WOODELF-HD: EFFICIENT BACKGROUND SHAP FOR HIGH-DEPTH DECISION TREES

Anonymous authors
Paper under double-blind review

ABSTRACT

Decision-tree ensembles are a cornerstone of predictive modeling, and *SHAP* is a standard framework for interpreting their predictions. Among its variants, *Background SHAP* offers high accuracy by modeling missing features using a background dataset. Historically, this approach did not scale well, as the time complexity for explaining n instances using m background samples included an $O(mn)$ component. Recent methods such as WOODELF and PLTREESHAP reduce this to $O(m + n)$, but introduce a preprocessing bottleneck that grows as 3^D with tree depth D, rendering them impractical for deep trees. We address this limitation with WOODELFHD, an extension of WOODELF that reduces the 3^D factor to 2^D. The key idea is a Strassen-like multiplication scheme that exploits the structure of WOODELF matrices, reducing matrix–vector multiplication from $O(k^2)$ to $O(k \log k)$ through a fully vectorized, non-recursive implementation. In addition, we merge path nodes with identical features, further reducing cache size and memory usage. When run in standard environments, WOODELFHD enables exact Background SHAP computation for tree models with depths up to 21, whereas previous methods fail due to excessive memory usage. For ensembles with depths of 12 and 15, it achieves speedups of $33\times$ and $162\times$, respectively, over the state of the art.

1 INTRODUCTION

Decision tree ensembles such as XGBoost (Chen & Guestrin, 2016), Random Forest (Breiman, 2001), and CatBoost (Dorogush et al., 2017) are widely used for classification and regression. As their adoption has grown, so has interest in explaining their predictions (Gill et al., 2020; Molnar, 2022). SHAP (Lundberg & Lee, 2017) provides a unified and consistent (Lundberg et al., 2019) framework by modeling explanations as a cooperative game and assigning feature attributions via Shapley values (Shapley, 1953). It further extends to Shapley interaction values, capturing pairwise feature interactions (Grabisch & Roubens, 1999; Fumagalli et al., 2023).

SHAP has two commonly used variants. *Background SHAP*, also known as *interventional SHAP*, uses a background dataset to accurately capture the model prediction when some features are missing. Concretely, each background row yields one hybrid instance, in which the features in the coalition keep the values of the instance being explained while the remaining features take the values of that background row. The value of the coalition is then the model's average prediction over these hybrid instances. The background dataset therefore supplies the distribution over which the missing features are marginalized, so the attribution reflects the data rather than the structure of the trees.

Path-Dependent SHAP estimates missing behavior based on node-coverage statistics. It can be inconsistent, producing different explanations for functionally equivalent trees (Liu et al., 2025; Filom et al., 2024). While our proposed solution applies to both variants, we focus primarily on Background SHAP, the more accurate approach, where we improve the state-of-the-art complexity.

The most widely used implementation of SHAP is the SHAP Python package (Lundberg et al., 2020), which supports both CPU (Lundberg et al., 2020) and GPU (Mitchell et al., 2022) computation. SHAP's key drawback is its quadratic complexity for Background SHAP: the complexity of explaining the predictions of n instances with a background dataset of size m includes an $O(mn)$ component. The PLTREESHAP algorithm (Zern et al., 2023) reduces this to $O(m + n)$ by preprocessing the decision tree and the background dataset.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Time complexities for Background SHAP (BG SHAP) and Background SHAP interaction values (BG SHAP IV). The table presents the complexities of the SHAP Python package, PLTreeSHAP, original WOODELF (as presented in (Nadel & Wettenstein, 2026)) and of WOODELFHD (as presented in this paper). Appendix D presents a more general formulation of WOODELFHD complexity. Notation: f: no. of features; n: no. of consumer data rows (number of samples to explain); m: no. of background data rows; T : no. of trees; D: max depth; L: max leaves per tree.

Task	SHAP Package	PLTreeSHAP	Original WOODELF	WOODELFHD
BG SHAP	$O(nmTLD)$	$O(mTL+ nT3^D D)$	$O(mTL+ nTLD + TL3^D D)$	$O(mTL+ nTLD + TL2^D D^2)$
BG SHAP IV	$O(nmTLDf)$	$O(mTL+ nT3^D D^2)$	$O(mTL+ nTLD^2 + TL3^D D^2)$	$O(mTL+ nTLD^2 + TL2^D D^3)$

Table 2: Space complexities of WOODELF (Nadel & Wettenstein, 2026) and WOODELFHD.

Task	Original WOODELF	WOODELFHD
PD SHAP	$O(L3^D D + n(f + L))$	$O(2^D D + n(f + D))$
BG SHAP	$O(L3^D D + n(f + L) + mL)$	$O(2^D D + n(f + D) + mD)$
PD SHAP IV	$O(L3^D D^2 + n(f^2 + L))$	$O(2^D D^2 + n(f^2 + D))$
BG SHAP IV	$O(L3^D D^2 + n(f^2 + L) + mL)$	$O(2^D D^2 + n(f^2 + D) + mD)$

Recent work introduced WOODELF, a GPU-friendly algorithm for both Background and Path-Dependent SHAP (Nadel & Wettenstein, 2026). In addition to SHAP, WOODELF supports Banzhaf values and Partial Dependence Plots (Wettenstein et al., 2026). Like PLTREESHAP, WOODELF computes Background SHAP with a complexity that scales as $O(m + n)$. It is significantly faster than SHAP, PLTREESHAP, and FASTTREESHAP, especially on large datasets. In this paper, we extend WOODELF to efficiently support high-depth trees.

