D_k|!} (W_{k,S∪\{i\}} - W_{k,S}) \right] v_k.
\]

Plugging in W_{k,S} = \prod_{j\in D_k \cup S} 1 \{x_{d_j} \in T_{k_j}\} \cdot \prod_{j\in S} R_{k_j} into the above equation, we have

\[
\phi_l = \sum_{k:i\in D_k} \left[ \sum_{S\subset D_k \{i\}} \frac{|S|!(|D_k|-|S|)!}{|D_k|!} \left( \prod_{j\in S \cup S_k} 1 \{x_{d_j} \in T_{k_j}\} \prod_{j\in D_k \cup S_k} R_{k_j} \right) \right] v_k.
\]

\[
= \sum_{k:i\in D_k} \left[ \sum_{m=0}^{D_k-1} \frac{m!(|D_k|-m-1)!}{|D_k|!} \sum_{S\subset D_k \{i\}, |S|=m} \left( \prod_{j\in S \cup S_k} 1 \{x_{d_j} \in T_{k_j}\} \prod_{j\in D_k \cup S_k} R_{k_j} \right) \right] v_k.
\]

\[
= \sum_{k:i\in D_k} \left[ \sum_{m=0}^{D_k-1} \frac{m!(|D_k|-m-1)!}{|D_k|!} \sum_{S\subset D_k \{i\}, |S|=m} \left( \prod_{j\in S \cup S_k} 1 \{x_{d_j} \in T_{k_j}\} \prod_{j\in D_k \cup S_k} R_{k_j} \right) \right] v_k.
\]

\[
\text{A.4 Proof of Theorem 2}
\]

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

\[
\phi_l = \sum_{k:i\in D_k} \left[ \sum_{m=0}^{D_k-1} \frac{m!(|D_k|-m-1)!}{|D_k|!} \sum_{S\subset D_k \{i\}, |S|=m} \left( \prod_{j\in S \cup S_k} 1 \{x_{d_j} \in T_{k_j}\} \prod_{j\in D_k \cup S_k} R_{k_j} \right) \right] v_k.
\]
I.e., only subsets $S \subset D_k \setminus \{\{i\} \cup S_k\}$ are necessary to be included in the equation.

Remind that $S_k = \{d_j : j \in P_k, x_{d_j} \notin T_k\}$, 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 $U_{m,D_k,C} := \frac{m!([D_k]-m)!}{(m+1)!}\sum_{S \subset C,|S|=m}\sum_{j \in P_k, d_j \in C \setminus S} R_{kl}$, $m = 0, \cdots , |C|$, and $U_{D_k,C} := \sum_{m=0}^{|C|} |D_k|-m U_{m,D_k,C}$, where $C \subset D_k$ is a subset of $D_k$.

Then it is easy to see that when $\prod_{j \in P_k, d_j = i} I \{x_{d_j} \in T_k\} = 0$ (i.e., $i \notin S_k$), we have $S_k \cup \{i\} = S_k$, and

$$\phi_i = \sum_{k:i \in D_k} \frac{|D_k|-|S_k|}{|D_k|!} m!([D_k]-m-1)! \left( \sum_{S \subset D_k \setminus S_k, |S|=m} \prod_{j \in P_k, d_j \in D_k \setminus (S \cup S_k)} R_{kl} \right)$$

$$= - \sum_{k:i \in D_k} \frac{|D_k|-|S_k|}{|D_k|!} m!([D_k]-m-1)! \sum_{j \in P_k, d_j \in S_k} R_{kl} v_k$$

$$= - \sum_{k:i \in D_k} U_{D_k,D_k \setminus S_k} \prod_{j \in P_k, d_j \in S_k} R_{kl} v_k.$$

