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.

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] [4] [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.

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 ≥ 10) still remains a computational concern in practice.

Preprint. Under review.
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.10).
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, 1, 7, 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 treeshap 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: \( g(z') = \phi_0 + \sum_{j \in N} \phi_j z'_j \). Here \( z' \in \{0,1\}^{|N|} \) represent a feature being observed \( (z'_j = 1) \) or unknown \( (z'_j = 0) \), and \( \phi_j \in \mathbb{R} \) 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:

\[
\phi_i = \sum_{S \in \mathcal{N}\backslash\{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} (f_{S\cup\{i\}}(x) - f_S(x)).
\]

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 \( S = N\backslash S \). While it is still debating which option is more appropriate [11, 17, 20, 12, 23, 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 [19, 18]. The authors of [19, 18] define a conditional expectation \( f_S(x) = E[f(x)|x_S] = E_{x_S|x_S}[f(x)] \), where \( x_S \) is the sub-vector of \( x \) restricted to the feature subset \( S \), and \( S = N\backslash S \). While it is still debating which option is more appropriate [11, 17, 20, 12, 23, 5], both options need exponential time in computation as Equation 1 considers all possible subsets in \( N \).

3.3 TreeSHAP

TreeSHAP proposed in [19, 18] runs in \( O(MTL^2D^2) \) 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 \( v_{u_k} \) to \( v_{v_k} \), corresponding to \( K \) paths \( P_j \) 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 \( T_{kj} = (\neg \infty, t_{kj}] \)
We now further simplify Equation 2 for an even faster computational speed. Consider a subset weight for subset size \(\phi\) where

\[
\phi = \sum_{k=1}^{K} \prod_{j \in P_k, d_j \in S} \mathbbm{1}\{x_{d_j} \in T_{kj}\} \cdot \prod_{j \in P_k, d_j \notin S} R_{kj} \cdot v_k,
\]

where \(\mathbbm{1}\{\cdot\}\) is an indicator function. Let \(W_{k,S} := \prod_{j \in P_k, d_j \in S} \mathbbm{1}\{x_{d_j} \in T_{kj}\} \cdot \prod_{j \in P_k, d_j \notin S} R_{kj}\), i.e., \(W_{k,S}\) is the "proportion" of subset \(S\) that flows down into the leaf node \(v_k\), then

\[
f_S(x) = \frac{\sum_{k=1}^{K} W_{k,S} v_k}{|S|!(|N|−|S|−1)!} \sum_{k=1}^{K} (W_{k,S∪\{i\}} − W_{k,S}) v_k.
\]

**Theorem 1.** \(\phi_i\) 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,

\[
\phi_i = \sum_{k : i \in D_k} \left[ \frac{m!([D_k]−m)!}{[D_k]!} \sum_{m=0}^{[D_k]-1} \prod_{S \subseteq D_k \setminus \{i\}, |S|=m} \left( \prod_{j \in P_k, d_j \in S} \mathbbm{1}\{x_{d_j} \in T_{kj}\} \prod_{j \in P_k, d_j \notin S} R_{kj} \right) \right] \cdot \left( \prod_{j \in P_k, d_j = i} \mathbbm{1}\{x_{d_j} \in T_{kj}\} - \prod_{j \in P_k, d_j \notin D_k} R_{kj} \right) v_k,
\]

(2)

The proof of Theorem 1 can be found in Appendix A.2. For convenience, we call \(\frac{m!([D_k]−m)!}{[D_k]!}\) 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" 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" of all feature subsets of size \(1, 2, \ldots\) up to the current depth. At the leaf node, EXTEND reaches the sequence of values \(\frac{m!([D_k]−m)!}{[D_k]!}\sum_{S \subseteq D_k, |S|=m} \prod_{j \in P_k, d_j \in S} \mathbbm{1}\{x_{d_j} \in T_{kj}\} \prod_{j \in P_k, d_j \notin S} R_{kj}\) for each possible subset size \(m = 1, \ldots, |D_k|\). 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 \(S_k \subseteq D_k\) 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 \(U_{m,D_k,C} := \frac{m!([D_k]−m)!}{[D_k]!} \sum_{S \subseteq C, |S|=m} \prod_{j \in P_k, d_j \in S} \mathbbm{1}\{x_{d_j} \in T_{kj}\} \prod_{j \in P_k, d_j \notin S} R_{kj}\) for \(m = 0, \ldots, |C|\), where \(C \subseteq D_k\) 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" 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 \(U_{D_k,C} := \sum_{m=0}^{[D_k]+1} \frac{m!([D_k]−m)!}{[D_k]!} U_{m,D_k,C}\). Theorem 2 further simplifies Equation 2 (proof in Appendix A.3).