Our Contributions A key limitation of WOODELF and PLTREESHAP is their poor scalability to deep trees, with time and space complexities containing a $(3^D D)$ factor, where D is the tree depth, making them impractical beyond moderate depths. Such depths are common in practice: methods like LIGHTGBM (Ke et al., 2017) employ a leaf-wise (depth-first) growth strategy that often yields deep trees, and even in XGBoost, which grows trees level-wise (breadth-first), empirical studies report optimal depths around 13 (Probst et al., 2019).

We introduce WOODELFHD, which improves the state-of-the-art complexity for Background SHAP by reducing the $3^D D$ factor to $2^D D^2$ in time and $2^D D$ in space (see Tables 1 and 2). These gains also translate to Path-Dependent SHAP, where in practice our method often outperforms the SHAP package on large datasets.

Empirically, WOODELFHD significantly extends the range of tractable depths, enabling efficient computation up to depth 21 on a 50GB machine, while prior methods fail beyond depth 15, and achieving up to 30× speedups at moderate depths (more details in Sect. 5).

Key Ideas and Optimizations The complexity improvements summarized in Tables 1 and 2 are achieved through several key ideas:

1. We run the entire WOODELF pipeline on each leaf independently, rather than on each tree. This change significantly reduces space complexity.
2. We merge repeated feature splits along each root-to-leaf path, enabling a compact cache that speeds up preprocessing.
3. A major bottleneck in WOODELF is computing the product of a $2^D \times 2^D$ matrix M and a 2^D vector f, which naively costs $O(4^D)$. Leveraging the structure of M, induced by the

linearity of Shapley values, we propose a Strassen-like multiplication scheme that reduces the complexity to $O(2^D D)$.

4. We further observe that this scheme depends only on the secondary diagonal of M, which we compute and store explicitly. Combined with the Strassen-like scheme, this reduces the multiplication time and space from $O(3^D)$ to $O(2^D D)$.
5. A naive implementation of the Strassen-like scheme requires $O(2^D)$ recursive calls. Instead, we derive a fully vectorized version that uses only $O(D)$ operations on 2^D-sized vectors, eliminating recursion and greatly improving performance.

Overall, our approach is vectorized using Numpy (Harris et al., 2020), utilizes SIMD (Single-Instruction Multiple Data (Hughes, 2015)), is GPU-friendly, and is implemented in pure Python.

2 THE WOODELF ALGORITHM

This section briefly reviews the WOODELF algorithm (Nadel & Wettenstein, 2026), upon which our method builds. We begin with a short description of decision trees and root-to-leaf paths.

A decision tree is a rooted binary tree in which each node is either an internal node or a leaf. Each leaf is associated with a weight. An internal node n, with left child $n.l$ and right child $n.r$, specifies a split $n.\textit{split}$ of the form $n.\textit{feature} < n.\textit{threshold}$: if the split holds, traversal proceeds to $n.l$, and otherwise to $n.r$. A *root-to-leaf path* for a leaf l is the unique path from the root to l.

The first observation underlying WOODELF is that SHAP computation at a given leaf depends only on the splits along its root-to-leaf path. For each consumer row (i.e., a sample being explained) and background row, only the split outcomes along this path influence the resulting Shapley values. WOODELF encodes these outcomes as a *decision pattern* (see Def. 1).

Definition 1 (Decision Pattern (Nadel & Wettenstein, 2026, Definition 7))**.** *Given a decision tree T, a leaf l, their root-to-leaf path $(n_1 \equiv r_T, n_2, \ldots, n_{D-1}, l)$, and feature values $c = (c_1, \ldots, c_f) \in \mathbb{R}^f$, the* decision pattern *p is a binary sequence of length $D-1$. Its i-th bit is defined as:*

$$p[i] = \begin{cases} 1 & \textit{if } (n_i.\textit{split}(c) = \textit{True}) \wedge (n_i.\textit{left} = n_{i+1}), \\ 1 & \textit{if } (n_i.\textit{split}(c) = \textit{False}) \wedge (n_i.\textit{right} = n_{i+1}), \\ 0 & \textit{otherwise}. \end{cases}$$

Decision patterns are computed efficiently using *CalcDecisionPatterns* (Alg. 5 in Appendix C.2).

A *literal* is a Boolean variable x_i or its negation $\neg x_i$, and a *cube* is a conjunction of literals. For example, $(x_2 \wedge x_3 \wedge \neg x_5)$ is a cube. The WOODELF algorithm observes that the characteristic function induced by a single leaf, consumer, and background row can be represented as a weighted cube (see Fig. 1 for an example). In this representation, each player corresponds to a Boolean

Algorithm 1 Map Patterns to Cube (Nadel & Wettenstein, 2026, Alg. 2)
Algorithm 1' Cubes In Diagonal (without line 11)

```
1:  function MapPatternsToCube(features)
2:      d ← {0 ↦ {0 ↦ (∅, ∅)}}
3:      for f ∈ features do
4:          d_old ← d
5:          d ← {}
6:          for p_c in d_old do
7:              for p_b in d_old[p_c] do
8:                  (S⁺, S⁻) ← d_old[p_c][p_b]
9:                  d[2p_c+1][2p_b+0] ← (S⁺ ∪ {f}, S⁻)
10:                 d[2p_c+0][2p_b+1] ← (S⁺, S⁻ ∪ {f})
11:                 d[2p_c+1][2p_b+1] ← (S⁺, S⁻)
12:     return d
```

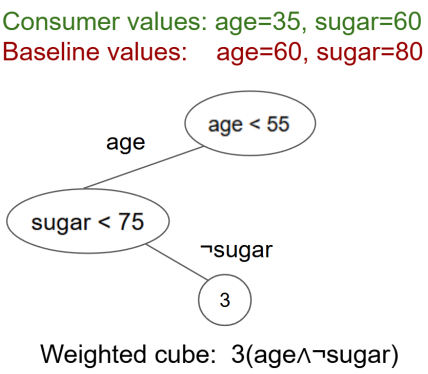


Figure 1: A consumer and baseline row along a root-to-leaf path, showing their decision patterns and the matching weighted cube (constructed as in Alg. 1).