When $\prod_{j \in P_k, d_j = i} I \{x_{d_j} \in T_k\} = 1$ (i.e., $i \in S_k$), we have $S_k = S_k$, and

$$\phi_i = \sum_{k:i \in D_k} \frac{|D_k|-|S_k|}{|D_k|!} m!([D_k]-m-1)! \left( \sum_{S \subset D_k \setminus (\{i\} \cup S_k), |S|=m} \prod_{j \in P_k, d_j \in D_k \setminus (S \cup \{i\} \cup S_k)} R_{kl} \right)$$

$$= \sum_{k:i \in D_k} \frac{|D_k|-|S_k|}{|D_k|!} m!([D_k]-m-1)! \left( \sum_{j \in P_k, d_j = i} R_{kl} \left(1 - \prod_{j \in P_k, d_j = i} R_{kl} \right) v_k \right)$$

$$= \sum_{k:i \in D_k} U_{D_k,D_k \setminus (\{i\} \cup S_k)} \prod_{j \in P_k, d_j \in S_k} R_{kl} v_k.$$

\[\square\]

### A.5 Fast TreeSHAP v1 Algorithm Details

In Algorithm 2, $m$ is the path to 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 $j$th internal node (i.e., $d_j$), ii) $m_{j+1,z}$, the covering ratio for the $j$th internal node (i.e., $R_{kl}$), iii) $m_{j+1,o}$, the threshold condition for the $j$th internal node (i.e., $I \{x_{d_j} \in T_k\}$) $(m_0,d, m_0,z, m_0,o$ record the information of a dummy node). Moreover, at the $j$th internal node, $w$ is used to record $U_{m,D_k,D_k \setminus S_k}$ $(m = 0, \cdots , j - |S_k|)$ where $D_k$ and $S_k$ are subsets of $D_k$ up to the $j$th internal node, and $q$ is used to record $\prod_{j \in P_k, d_j = i} R_{kl}$ where $P_k$ is the subpath of $P_k$ up to the $j$th 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 $p_t$ 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 TREEHAP(x, tree = \{v, a, b, t, r, d\})
    \(\phi = \text{array of len}(x)\) zeros
    RECURSE(0, [], [], 1, 1, 1, -1) \(\triangleright\) Start recursion at the root node
    return \(\phi\)
end procedure

procedure RECURSE(j, m, w, q, p, p_o, p_i)
    m, w, q = EXTEND(m, w, q, p, p_o, p_i) \(\triangleright\) Update m, w, q for the feature of the last split
    if \(v_j \neq \text{internal}\) then
        if len(m) > len(w) then
            \(s_0 = -\text{sum}(\text{UNWIND}(m, w, q, -1)[1])\) \(\triangleright\) Pre-calculate \(U_{D_k, D_h\setminus S_h}\) for \(d_i\)'s in \(S_k\)
        end if
        for \(i \leftarrow 1\) to len(m) - 1 do
            \(\phi_{m, o} = \phi_{m, o} + s_0 \cdot q \cdot v_j\) \(\triangleright\) Update contribution for feature \(d_i \in S_k\)
        end if
    else
        if \(x \leq t_j\) then \(\triangleright\) Determine hot and cold children
            \(i_o = i_o = 1\)
            \(k = \text{FINDFIRST}(m, d_j)\)
            if \(k \neq \text{nothing}\) then \(\triangleright\) 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 \cdot r_h/r_j, i_o, d_j)
            RECURSE(c, m, w, q, i_z \cdot r_c/r_j, 0, d_j)
        end if
        else
            \(h, c = (a_j, b_j)\) if \(x = t_j\) else \((b_j, a_j)\)
            if \(p_o = 0\) then
                \(q = q \cdot p_z\)
                for \(i \leftarrow l_w - 1\) to 0 do
                    \(w_i = w_i \cdot (l - i)/(l + 1)\)
                end for
            else
                \(w_i = 1\) if \(l_w = 0\) else 0 \(\triangleright\) Extend \(w\) if the feature of the last split is not in \(S_k\)
                for \(i \leftarrow l_w - 1\) to 0 do
                    \(w_{i+1} = w_{i+1} + w_i \cdot (i + 1)/(l + 1)\)
                    \(w_i = p_z \cdot w_i \cdot (l - i)/(l + 1)\)
                end for
            end if
        end if
    end if
end procedure