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

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

(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.4), 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” 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 it contributes to the sum of “proportion” of all subset sizes of subset 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 $U_{m,D_k,D_k \setminus (i) \setminus S_k}$ 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 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., $E[|D_k| - |S_k|]/|D_k| \approx \sum_{i=0}^{D} C_i^D (D - i)/(2^D D) \approx 50%$. 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% x 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 $O(TL2^D + MTLD^2)$ time and $O(L2^D)$ memory. For balanced trees it becomes $O(TL^2D + MTLD^2)$ 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.
We split Fast TreeSHAP v2 algorithm into two parts: Fast TreeSHAP Prep and Fast TreeSHAP Score. Fast TreeSHAP Prep (Algorithm 3 in Appendix A.5) 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.5) calculates \( \phi_i \)'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 \( \{U_{m,D_k,C} : m = 0, \ldots, |C|\} \) 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 subset sizes 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            | $O(MTLD^2)$          | $O(D^2 + |N|)$         |
| Fast TreeSHAP v2 (general)  | $O(TL^2D + MTLD)$     | $O(L^2D)$             |
| Fast TreeSHAP v2 (balanced) | $O(TL^2D + MTLD)$     | $O(L^2)$              |

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 $\leq 16$). 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 upper bound. 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.6.

Table 2: Datasets.

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

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 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)         | 2.25    |
| 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    |

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 down 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: 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                   |

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

reflects the $D$-time improvement in computational complexity between Fast TreeSHAP Score and original TreeSHAP. Finally, the last column shows the space allocation of matrix $S$ which dominates the memory usage of Fast TreeSHAP v2. 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.

$^2$Space allocation of $S$ is calculated by $\#\text{maximum leaves} \cdot 2^{\#\text{maximum depth}} \cdot 8B$ for double type entries.
We also compare the performance of original TreeSHAP, Fast TreeSHAP v1 and Fast TreeSHAP v2 on varying number of trees in Section A.7. We find that the execution time is approximately linear in 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.8. 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.9. 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.10.

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

[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 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] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pages 785–794, 2016.

[7] Christopher Frye, Damien de Mijolla, Tom Begley, Laurence Cowton, Megan Stanley, and Illya Feige. Shapley explainability on the data manifold. arXiv preprint arXiv:2006.01727, 2020.

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] Konrad Komisarczyk, Paweł Kozminski, Szymon Maksymiuk, and Przemysław Biecek. treeshap. https://github.com/ModelOriented/treeshap, 2020.

[17] 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.

[18] 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.

[19] Scott M Lundberg, Gabriel G Erion, and Su-In Lee. Consistent individualized feature attribution for tree ensembles. arXiv preprint arXiv:1802.03888, 2018.
[20] 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.

[21] Scott M Lundberg, Bala Nair, Monica S Vavilala, Mayumi Horibe, Michael J Eissses, 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.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

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

[34] 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.
A Appendix

A.1 Algorithm Details of Estimating $f_S(x)$

Algorithm 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 $f_S(x)$

| Procedure EXPVALUE($x, S, tree = \{v, a, b, t, r, d\}$) |
|-----------------------------------------------|
| return $G(1)$                                |
| $\triangleright$ Start recursion at the root node |

| Procedure $G(j)$ |
|------------------|
| if $v_j \neq$ internal then |
| return $v_j$ |
| $\triangleright$ Check if node $j$ is a leaf |
| else |
| if $d_j \in S$ then |
| return $G(a_j)$ if $x_{d_j} \leq t_j$ else $G(b_j)$ |
| $\triangleright$ Check if the split feature is in $S$ |
| else |
| return $G(a_j)r_{a_j}/r_j + G(b_j)r_{b_j}/r_j$ |
| $\triangleright$ Use value of child on the decision path |
| end if |
| end if |