p_c \ p_b	00	01	10	11
00	$\emptyset$	$\emptyset$	$\emptyset$	$(\neg age \wedge \neg sugar)$
01	$\emptyset$	$\emptyset$	$(\neg age \wedge sugar)$	$(\neg age)$
10	$\emptyset$	$(age \wedge \neg sugar)$	$\emptyset$	$(\neg sugar)$
11	$(age \wedge sugar)$	(age)	$(sugar)$	$()$

Figure 2: The matrix produced by *MapPatternsToCube* for the root-to-leaf path shown in Fig. 1. The entry at row index 10 and column index 01 is the cube that matches the consumer and baseline rows in Fig. 1. Applying any function that satisfies linearity, such as the Shapley value of $sugar$ or age, to the cubes in this table yields an M matrix. These kinds of matrices are discussed in Sect. 3.

variable, and the value of any coalition of players is obtained by evaluating the weighted cube with participating players set to True and all others set to False.

Given the features along a root-to-leaf path, the procedure *MapPatternsToCube* (Alg. 1) constructs a matrix that maps each pair of consumer and background decision patterns (row and column indices, respectively) to the cube they induce at that leaf. For a path with D unique features, this matrix has 2^D rows, 2^D columns, and 3^D non-empty entries. See an example of such a matrix in Fig. 2.

Nadel & Wettenstein (2026) further provides a simple formula for computing Shapley values for games represented by weighted cubes. For each feature along the root-to-leaf path, WOODELF applies its Shapley-values formula to all cubes in the matrix constructed by *MapPatternsToCube*. The resulting matrix for feature i maps each pair of consumer and background decision patterns to their corresponding Shapley value for feature i. We refer to these matrices as the M *matrices*. They play a central role in this work: we analyze their structure to accelerate matrix–vector multiplication.

At a high level, the WOODELF algorithm operates on a single tree and consists of four main steps:

1. **Computing the f vectors**: The algorithm finds the background decision patterns for all leaves, computes their *value_counts*, and stores the results as the f vectors.
2. **Computing the M matrices**: For each leaf, WOODELF precomputes the Shapley values for all possible pairs of consumer and background decision patterns. This is done by generating matrices using *MapPatternsToCube* and applying the Shapley value function to their cubes. An alternative definition of the M matrices is provided in Appendix A. The appendix defines them purely in pseudo-Boolean terms, offering a more general perspective that may extend beyond decision tree ensembles.
3. **Computing the s vectors** using sparse matrix-vector multiplication, $s = M_i \cdot f$. For each leaf and each feature i on its root-to-leaf path, the resulting s vector maps every possible consumer pattern to the Shapley value assigned to feature i at that leaf.
4. **Computing final Shapley values using the s vectors**: For each leaf, using the consumer's decision pattern with the leaf's s vectors to get its contribution, then summing over leaves.

3 STRASSEN-LIKE MULTIPLICATION

This section presents an efficient approach for computing the product $M \cdot f$ described in Sect. 2.

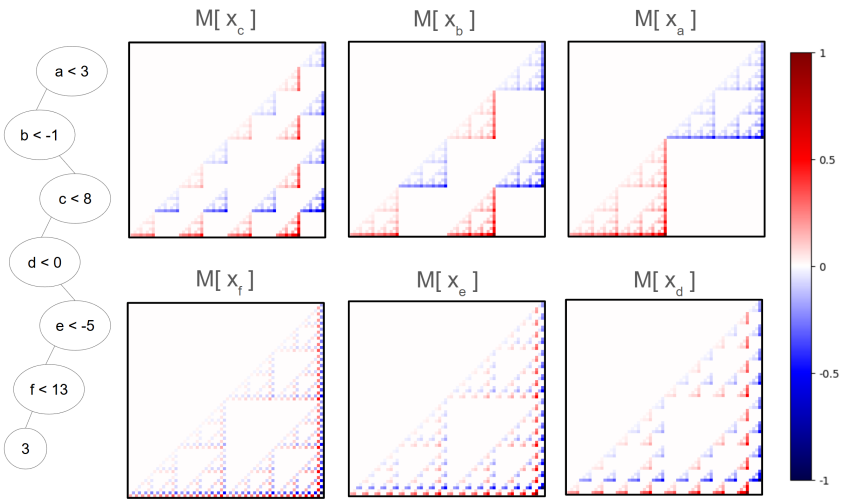


Figure 3: The M matrices corresponding to a root-to-leaf path with 6 unique features. The matrix $M[x_a]$ matches feature a from the first split, $M[x_b]$ matches feature b, and so on. Each cell is colored according to its value: 1 appears as dark red, -1 as dark blue, 0 as white, and intermediate values are shown with opacity proportional to their magnitude.

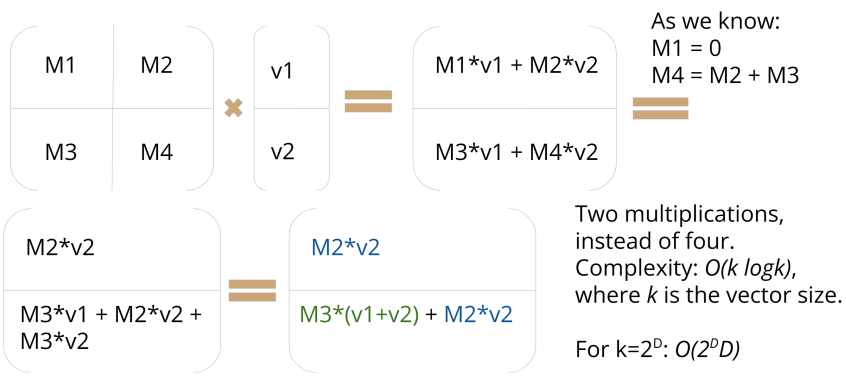


Figure 4: Using the matrix properties to perform matrix–vector multiplication with only 2 inner multiplications instead of 4. This Strassen-like formulation reduces the complexity from $O(k^2)$ to $O(k \cdot \log k)$, i.e., from $O(4^D)$ to $O(2^D D)$.