procedure EXTEND(m, w, q, p, p_o, p_i)
    \(l_w = \text{len}(m), l_w = \text{len}(w)\)
    m = copy(m), copy(w)
    \(m_i, (d, z, o) = (p_i, p_z, p_o)\) \(\triangleright\) Extend \(m\) given the feature of the last split
    if \(p_o = 0\) then
        \(q = q \cdot p_z\)
        for \(i \leftarrow l_w - 1\) to 0 do
            \(w_i = w_i \cdot (l - i)/(l + 1)\)
        end for
    else
        \(w_i = 1\) if \(l_w = 0\) else 0 \(\triangleright\) Extend \(w\) if the feature of the last split is not in \(S_k\)
        for \(i \leftarrow l_w - 1\) to 0 do
            \(w_{i+1} = w_{i+1} + w_i \cdot (i + 1)/(l + 1)\)
            \(w_i = p_z \cdot w_i \cdot (l - i)/(l + 1)\)
        end for
    end if
end procedure

procedure UNWIND(m, w, q, i)
    \(l_w = \text{len}(m) - 1, l_w = \text{len}(w) - 1\)
    m = copy(m-1)
    if \(i < 0\) then
        \(w = \text{copy}(w)\)
        for \(j \leftarrow l_w\) to 0 do
            \(w_j = w_j \cdot (l + 1)/(l - j)\)
        end for
    end if
end procedure
```
where each row records the number of nonzero elements in each row of $D$. $d$ is a vector of size $P$ to a subset $C$.

In Algorithm 3 and 4, $A.6$ 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 $\{U_{D_k,C} : C \subseteq D_k\}$ for one path $P_k$, and the subsets $C$ in each row are ordered according to the reverse ordering of binary digits. $W$ is a matrix of size $2^D \times (D + 1)$, where each row corresponds to a subset $C \subseteq D_k$ ordered according to the reverse ordering of binary digits. At the $j$th internal node along path $P_k$, each row of $W$ records $\{U_{m,D_k,C} : m = 0, \cdots, |C|\}$ for a subset $C \subseteq D_{kj}$ where $D_{kj}$ is the subset of $D_k$ up to the $j$th 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

```plaintext
procedure TREESHOPPREP(tree = \{v, a, b, t, r, d\})
    $S$ = matrix of $L \times 2^D$ zeros
    $e$ = array of $\text{len}(d)$ negative ones
    $W$ = matrix of $2^D \times (D + 1)$ zeros
    $l_w$ = array of $2^D$ zeros
    $c$ = 0
    RECURSE(0, [], $W$, $l_w$, 1, -1)  \triangleright Start recursion at the root node
    return $S$, $e$
end procedure
```