A.2 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

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

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

3 A node in a tree can be either an internal node or a leaf node.
Similarly, $\forall l \notin D_k$ where $l \neq i, \forall S \subset N \setminus \{i, l\}$, we have $W_{k,S \cup \{i,l\}} = W_{k,S \cup \{i\}}$ and $W_{k,S \cup \{l\}} = W_{k,S \cup \{i,l\}}$, thus $W_{k,S \cup \{i\}} = W_{k,S \cup \{i,l\}} - W_{k,S}$. Therefore, $\forall l \notin D_k$,

\[
\phi_i = \sum_{k,i \in D_k} \left[ \sum_{S \subset N \setminus \{i,l\}} \frac{|S|!{||N|-|S|-1}|!}{|N|!}(W_{k,S \cup \{i\}} - W_{k,S}) + \sum_{S \subset N \setminus \{i,l\}} \frac{|S|!{||N|-|S|-2}|!}{|N|!}(W_{k,S \cup \{i\}} - W_{k,S\cup\{l\}})\right]v_k
\]

\[
= \sum_{k,i \in D_k} \left[ \sum_{S \subset N \setminus \{i,l\}} \left( \frac{|S|!{||N|-|S|-1}|!}{|N|!} + \frac{|S|!{||N|-|S|-2}|!}{|N|!}\right)(W_{k,S \cup \{i\}} - W_{k,S})\right]v_k
\]

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

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

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

\[
\phi_i = \sum_{k,i \in D_k} \left[ \sum_{S \subset D_k \setminus \{i\}} \frac{|S|!{||D_k|-|S|-1}|!}{|D_k|!}\left( \prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{k_j}\} \prod_{j \in P_k, d_j \notin S} R_{k_j}\right)\right]v_k
\]

\[
= \sum_{k,i \in D_k} \left[ \sum_{m=0}^{D_k-1} \frac{m!{||D_k|-m-1}|!}{|D_k|!}\sum_{S \subset D_k \setminus \{i\}, |S|=m} \left( \prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{k_j}\} \prod_{j \in P_k, d_j \notin S} R_{k_j}\right)\right]
\]

\[
\cdot \left( \prod_{j \in P_k, d_j = i} \mathbb{1}\{x_{d_j} \in T_{k_j}\} - \prod_{j \in P_k, d_j = i} R_{k_j}\right)\right)v_k.
\]

\[\square\]

A.3 Proof of Theorem 2

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

\[
\phi_i = \sum_{k,i \in D_k} \left[ \sum_{m=0}^{D_k-1} \frac{m!{||D_k|-m-1}|!}{|D_k|!}\left( \sum_{S \subset D_k \setminus \{i\}, |S|=m} \prod_{j \in P_k, d_j \in S} \mathbb{1}\{x_{d_j} \in T_{k_j}\} \prod_{j \in P_k, d_j \notin S} R_{k_j}\right)\right]
\]

\[
\cdot \prod_{j \in P_k, d_j \in S_{ki}} \mathbb{1}\{x_{d_j} \in T_{k_j}\} \cdot \left( \prod_{j \in P_k, d_j = i} \mathbb{1}\{x_{d_j} \in T_{k_j}\} - \prod_{j \in P_k, d_j = i} R_{k_j}\right)\right)v_k.
\]
I.e., only subsets $S \subset D_k \setminus \{i\} \cup S_{ki}$ are necessary to be included in the equation.

Let $W_{ki} := \left[ \sum_{m=0}^{\frac{|D_k|-1-|S_k|}{|D_k|!}} m! \left( \frac{|D_k|-m-1)!}{|D_k|} \right) \left( \sum_{S \subset \{D_k\} \setminus \{i\} \cup S_{ki}} \prod_{j \in P_k, d_j \in S_k} R_{kj} \right) \right]$;

\[
\prod_{j \in P_k, d_j \in S_k} R_{kj} \cdot \left( \prod_{j \in P_k, d_j = i} \mathbb{1}\{x_{d_j} \in T_{kj}\} \right) - \prod_{j \in P_k, d_j = i} R_{kj},
\]

then $\phi_i = \sum_{k \in D_k} W_{ki} v_k$.