Let us start by building some intuition about the structure of the M matrices. A simple way to explore this structure is to plot the matrices themselves. Fig. 3 shows examples of M matrices generated by a single root-to-leaf path. These matrices exhibit a visually striking and highly structured pattern with numerous symmetries. For example, all non-zero elements (the blue and red cells) are a subset of a Sierpinski triangle (Sierpiński, 1915; Carli et al., 2025) (see the proof of Lemma 1).

The Strassen matrix multiplication algorithm recursively partitions each matrix into four equally sized quadrants at every recursion step. Let M be a $2^D \times 2^D$ matrix generated by the WOODELF algorithm, as discussed in Sect. 2. Consider a sub-matrix M_{sub} encountered at recursion depth i ($1 \leq i \leq D$).

Let us further partition M_{sub} into quarters (where r and c are common $(i-1)$-bit prefixes):

1. $M_1 = M_{\text{sub}}[c\mathbf{0}\ldots][r\mathbf{0}\ldots]$
2. $M_2 = M_{\text{sub}}[c\mathbf{1}\ldots][r\mathbf{0}\ldots]$
3. $M_3 = M_{\text{sub}}[c\mathbf{0}\ldots][r\mathbf{1}\ldots]$
4. $M_4 = M_{\text{sub}}[c\mathbf{1}\ldots][r\mathbf{1}\ldots]$

These matrices have useful mathematical properties:

Lemma 1. *The matrices M_1, M_2, M_3, M_4 satisfy: $M_1 = 0$ and $M_4 = M_2 + M_3$.*

Proof. $M_1 = 0$**:** By construction, all row and column indices of M_1 have a 0 at position i. In this case, ***MapPatternsToCube*** does not generate any cube (the $d[2p_c + 0][2p_b + 0]$ case), hence all entries are zero and $M_1 = 0$. This recursive property, which applies to the upper-left quadrant, is what gives rise to the Sierpinski triangle structure shown in Fig. 3 (the $d[2p_c + 0][2p_b + 0]$ case is a structure analogous and symmetric to (Devaney, 1991, Exercise 1.2.4)).

$M_4 = M_2 + M_3$**:** Let M_2', M_3', and M_4' be the matrices produced by ***MapPatternsToCube*** before applying the linear function v (e.g., v can be the Shapley value function for feature f_k), so that

$$M_2 = v(M_2'), \qquad M_3 = v(M_3'), \qquad M_4 = v(M_4').$$

For any indices $0 \leq g, h \leq 2^{D-i} - 1$, the cubes at $M_2'[g][h]$, $M_3'[g][h]$, and $M_4'[g][h]$ differ only in the literal of x_i. Specifically, from lines 9–11 of Alg. 1, we have:

$$x_i \in M_2'[g][h], \qquad \neg x_i \in M_3'[g][h], \qquad x_i, \neg x_i \notin M_4'[g][h].$$

Hence,

$$M_4'[g][h] = M_2'[g][h] + M_3'[g][h].$$

For example $(x_j \wedge x_k) = (x_i \wedge x_j \wedge x_k) + (\neg x_i \wedge x_j \wedge x_k)$. Applying the linearity of v yields

$$v(M_4'[g][h]) = v(M_2'[g][h]) + v(M_3'[g][h]),$$

and therefore $M_4 = M_2 + M_3$. □

These properties apply to M matrices constructed using probabilistic values (Weber, 1988) (including the Shapley and the Banzhaf values (Banzhaf, 1965)), cardinal interaction indices (Grabisch & Roubens, 1999) (including Shapley interaction values of any order), and any other linear function.

We use these properties to propose a Strassen-like matrix-vector multiplication that requires only two inner multiplications instead of four. This reduces the computational complexity to $O(2^D D)$. The approach is straightforward, see Fig. 4 for details.

A key observation is that the two inner multiplications depend only on M_2 and M_3, and not on M_1 or M_4. Propagating this through the recursion shows that the computation depends solely on the secondary diagonal of M (top-right to bottom-left). We therefore compute and store only this diagonal, reducing the storage per matrix from $O(3^D)$ (the number of nonzero entries) to $O(2^D)$.

When constructing the secondary diagonal, we ignore all cases in which the row and column indices share the same bit value in any position. The ***MapPatternsToCube*** algorithm (Alg. 1) handles these cases at line 11 (the $d[2p_c{+}1][2p_b{+}1]$ case). Omitting this line yields the ***CubesInDiagonal*** algorithm (Alg. 1'), which computes only the cubes on the secondary diagonal.

Another improvement of our work is a fully vectorized implementation of the proposed multiplication scheme. The complete Python code is shown in Fig. 5, which also serves as a formal definition of the ***StrassenLikeMult*** algorithm. Our implementation avoids recursion entirely and requires only $O(D)$ vectorized operations. Intuition and further details are provided in Appendix B.

4 Merging Same-Feature Splits

In this section, we introduce a modified definition of decision patterns in which all nodes that split on the same feature are merged into a single constraint. This idea is inspired by a similar node-merging technique used in GPUTreeSHAP ((Mitchell et al., 2022, Sect. 3.2)), but here it is incorporated directly into the formal definition of decision patterns.