```plaintext
procedure RECURSE($j, m, W, l_w, p_z, p_i$)
    $m, W, l_w = \text{EXTEND}(m, W, l_w, p_z, p_i)$  \triangleright Update $m, W, l_w$ for the feature of the last split
    if $v_j$ \neq \text{internal} then
        $l = \text{len}(m) - 1$
        for $t \leftarrow 0$ to $2^l - 1$ do
            \triangleright Calculate $\{U_{D_k,C} : C \subseteq D_k\}$ for the current path
            for $i \leftarrow l_w[t]$ to 0 do
                $S[c, t] = S[c, t] + W[t, i] \cdot (l + 1)/(l - i)$
            end for
        end for
    end if
end procedure
```

$^3$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 $\{(0, 0, 0), (1, 0, 0), (0, 1, 0), (1, 1, 0), (0, 0, 1), (1, 0, 1), (0, 1, 1), (1, 1, 1)\}$.  

17
\( c = c + 1 \)

\( \text{else} \)
\( i_z = 1 \)
\( k = \text{FINDFIRST}(m.d, d_j) \)
\( \text{if } k \neq \text{nothing} \) then
\( e_j = k \)
\( i_z = m_k.z \)
\( m, W, l_w = \text{UNWIND}(m, W, l_w, k) \) \( \triangleright \) Undo previous update for \( d_k \) if we see it again
\( \text{end if} \)
\( l_w = l_w + 1 \)
\( \text{RECURSE}(a_j, m, W, l_w, i_z r_a/j, r_j, d_j) \)
\( \text{RECURSE}(b_j, m, W, l_w, i_z r_b/j, d_j) \)
\( \text{end if} \)
\( \text{end procedure} \)

\( \text{procedure} \) EXTEND\((m, W, l_w, p_z, p_i) \) \( \triangleright \) Update \( m, W, l_w \) for the feature of the last split
\( l = \text{len}(m) \)
\( m = \text{copy}(m) \)
\( m_z((d, z) = (p_i, p_z)) \) \( \triangleright \) Extend \( m \) given the feature of the last split
\( \text{if } l = 0 \) then
\( W[0,0] = 1 \)
\( \text{else} \)
\( W, l_w = \text{copy}(W), \text{copy}(W), \text{copy}(l_w) \)
\( W[2^{l-1} : 2^l, :] = \text{copy}(W[; : 2^{l-1}, :]) \)
\( l_w[2^{l-1} : 2^l] = \text{copy}(l_w[; 2^{l-1}]) \)
\( \text{for } t \leftarrow 0 \text{ to } 2^{l-1} - 1 \text{ do} \)
\( \quad \text{for } i \leftarrow l_w[t] - 1 \text{ to } 0 \text{ do} \)
\( \qquad W[t,i] = W[t,i] \cdot (l-i)/(l+1) \)
\( \text{end for} \)
\( \text{end for} \)
\( \text{for } t \leftarrow 2^{l-1} \text{ to } 2^l - 1 \text{ do} \)
\( \quad \text{for } i \leftarrow l_w[t] - 1 \text{ to } 0 \text{ do} \)
\( \qquad W[t,i+1] = W[t,i+1] + W[t,i] \cdot (i + 1)/(l+1) \)
\( \qquad W[t,i] = p_z \cdot W[t,i] \cdot (l-i)/(l+1) \)
\( \text{end for} \)
\( \text{end for} \)
\( l_w[; 2^{l-1}] = l_w[; 2^{l-1}] - 1 \)
\( \text{end if} \)
\( \text{return } m, W, l_w \)
\( \text{end procedure} \)

\( \text{procedure} \) UNWIND\((m, W, l_w, i) \) \( \triangleright \) Undo previous update for feature \( d_i \)
\( l = \text{len}(m) - 1 \)
\( m, W, l_w = \text{copy}(m_0, ..., m_{l-1}), \text{copy}(W), \text{copy}(l_w) \)
\( \text{ind} = \text{empty array} \)
\( \text{for } t_1 \leftarrow \{0, 2, 2 \cdot 2, 3 \cdot 2^i, \ldots, 2^l - 2^i \} \text{ do} \)
\( \quad \text{end for} \)
\( \text{for } t_2 \leftarrow t_1 \text{ to } t_1 + 2^{l-1} - 1 \text{ do} \)
\( \quad \text{end for} \)
\( W[; : 2^{l-1}, :] = W[\text{ind}, :) \]
\( l_w[; 2^{l-1}] = l_w[\text{ind}] \)
\( \text{for } t \leftarrow 0 \text{ to } 2^{l-1} - 1 \text{ do} \)
\( \quad \text{for } i \leftarrow l_w[t] \text{ to } 0 \text{ do} \)
\( \qquad W[t,i] = W[t,i] \cdot (l+1)/(l-i) \)
\( \text{end for} \)
\( \text{end for} \)
\( \text{for } j \leftarrow i \text{ to } l - 1 \text{ do} \)
\( \quad m_j((d, z) = m_{j+1}((d, z)) \) \( \triangleright \) Shrink \( m \) given feature \( d_i \)
\( \text{end for} \)
return \( m, W, l_w \)
end procedure

---

**Algorithm 4** Fast TreeSHAP v2: Score