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 $U_{m,D_k,C} := \frac{m! |D_k|^{-m} - 1}{|D_k|!} \sum_{S \subset C, |S| = m} \prod_{j \in P_k, d_j \in C \setminus S} R_{kj}$, $m = 0, \ldots, |C|$, and $U_{D_k,C} := \sum_{m=0}^{C+1} |D_k|^{-m} U_{m,D_k,C}$, 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., $i \notin S_k$), we have $S_k \cup \{i\} = S_k$, and

\[
W_{ki} = \left[ \sum_{m=0}^{\frac{|D_k|-1-|S_k|}{|D_k|!}} m! \left( \frac{|D_k|-m-1)!}{|D_k|} \right) \left( \sum_{S \subset \{D_k\} \setminus \{i\} \cup S_{ki}} \prod_{j \in P_k, d_j \in D_k \setminus (S \cup i)} R_{kj} \right) \right]
\]

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 $S_k = S_k$, and

\[
W_{ki} = \left[ \sum_{m=0}^{\frac{|D_k|-1-|S_k|}{|D_k|!}} m! \left( \frac{|D_k|-m-1)!}{|D_k|} \right) \left( \sum_{S \subset \{D_k\} \setminus \{i\} \cup S_{ki}} \prod_{j \in P_k, d_j \in D_k \setminus (S \cup i)} R_{kj} \right) \right]
\]

\[
U_{D_k,C} := \sum_{m=0}^{C+1} |D_k|^{-m} U_{m,D_k,C}
\]

A.4 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\footnote{We use the same notation as in the original TreeSHAP algorithm for easy reference.} 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., $R_{kj}$), iii) $m_{j+1,o}$, the threshold condition for the $j$th 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 $j$th internal node, $w$ is used to record $U_{m,D_k,C,D_k \setminus (i \cup S_k)}$, where $D_k$ and $S_k$ are subsets of $D_k$ and $S_k$ up to the $j$th internal node, and $q$ is used to record $\prod_{j \in P_k, d_j \in S_k} R_{kj}$ 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
Algorithm 2 Fast TreeSHAP v1

```latex
\begin{algorithm}
\textbf{procedure} TREE\textsc{SHAP}(x, tree = \{v, a, b, t, r, d\})
  \begin{align*}
  &\phi = \text{array of len}(x) \text{ zeros} \\
  &\text{RECURSE}(0, [], [], 1, 1, 1, -1) \quad \triangleright \text{Start recursion at the root node} \\
  &\text{return } \phi
  \end{align*}
\textbf{end procedure}

\textbf{procedure} RECURSE(j, m, w, q, p_z, p_o, p_i)
  \begin{align*}
  &m, w, q = \text{EXTEND}(m, w, q, p_z, p_o, p_i) \quad \triangleright \text{Update } m, w, q \text{ for the feature of the last split} \\
  &\text{if } v_j \neq \text{internal} \text{ then} \quad \triangleright \text{Check if we are at a leaf node} \\
  &\quad \text{if } \text{len}(m) > \text{len}(w) \text{ then} \quad \triangleright \text{Pre-calculate } U_{D_k, D_k \setminus S_k} \text{ for } d_i \text{'s in } S_k \\
  &\quad \quad s_0 = -\text{sum}(\text{UNWIND}(m, w, q, -1)[1]) \\
  &\quad \text{end if} \\
  &\quad \text{for } i \leftarrow 1 \text{ to } \text{len}(m) - 1 \text{ do} \\
  &\quad \quad \text{if } m_i.o = 0 \text{ then} \quad \triangleright \text{Update contribution for feature } d_i \in S_k \\
  &\quad \quad \quad \phi_{m_i.d} = \phi_{m_i.d} + s_0 \cdot q \cdot v_j \\
  &\quad \quad \text{else} \quad \triangleright \text{Calculate } U_{D_k, D_k \setminus (\{d_i\} \cup S_k)} \text{ for } d_i \notin S_k \\
  &\quad \quad \quad s = \text{sum}(\text{UNWIND}(m, w, q, i)[1]) \\
  &\quad \quad \quad \phi_{m_i.d} = \phi_{m_i.d} + s \cdot q \cdot (1 - m_i.z) \cdot v_j \\
  &\quad \text{end if} \\
  &\quad \text{end for} \\
  &\quad \text{else} \\
  &\quad \quad h, c = (a_j, b_j) \text{ if } x_{d_i} \leq t_j \text{ else } (b_j, a_j) \\
  &\quad \quad i_z = i_o = 1 \\
  &\quad \quad k = \text{FINDFIRST}(m.d, d_j) \\
  &\quad \text{if } k \neq \text{nothing} \text{ then} \quad \triangleright \text{Undo previous update for feature } d_k \text{ if we see this feature again} \\
  &\quad \quad \quad i_z, i_o = m_k.z, m_k.o \\
  &\quad \quad \quad m, w, q = \text{UNWIND}(m, w, q, k) \\
  &\quad \text{end if} \\
  &\quad \text{RECURSE}(h, m, w, q, i_z r_h/r_j, i_o, d_j) \\
  &\text{RECURSE}(c, m, w, q, i_z r_c/r_j, 0, d_j) \\
  &\text{end if}
\end{algorithm}
```