Using this formulation, we derive an algorithm for computing decision patterns that significantly reduces the space complexity compared to WOODELF. For Background SHAP, for example, it reduces the space complexity from $O(L3^D D + n(f + L) + mL)$ to $O(3^D D + n(f + D) + mD)$, see the notations in Table 1. Storing only the secondary diagonal of the M matrix, as discussed in Sect. 3, further reduces the 3^D component to 2^D.

```python
def StrassenLikeMult(M_diag, f, D):
    idx = np.arange(len(f))
    for d in range(D-1, -1, -1):
        f_copy = f.copy()
        # shift the array to the left 2**d bits
        f_copy[:-2**d] = f[2**d:]
        # zero all elements whose d-th bit in the index is 1
        f_copy[idx & (1 << d) != 0] = 0
        f = f + f_copy
    f_reverse = f[::-1]
    s = M_diag * f_reverse
    for d in range(D):
        s_copy = s.copy()
        s_copy[2**d:] = s[:-2**d]

        s_copy[idx & (1 << d) == 0] = 0
        s = s + s_copy
    return s
```

Figure 5: Python implementation of *StrassenLikeMult*. Here, M_{diag} is a NumPy array of length 2^D containing the secondary diagonal of M, f is a NumPy array of length 2^D, and D is the tree depth. The function computes $M \cdot f$.

An additional advantage of this representation is that it enables a simple and compact caching mechanism that accelerates preprocessing. The mathematical justification for this approach is illustrated in Fig. 9 in Appendix C.

The Unique Features Decision Pattern (UFDP) for a leaf l encodes the decisions along the root-to-leaf path using one bit per unique feature, instead of one bit per node. A bit is set to 1 if all splits on that feature follow the path to l, and to 0 if any split deviates from it.

Definition 2 (Unique Features Decision Pattern (UFDP))**.** *Let p be a decision pattern on a root-to-leaf path $(n_1 \equiv r_T, \ldots, n_{D-1}, l)$. Let k be the number of unique features appearing on this path, and let $f_1, \ldots, f_k$ denote these features in the order of their first appearance (e.g., f_1 is the feature used at the root). For each f_i, let $n_{i_1}, \ldots, n_{i_{h_i}}$ be the nodes on the path that split on feature f_i.*

The UFDP of the path is a binary sequence $u \in \{0,1\}^k$, such that its i-th bit is obtained from the decision pattern p by:

$$u[i] = \bigwedge_{\forall i_t \in \{i_1, \ldots, i_{h_i}\}} p[i_t].$$

To implement WOODELF with UFDPs, we modify the computation of the f vectors while preserving the final algorithm output and correctness. For Background SHAP, the f vector is obtained by applying *value_counts* to the UFDPs, which returns the frequency of each distinct value. For Path-Dependent SHAP, the $f_{l_{pd}}$ vector of leaf l is computed by mapping each possible UFDP u_b with a value, as defined below using the notation of Def. 2.

$$\mathbf{f}_{\mathbf{l_{pd}}}[u_b] = \prod_{i=1}^{k} \begin{cases} \prod_{t=1}^{h_i} \frac{n_{i_t+1}.\text{cover}}{n_{i_t}.\text{cover}} & \text{if } u_b[i] = 1, \\ 1 - \prod_{t=1}^{h_i} \frac{n_{i_t+1}.\text{cover}}{n_{i_t}.\text{cover}} & \text{if } u_b[i] = 0. \end{cases} \tag{1}$$

The *DecisionPatternsGenerator* algorithm efficiently computes UFDPs and replaces the *CalcDecisionPatterns* algorithm used in WOODELF. The idea is straightforward: traverse the path from the root to the leaf, and at each node examine its splitting feature. Two cases may arise:

1. If this feature has not been encountered before, append a new bit to the pattern with the current split result.
2. If the feature has already been encountered, locate its existing bit in the pattern and apply a logical AND between that bit and the current split result.

An additional advantage of the new approach is its improved space complexity. The algorithm is implemented as a generator that produces patterns leaf by leaf. This allows the system to limit storage to $O(D)$ nodes by immediately discarding the patterns of nodes whose children have all been processed. This represents a substantial improvement over *CalcDecisionPatterns*, which stores the patterns of all nodes in the tree. Let K denote the number of nodes in the tree (for a balanced tree, $K = L$). With n decision patterns per node, the space complexity is reduced from $O(nK)$ to $O(nD)$. The pseudocode of *DecisionPatternsGenerator* is provided in Appendix C.2.

A Cache Mechanism: Using UFDPs, the input to *CubesInDiagonal* is a list of unique features. For any semivalue (Dubey et al., 1981) or cardinal interaction index (Grabisch & Roubens, 1999) (including Shapley values and interactions), the computation is invariant to feature identities: any root-to-leaf path with k unique features yields the same dictionary d_l and the same k matrices. Hence, for a tree of maximum depth D, only D distinct cases arise. The *ComputeMs* algorithm (Appendix C.1) exploits this by precomputing all cases and caching them.

The WOODELFHD Algorithm: To conclude, the high-level steps of WOODELFHD are:

1. Precompute the secondary diagonals of all required matrices via *ComputeMs*.
2. For each leaf:
   (a) Compute the leaf's UFDPs using *DecisionPatternsGenerator*, storing up to $2D$ vectors during the traversal.
   (b) Compute the f vector from the UFDPs: use *value_counts* for Background SHAP, or Formula 1 for Path-Dependent SHAP.
   (c) Fetch the relevant M matrix from the cache.
   (d) Compute $s = f \cdot M$ efficiently using *StrassenLikeMult*.
   (e) Compute the leaf's Shapley contribution from s and the consumer's UFDP, and add it to the final Shapley values.

The full pseudocode is given in Appendix C.3.

5 EXPERIMENTAL RESULTS

We implemented the proposed WOODELFHD approach. All code and experimental notebooks are provided in the supplementary material.