```plaintext
procedure TREEHAPSCORE(x, tree = \{v, a, b, t, r, d, S, e\})
φ = array of \( len(x) \) zeros
b = array of \([0, 1, 2, 2^2, \ldots, 2^D]\)
c = 0
RECURSE(0, [1, 1, 1, 1, -1]) \( \triangleright \) 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_i.(d, z, o) = (p_i, p_o, p)
\( \triangleright \) Update \( m, q \) for the feature of the last split
if \( p_o = 0 \) then
q = q \cdot p
end if
if \( v_j \neq \text{internal} \) then
\( b_{\text{sum}} = 0 \)
for \( i \leftarrow 1 \) to \( l \) do
\( b_{\text{sum}} = b_{\text{sum}} + b_i \) if \( m_i.o \neq 0 \)
end for
for \( i \leftarrow 1 \) to \( l \) do
if \( m_i.o = 0 \) then
φ_{m_i.d} = φ_{m_i.d} - S[ε, b_{\text{sum}}] \cdot q \cdot v_j
else
φ_{m_i.d} = φ_{m_i.d} + S[ε, b_{\text{sum}} - b_i] \cdot q \cdot (1 - m_i.o) \cdot v_j
end if
end for
c = c + 1
else
i_z = i_o = 1
k = e_j
if \( k \neq -1 \) then
\( i_z, i_o = m_k.z, m_k.o \)
m = copy(m_{0..l-1})
for \( j \leftarrow k \) 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 \cdot 1(x_d \leq t_j), d_j)
RECURSE(b_j, m, q, i_z r_{b_j}/r_j, i_o \cdot 1(x_d > t_j), d_j)
end if
end procedure
```

---

A.7 Additional Tables in Evaluation

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 5: Models.

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

Figure 2: Execution time v.s. number of trees. Left: Adult-Med, Right: Adult-Large.

A.8 Evaluation on Varying Number of Trees

We compare the performance of original TreeSHAP, Fast TreeSHAP v1 and Fast TreeSHAP v2 on varying number of trees. Figure 2 plots the execution time versus number of trees for models Adult-Med and Adult-Large. 95% confidence interval of the execution time is also indicated by the shaded area. We find that in both plots, the execution time is approximately linear in terms of the number of trees for all the three approaches, which justifies the theoretical complexity formulas in Table 1. We also find that the relative performance of the three approaches is consistent with the number of trees, where Fast TreeSHAP v2 consistently performs the best, while original TreeSHAP consistently performs the worst.

A.9 Evaluation on XGBoost Model

We compare the performance of original TreeSHAP, Fast TreeSHAP v1 and Fast TreeSHAP v2 on XGBoost [6] models trained on datasets Adult [15] and Superconductor [10]. For fair comparison, all the evaluations were run on a single core in Azure Virtual Machine with size Standard_D8_v3 (8 cores and 32GB memory). We construct Table 6, 7, and 8 in the same format of Table 5, 3, and 4. Overall we see that Fast TreeSHAP v1 and v2 can still achieve ~1.5x and ~2.5-3x speedups than original TreeSHAP.
Table 6: Models.

<table>
<thead>
<tr>
<th>Model</th>
<th># Trees</th>
<th>Max Depth</th>
<th># Leaves</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adult-Small</td>
<td>100</td>
<td>4</td>
<td>1,480</td>
</tr>
<tr>
<td>Adult-Med</td>
<td>100</td>
<td>8</td>
<td>7,375</td>
</tr>
<tr>
<td>Adult-Large</td>
<td>100</td>
<td>12</td>
<td>21,085</td>
</tr>
<tr>
<td>Adult-xLarge</td>
<td>100</td>
<td>16</td>
<td>36,287</td>
</tr>
<tr>
<td>Super-Small</td>
<td>100</td>
<td>4</td>
<td>1,512</td>
</tr>
<tr>
<td>Super-Med</td>
<td>100</td>
<td>8</td>
<td>12,802</td>
</tr>
<tr>
<td>Super-Large</td>
<td>100</td>
<td>12</td>
<td>47,840</td>
</tr>
<tr>
<td>Super-xLarge</td>
<td>100</td>
<td>16</td>
<td>108,010</td>
</tr>
</tbody>
</table>