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

\textbf{procedure} UNWIND(m, w, q, i)
  \begin{align*}
  &l, l_w = \text{len}(m) - 1, \text{len}(w) - 1 \\
  &m = \text{copy}(m_{a_i \cdots i - 1}) \quad \triangleright \text{Undo previous update for feature } d_i \\
  &\text{if } i < 0 \text{ then} \\
  &\quad w = \text{copy}(w) \\
  &\quad \text{for } j \leftarrow l_w \text{ to } 0 \text{ do}
\end{algorithm}
```

```
```
```
```
```
\[ w_j = w_j \cdot (l+1)/(l-j) \]

end for

else
if \( m_i.o = 0 \) then
\[ w = \text{copy}(w) \]
for \( j \leftarrow l_w \) to 0 do
\[ w_j = w_j \cdot (l+1)/(l-j) \]
end for
\[ q = q/m_i.z \]
else
\[ n = w_{l_w} \]
\[ w = \text{copy}(w_{0..l_w-1}) \quad \n \text{Shrink } w \text{ if feature } d_i \notin S_k \]
for \( j \leftarrow l_w - 1 \) to 0 do
\[ t = w_j \]
\[ w_j = n \cdot (l+1)/(j+1) \]
\[ n = t - w_j \cdot m_i.z \cdot (l-j)/(l+1) \]
end for
end if
for \( j \leftarrow i \) to \( l - 1 \) do
\[ m_j.(d, z, o) = m_{j+1}.(d, z, o) \quad \n \text{Shrink } m \text{ given feature } d_i \]
end for
end if
return \( m, w, q \)
end procedure

A.5 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. \( W \) 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 \( j \)th internal node along path \( P_k \), each row of \( W \) records \( \{U_{m,D_k,C} : m = 0, \cdots, |C|\} \) for a subset \( C \subset D_{kj} \) 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

\begin{algorithm}
\begin{algorithmic}
\Procedure{TreeSHAPPrep}{tree = \{v, a, b, t, r, d\}}
\State \( S \) = matrix of \( L \times 2^D \) zeros
\State \( e \) = array of \( \text{len}(d) \) negative ones
\State \( W \) = matrix of \( 2^D \times (D + 1) \) zeros
\State \( l_w \) = array of \( 2^D \) zeros
\State \( c = 0 \)
\State \text{RECURSE}(0, [], W, l_w, 1, -1) \Comment{Start recursion at the root node}
\Return \( S, e \)
\EndProcedure
\Procedure{RECURSE}{j, m, W, l_w, p_z, p_l}
\State \( m, W, l_w \) = \text{EXTEND}(m, W, l_w, p_z, p_l) \Comment{Update \( m, W, l_w \) for the feature of the last split}
\If{\( v_j \neq \text{internal} \)} \Comment{Check if we are at a leaf node}
\State \( l = \text{len}(m) - 1 \)
\For{\( t \leftarrow 0 \) to \( 2^l - 1 \)} \Comment{Calculate \( \{U_{D_k,C} : C \subset D_k\} \) for the current path}
\For{\( i \leftarrow l_w[t] \) to 0 do}
\State \( S[c, t] = S[c, t] + W[t, i] \cdot (l + 1)/(l - i) \)
\EndFor
\EndFor
\EndIf
\EndProcedure
\end{algorithmic}
\end{algorithm}