We compared WOODELFHD with WOODELF, `PLTreeSHAP`, and the SHAP package on the IEEE-CIS fraud detection dataset[1], a widely used benchmark (Jiang et al., 2023; Xiao, 2024; Chen et al., 2021; B et al., 2024; Deng et al., 2021). After one-hot encoding, the dataset contains 397 features, with 472 432 training samples used as the background dataset and 118 108 test samples used as consumers. Across IEEE-CIS and the three additional datasets described in Appendix E, WOODELFHD consistently achieved the best Background SHAP performance.

In our experiments, we trained a LIGHTGBM classifier with 100 trees of varying depths. To allow deep paths while maintaining a realistic overall tree size, we increased the maximum number of leaves to 2024 (from the default of 31) and set `min_data_in_leaf = 500` to limit overfitting. All algorithms were executed sequentially in Google Colab's 50GM RAM CPU environment. GPU experiments used the Colab T4 environment with 50GM RAM.

This section focuses on Background SHAP, where WOODELFHD improves the state-of-the-art complexity. Results for Path-Dependent SHAP on the IEEE-CIS dataset, along with three additional datasets, are presented in Appendix E. In these experiments, WOODELFHD outperforms the SHAP package, FASTTREESHAP, and the original WOODELF algorithm for tree depths up to 15. However, at depths of 18 and above, the SHAP package becomes faster.

Table 3 summarizes the runtime of the Background SHAP and Background SHAP interaction values tasks on the IEEE-CIS dataset across different tree depths. For clarity of presentation and to keep the experimental runtime manageable, we report results for depths in increments of 3.

[1]The IEEE-CIS dataset: `https://www.kaggle.com/c/ieee-fraud-detection`

Table 3: Runtime of Background SHAP (BG SHAP) and Shapley interaction values (BG SHAP IV) on the IEEE-CIS dataset. Columns d=6, d=9, etc. denote tree depth; WHD GPU is WOODELFHD on GPU. "X" marks runs failed due to excessive memory usage (>50GB RAM). BG SHAP IV is not implemented in SHAP. Estimations: * marks runtimes measured on one tree and scaled to 100 trees. SHAP runtimes were measured with 10 background samples and extrapolated.

Task	**Algorithm**	d=6	d=9	d=12	d=15	d=18	d=21
BG SHAP	SHAP	33 days	61 days	97 days	134 days	173 days	209 days
	PLTREESHAP	186 sec	39 min	3.8 hrs	12.8 hrs*	X	X
	WOODELF	12 sec	84 sec	58 min	36 hrs*	X	X
	WOODELFHD	14 sec	47 sec	104 sec	285 sec	24 min	3 hrs
	WHD GPU	18 sec	44 sec	99 sec	171 sec	4.6 min	12 min
BG SHAP IV n=10 000	PLTREESHAP	63 sec	383 sec	33 min	X	X	X
	WOODELF	9 sec	108 sec	2.2 hrs	X	X	X
	WOODELFHD	9 sec	32 sec	111 sec	17 min	2.6 hrs	X
	WHD GPU	15 sec	50 sec	118 sec	3.8 min	12 min	X

To examine the depths of tuned models, Appendix G discuss hyperparameter tuning on five ensemble types across four datasets. The resulting depths ranged from 4 to 60. WOODELFHD supported 15 of the 20 tuned models while WOODELF supported only 10 of them.

As shown in the table, computing Background SHAP on a large background dataset using the SHAP package is impractical. Since the complexity of the SHAP algorithm scales as $O(mn)$, its runtime can reach several days even for shallow trees. PLTREESHAP and WOODELF reduce this to $O(m+n)$ and achieve substantially better performance. Consistent with the results reported in (Nadel & Wettenstein, 2026), WOODELF is more efficient than PLTREESHAP. At depth 6, WOODELF is slightly faster than WOODELFHD due to overhead from the space-optimized *DecisionPatternsGenerator*.

However, since both PLTREESHAP and WOODELF have time and space complexities that depend on $O(3^D D)$, they struggle with deeper trees: their runtimes increase sharply at depths 12 and 15, and at higher depths they exceed the available 50GB of memory. In contrast, WOODELFHD scales to deeper trees while providing substantial speedups even at moderate depths. For Background SHAP on CPU, at depth 12 it achieves a 33× speedup over the previous state of the art, and at depth 15 it achieves a 162× speedup. For Background SHAP IV, the speedups are even larger, and for deep trees, GPU acceleration further improves performance. Experiments on three additional datasets, presented in Appendix E, match these trends: WOODELFHD is the top-performing method, consistently outperforming all competing approaches for Background SHAP.

Overall, these results show that exact SHAP computation, long considered impractical for deep trees, is now efficient and scalable in practice.

6 CONCLUSION

By reducing the space and time complexity dependence on tree depth D from $O(3^D D)$ to $O(2^D D^2)$, WOODELFHD significantly advances the state of the art for Background SHAP. On the IEEE-CIS dataset, WOODELFHD achieves a 33× speedup at depth 12 and a 162× speedup at depth 15. Moreover, it enables the computation of Shapley values at depths 18 and 21 when running in standard environments, where previous methods fail due to excessive memory usage.

While WOODELFHD handles most practical settings, very deep trees can remain challenging. The ultimate goal is either to discover an $O(m+n)$ Background SHAP algorithm with complexity that is purely polynomial in D, enabling efficient Background SHAP computation at any depth, or to establish formal lower bounds that rule out such an algorithm, thereby providing fundamental insight into the inherent computational limits of Background SHAP. We believe that WOODELFHD, through its Strassen-like multiplication scheme, provides a promising step toward this goal. The complexity gains and practical performance demonstrated in this work offer an encouraging proof-of-concept for utilizing such schemes to navigate the inherent limits of Background SHAP.