Table 7: TreeSHAP vs Fast TreeSHAP v1 vs Fast TreeSHAP v2 - 10,000 samples (s.d. in parathesis).

<table>
<thead>
<tr>
<th>Model</th>
<th>Original TreeSHAP (s)</th>
<th>Fast TreeSHAP v1 (s)</th>
<th>Speedup</th>
<th>Fast TreeSHAP v2 (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adult-Small</td>
<td>2.03 (0.05)</td>
<td>1.92 (0.05)</td>
<td>1.06</td>
<td>0.85 (0.01)</td>
<td>2.39</td>
</tr>
<tr>
<td>Adult-Med</td>
<td>23.61 (0.11)</td>
<td>18.83 (0.14)</td>
<td>1.25</td>
<td>7.65 (0.07)</td>
<td>3.09</td>
</tr>
<tr>
<td>Adult-Large</td>
<td>113.77 (0.44)</td>
<td>83.58 (0.55)</td>
<td>1.36</td>
<td>36.98 (0.19)</td>
<td>3.08</td>
</tr>
<tr>
<td>Adult-xLarge</td>
<td>256.32 (1.24)</td>
<td>183.07 (1.65)</td>
<td>1.40</td>
<td>78.02 (0.33)</td>
<td>3.29</td>
</tr>
<tr>
<td>Super-Small</td>
<td>2.61 (0.10)</td>
<td>2.42 (0.10)</td>
<td>1.08</td>
<td>1.12 (0.06)</td>
<td>2.33</td>
</tr>
<tr>
<td>Super-Med</td>
<td>47.00 (1.64)</td>
<td>33.68 (0.22)</td>
<td>1.40</td>
<td>17.45 (0.12)</td>
<td>2.69</td>
</tr>
<tr>
<td>Super-Large</td>
<td>334.28 (12.46)</td>
<td>221.38 (2.63)</td>
<td>1.51</td>
<td>105.85 (2.71)</td>
<td>3.16</td>
</tr>
<tr>
<td>Super-xLarge</td>
<td>1124.07 (4.50)</td>
<td>720.21 (7.84)</td>
<td>1.56</td>
<td>445.49 (5.03)</td>
<td>2.52</td>
</tr>
</tbody>
</table>

A.10 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(L_2D)) 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(BL_2D) 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.11 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.10 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 3 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 4, we can reasonably estimate that the matrix \(S\) occupies about \(400 \times 30\) MB = 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 4 that Fast TreeSHAP v2 provides \(\sim 3\)x 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 [13, 14], 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 3, 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 \(15 \times 4\) MB = 60MB in memory, and can further reduce the execution time to \(130 \times 32 \times 5 / 4\) s = 1.4h.

<table>
<thead>
<tr>
<th>Model</th>
<th>Original TreeSHAP (s)</th>
<th>Fast TreeSHAP Prep (s)</th>
<th>Fast TreeSHAP Score (s)</th>
<th>Speedup (Large (M))</th>
<th>Space Allocation of (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adult-Small</td>
<td>2.03 (0.05)</td>
<td>&lt;0.01 (&lt;0.01)</td>
<td>0.85 (0.01)</td>
<td>2.39</td>
<td>2KB</td>
</tr>
<tr>
<td>Adult-Med</td>
<td>23.61 (0.11)</td>
<td>0.05 (&lt;0.01)</td>
<td>7.60 (0.07)</td>
<td>3.11</td>
<td>258KB</td>
</tr>
<tr>
<td>Adult-Large</td>
<td>113.77 (0.44)</td>
<td>1.03 (0.02)</td>
<td>35.95 (0.19)</td>
<td>3.16</td>
<td>14.2MB</td>
</tr>
<tr>
<td>Adult-xLarge</td>
<td>256.32 (1.24)</td>
<td>6.76 (0.08)</td>
<td>71.26 (0.33)</td>
<td>3.60</td>
<td>421MB</td>
</tr>
<tr>
<td>Super-Small</td>
<td>2.61 (0.10)</td>
<td>&lt;0.01 (&lt;0.01)</td>
<td>1.12 (0.06)</td>
<td>2.33</td>
<td>2KB</td>
</tr>
<tr>
<td>Super-Med</td>
<td>47.00 (1.64)</td>
<td>0.15 (0.01)</td>
<td>17.30 (0.12)</td>
<td>2.72</td>
<td>392KB</td>
</tr>
<tr>
<td>Super-Large</td>
<td>334.28 (12.46)</td>
<td>7.42 (0.06)</td>
<td>98.43 (2.71)</td>
<td>3.40</td>
<td>27.7MB</td>
</tr>
<tr>
<td>Super-xLarge</td>
<td>1124.07 (4.50)</td>
<td>147.00 (3.87)</td>
<td>298.49 (5.03)</td>
<td>3.77</td>
<td>936MB</td>
</tr>
</tbody>
</table>