\footnote{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\}\}.}
procedure UNWIND(m, W, lw)

\[ m, W, lw = \text{UNWIND}(m, W, lw, k) \]  
\[ \triangleright \text{Undo previous update for } d_k \text{ if we see it again} \]

end procedure

procedure EXTEND(m, W, lw, p_z, p_i)

\[ l = \text{len}(m) \]
\[ m = \text{copy}(m) \]
\[ m_j, (d, z) = (p_i, p_z) \]

if \( l = 0 \) then
\[ W[0, 0] = 1 \]
else
\[ W, lw = \text{copy}(W, copy(lw)) \]
\[ W[2^{l-1} : 2^l, \cdot] = \text{copy}(W[2^{l-1} : 2^l, \cdot]) \]
\[ lw[2^{l-1} : 2^l] = \text{copy}(lw[2^{l-1}]) \]

for \( t \leftarrow 0 \) to \( 2^{l-1} - 1 \) do
\[ W[t, i] = W[t, i] \cdot (l-i)/(l+1) \]
end for

end if

return \( m, W, lw \)

end procedure

procedure UNWIND(m, W, lw, i)

\[ l = \text{len}(m) - 1 \]
\[ m, W, lw = \text{copy}(m_0, \ldots, m_{l-1}), \text{copy}(W), \text{copy}(lw) \]
\[ \text{ind} = \text{empty array} \]

for \( t_1 \leftarrow \{0, 2, 2 \cdot 2^2, 3 \cdot 2^2, \ldots, 2^l - 2^l \} \) do
\[ \text{ind.append}(t_2) \]
end for

for \( t \leftarrow 0 \) to \( 2^{l-1} - 1 \) do
\[ W[t, i] = W[t, i] \cdot (l+1)/(l-i) \]
end for

end procedure

procedure UNWIND(m, W, lw, k)

\[ i_z = 1 \]
\[ k = \text{FINDFIRST}(m.d, d_j) \]
if \( k \neq \text{nothing} \) then
\[ e_j = k \]
\[ i_z = m_k.z \]
\[ m, W, lw = \text{UNWIND}(m, W, lw, k) \]  
\[ \triangleright \text{Record node index of duplicated feature} \]

end if

\[ lw = lw + 1 \]
RECURSE(a_j, m, W, lw, i_z a_j/r_j, d_j)
RECURSE(b_j, m, W, lw, i_z b_j/r_j, d_j)