AI USE STATEMENT

In this work, we used generative AI tools for manuscript rephrasing and related-literature discovery. We wrote the first draft of the paper ourselves and used ChatGPT to improve the phrasing and help fit the manuscript within the 9-page limit. In addition, ChatGPT assisted in the preparation of Fig. 3, and Claude was used in the experiments in Appendix G and in generating the html reports discussed in Appendix F. We reviewed all generated code and figures.

We did not use generative AI tools to write the main algorithm or to build and run the other experiments. We did not use generative AI tools to formalize mathematical statements or write mathematical proofs.

We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The algorithm code is available in the supplementary material, see the `code_and_notebooks/woodelfhd.py` file. The experiments are also easily reproducible: we provide the same notebooks used to run the experiments in the supplementary material, and their outputs include all runtime results reported in the paper. We clearly state the environment used for our experiments: Google Colab CPU and T4 GPU with high RAM enabled. We deliberately used Google Colab because it is an environment that is widely available to researchers in the ML community. All datasets are publicly available, and links to them are provided in the paper. The paper also includes pseudocode for our approach, enabling readers to implement it from scratch if desired.

REFERENCES

Mohamed Agalit and Youness Khamlichi. Optimization of intrusion detection with deep learning: A study based on the kdd cup 99 database. *International Journal of Safety and Security Engineering*, 14:1029–1038, 08 2024. doi: 10.18280/ijsse.140402.

Sreenivas Prasad B, Akash Babu N, Harthikeswar Reddy, Rimjhim Padam Singh, and Sneha Kanchan. A machine learning approach for credit card fraud detection in massive datasets using smote and random sampling. In *2024 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, pp. 1–8, 2024. doi: 10.1109/RAICS61201.2024.10690025.

John F. Banzhaf. Weighted voting doesn't work: A mathematical analysis. *Rutgers Law Review*, 19: 317–343, 1965.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. doi: 10.1023/A:1010933404324.

L. De Carli, A. Echezabal, and I. Morell. On the sierpiński triangle and its generalizations. *Journal of Mathematics and the Arts*, 19(3-4):79–92, 2025. doi: 10.1080/17513472.2025.2576836. URL `https://doi.org/10.1080/17513472.2025.2576836`.

Liiie Chen, Qihan Guan, Ning Chen, and Zhou YiHang. A stacknet based model for fraud detection. In *2021 2nd International Conference on Education, Knowledge and Information Management (ICEKIM)*, pp. 328–331, 2021. doi: 10.1109/ICEKIM52309.2021.00079.

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*, KDD '16, pp. 785–794, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/2939672.2939785.

Wenkai Deng, Ziming Huang, Jiachen Zhang, and Junyan Xu. A data mining based system for transaction fraud detection. In *2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE)*, pp. 542–545, 2021. doi: 10.1109/ICCECE51280.2021.9342376.

Robert L. Devaney. Measure, topology, and fractal geometry. *SIAM review*, 33(4):668–669, 1991. ISSN 0036-1445.

NumPy Developers. Nep 38 – simd optimizations. https://numpy.org/neps/nep-0038-SIMD-optimizations.html, 2019.

NumPy Developers. Simd optimizations documentation. https://numpy.org/doc/stable/reference/simd/index.html, 2024.

Anna Veronika Dorogush, Andrey Gulin, Gleb Gusev, Nikita Kazeev, Liudmila Ostroumova Prokhorenkova, and Aleksandr Vorobev. Fighting biases with dynamic boosting. *CoRR*, 2017. URL http://arxiv.org/abs/1706.09516.

Pradeep Dubey, Abraham Neyman, and Robert James Weber. Value theory without efficiency. *Mathematics of Operations Research*, 6(1):122–128, 1981. ISSN 0364765X, 15265471. URL http://www.jstor.org/stable/3689271.

Khashayar Filom, Alexey Miroshnikov, Konstandinos Kotsiopoulos, and Arjun Ravi Kannan. On marginal feature attributions of tree-based models. *Foundations of Data Science*, 6(4):395–467, 2024. doi: 10.3934/fods.2024021. URL https://www.aimsciences.org/article/id/6640081f475da12c51d5e2f8.

Fabian Fumagalli, Maximilian Muschalik, Patrick Kolpaczki, Eyke Hüllermeier, and Barbara Hammer. Shap-iq: unified approximation of any-order shapley interactions. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, Apr 2006. ISSN 1573-0565. doi: 10.1007/s10994-006-6226-1. URL https://doi.org/10.1007/s10994-006-6226-1.

Navdeep Gill, Patrick Hall, Kim Montgomery, and Nicholas Schmidt. A responsible machine learning workflow with focus on interpretable models, post-hoc explanation, and discrimination testing. *Information*, 11(3), 2020. doi: 10.3390/info11030137.

Michel Grabisch and Marc Roubens. An axiomatic approach to the concept of interaction among players in cooperative games. *International Journal of Game Theory*, 28(4):547–565, Nov 1999. doi: 10.1007/s001820050125.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2.

Christopher Hughes. Single-instruction multiple-data execution. *Synthesis Lectures on Computer Architecture*, 10:1–121, 05 2015. doi: 10.2200/S00647ED1V01Y201505CAC032.

Shanshan Jiang, Ruiting Dong, Jie Wang, and Min Xia. Credit card fraud detection based on unsupervised attentional anomaly detection network. *Systems*, 11:305, 06 2023. doi: 10.3390/systems11060305.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

Weida Li, Yaoliang Yu, and Bryan Kian Hsiang Low. Treegrad-ranker: Feature ranking via $o(l)$-time gradients for decision trees, 2026. URL https://arxiv.org/abs/2602.11623.