end if

end procedure

\[ c = c + 1 \]
Algorithm 4 Fast TreeSHAP v2: Score

```latex
\begin{algorithm}
\begin{algorithmic}
\Procedure{TREESHAPSCORE}{x, tree = \{v, a, b, t, r, d, S, e\}}
\State $\phi = \text{array of } \text{len}(x) \text{ zeros}$
\State $b = \text{array of } [0, 1, 2, 2^2, \ldots, 2^D]$ \Comment{Start recursion at the root node}
\State $c = 0$
\State $\text{RECURSE}(0, [], 1, 1, 1, -1)$ \Comment{Start recursion at the root node}
\Return $\phi$
\EndProcedure
\Procedure{RECURSE}{j, m, q, p, z, p_o, p_i}
\State $l = \text{len}(m)$
\State $m = \text{copy}(m)$
\State $m_{j} (d, z, o) = (p_i, p_z, p_o)$ \Comment{Update $m, q$ for the feature of the last split}
\If {$p_o = 0$}
\State $q = q \cdot p_z$
\EndIf
\If {$v_j \neq \text{internal}$} \Comment{Check if we are at a leaf node}
\State $b_{\text{sum}} = 0$
\For {$i \leftarrow 1 \to l$} \Comment{Use $b_{\text{sum}}$ to search in $\{U_{D_k, C} : C \subset D_k\}$ for the current path}
\State $b_{\text{sum}} = b_{\text{sum}} + b_i$ if $m_{j}.o \neq 0$
\EndFor
\For {$i \leftarrow 1 \to l$} \Comment{Update contribution for feature $d_i \in S_k$}
\If {$m_{j}.o = 0$}
\State $\phi_{m_{j}.d} = \phi_{m_{j}.d} - S[e, b_{\text{sum}}] \cdot q \cdot v_j$
\Else
\State $\phi_{m_{j}.d} = \phi_{m_{j}.d} + S[e, b_{\text{sum}} - b_i] \cdot q \cdot (1 - m_{j}.z) \cdot v_j$
\EndIf
\EndFor
\State $c = c + 1$
\Else
\State $i_z = i_o = 1$
\State $k = e_j$
\If {$k \neq -1$} \Comment{Undo previous update for feature $d_k$ if we see this feature again}
\State $i_z, i_o = m_k.z, m_k.o$
\State $m = \text{copy}(m_{0..l-1})$
\For {$j \leftarrow k \to l - 1$}
\State $m_{j+1} (d, z, o) = m_{j+1} (d, z, o)$
\EndFor
\If {$i_o = 0$} \Comment{update contribution for feature $d_i \notin S_k$}
\State $q = q / i_z$
\EndIf
\EndIf
\EndIf
\EndProcedure
\EndAlgorithm
```

A.6 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.

| 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  |

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

A.7 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.8 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.

| Model       | # Trees | Max Depth | # Leaves |
|-------------|---------|-----------|----------|
| Adult-Small | 100     | 4         | 1,480    |
| Adult-Med   | 100     | 8         | 7,375    |
| Adult-Large | 100     | 12        | 21,085   |
| Adult-xLarge| 100     | 16        | 36,287   |
| Super-Small | 100     | 4         | 1,512    |
| Super-Med   | 100     | 8         | 12,802   |
| Super-Large | 100     | 12        | 47,840   |
| Super-xLarge| 100     | 16        | 108,010  |

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

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

A.9 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., \( \text{#thread} \cdot O(L^2D) \)) 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.10 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.9 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
Table 8: 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 | 2.03 (0.05)          | <0.01 (<0.01)          | 0.85 (0.01)             | 2.39                   | 2KB                         |
| Adult-Med  | 23.61 (0.11)         | 0.05 (<0.01)           | 7.60 (0.07)             | 3.11                   | 258KB                       |
| Adult-Large | 113.77 (0.44)        | 1.03 (0.02)            | 35.95 (0.19)            | 3.16                   | 14.2MB                      |
| Adult-xLarge | 256.32 (1.24)       | 6.76 (0.08)            | 71.26 (0.33)            | 3.60                   | 421MB                       |
| Super-Small | 2.61 (0.10)          | <0.01 (<0.01)          | 1.12 (0.06)             | 2.33                   | 2KB                         |
| Super-Med  | 47.00 (1.64)         | 0.15 (0.01)            | 17.30 (0.12)            | 2.72                   | 392KB                       |
| Super-Large | 334.28 (12.46)      | 7.42 (0.06)            | 98.43 (2.71)            | 3.40                   | 27.7MB                      |
| Super-xLarge | 1124.07 (4.50)     | 147.00 (3.87)          | 298.49 (5.03)           | 3.77                   | 936MB                       |

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 \(400 \times 30MB = 12GB\) 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 4 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 [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 \(315 \times 32 \times 5/4s = 3.5h\) when parallelizing on the entire 4 cores. Fast TreeSHAP v1 can help reduce the execution time to \(216 \times 32 \times 5/4s = 2.4h\). Since this is the one-time usage scenario, per-tree parallelization can be implemented for Fast TreeSHAP v2, which costs around \(15 \times 4MB = 60MB\) in memory, and can further reduce the execution time to \(130 \times 32 \times 5/4s = 1.4h\).