Jinyang Liu, Tessa Steensgaard, Marvin N. Wright, Niklas Pfister, and Munir Hiabu. Fast estimation of partial dependence functions using trees. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 39496–39534. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/liu25bm.html.

Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Scott M. Lundberg, Gabriel G. Erion, and Su-In Lee. Consistent individualized feature attribution for tree ensembles, 2019. URL https://arxiv.org/abs/1802.03888.

Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2:56–67, 2020. doi: 10.1038/s42256-019-0138-9.

Rory Mitchell, Eibe Frank, and Geoffrey Holmes. Gputreeshap: massively parallel exact calculation of shap scores for tree ensembles. *PeerJ. Computer science*, 8:e880, 2022. ISSN 2167-9843. doi: 10.7717/peerj-cs.880. URL https://europepmc.org/articles/PMC9044362.

Christoph Molnar. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. Lulu.com, 2022. URL https://christophm.github.io/interpretable-ml-book/.

Alexander Nadel and Ron Wettenstein. From decision trees to boolean logic: A fast and unified shap algorithm. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(29):24476–24485, Mar. 2026. doi: 10.1609/aaai.v40i29.39630. URL https://ojs.aaai.org/index.php/AAAI/article/view/39630.

R. Kelley Pace and Ronald Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997. doi: 10.1016/S0167-7152(96)00140-X.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Bernhard Pfahringer. Winning the kdd99 classification cup: bagged boosting. *SIGKDD Explor. Newsl.*, 1(2):65–66, January 2000. doi: 10.1145/846183.846200.

Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, 20(53):1–32, 2019.

L. S. Shapley. A value of n-person games. *Contributions to the Theory of Games*, pp. 307–317, 1953.

Wacław Sierpiński. Sur une courbe dont tout point est un point de ramification. *Comptes Rendus de l'Académie des Sciences de Paris*, 160:302–305, 1915.

Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A. Ghorbani. A detailed analysis of the kdd cup 99 data set. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pp. 1–6, 2009. doi: 10.1109/CISDA.2009.5356528.

Robert J. Weber. Probabilistic values for games. *The Shapley Value: Essays in Honor of Lloyd S. Shapley*, pp. 101–119, 1988.

Ron Wettenstein, Alexander Nadel, and Udi Boker. Woodelf++: A fast and unified partial dependence plot algorithm for decision tree ensembles, 2026. URL https://arxiv.org/abs/2605.14578.

Zhijia Xiao. Ieee-cis fraud detection based on xgb. In Xiaolong Li, Chunhui Yuan, and John Kent (eds.), *Proceedings of the 7th International Conference on Economic Management and Green Development*, pp. 1785–1796, Singapore, 2024. Springer Nature Singapore.

Artjom Zern, Klaus Broelemann, and Gjergji Kasneci. Interventional shap values and interaction values for piecewise linear regression trees. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(9):11164–11173, Jun. 2023. doi: 10.1609/aaai.v37i9.26322.

		add x_1			
		add x_2		add x_2	
add $\neg x_1$	add $\neg x_2$	$\emptyset$	$\emptyset$	$\emptyset$	$(\neg x_1 \wedge \neg x_2)$
		$\emptyset$	$\emptyset$	$(\neg x_1 \wedge x_2)$	$(\neg x_1)$
	add $\neg x_2$	$\emptyset$	$(x_1 \wedge \neg x_2)$	$\emptyset$	$(\neg x_2)$
		$(x_1 \wedge x_2)$	(x_1)	(x_2)	$()$

Figure 6: Example of the M_{cubes} matrix for x_1 and x_2.

			add x_1							
			add x_2				add x_2			
			add x_3		add x_3		add x_3		add x_3	
add $\neg x_1$	add $\neg x_2$	add $\neg x_3$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$(\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$
			$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$(\neg x_1 \wedge \neg x_2 \wedge x_3)$	$(\neg x_1 \wedge \neg x_2)$
		add $\neg x_3$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$(\neg x_1 \wedge x_2 \wedge \neg x_3)$	$\emptyset$	$(\neg x_1 \wedge \neg x_3)$
			$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$(\neg x_1 \wedge x_2 \wedge x_3)$	$(\neg x_1 \wedge x_2)$	$(\neg x_1 \wedge x_3)$	$(\neg x_1)$
	add $\neg x_2$	add $\neg x_3$	$\emptyset$	$\emptyset$	$\emptyset$	$(x_1 \wedge \neg x_2 \wedge \neg x_3)$	$\emptyset$	$\emptyset$	$\emptyset$	$(\neg x_2 \wedge \neg x_3)$
			$\emptyset$	$\emptyset$	$(x_1 \wedge \neg x_2 \wedge x_3)$	$(x_1 \wedge \neg x_2)$	$\emptyset$	$\emptyset$	$(\neg x_2 \wedge x_3)$	$(\neg x_2)$
		add $\neg x_3$	$\emptyset$	$(x_1 \wedge x_2 \wedge \neg x_3)$	$\emptyset$	$(x_1 \wedge \neg x3)$	$\emptyset$	$(x_2 \wedge \neg x_3)$	$\emptyset$	$(\neg x_3)$
			$(x_1 \wedge x_2 \wedge x_3)$	$(x_1 \wedge x_2)$	$(x_1 \wedge x_3)$	(x_1)	$(x_2 \wedge x_3)$	(x_2)	(x_3)	$()$

Figure 7: Example of the M_{cubes} matrix for x_1, x_2, x_3.

A A BOOLEAN LOGIC VIEW OF THE *M* MATRICES

In this section, we define the WOODELF M matrix using only pseudo-Boolean terminology. The definition relies solely on the notion of a pseudo-Boolean function (Def. 3) and the definition of a cube (Def. 4). It does not refer to Shapley values or decision trees. We believe this perspective is advantageous because it generalizes our results and may facilitate their application in other domains, such as Boolean optimization and logic.

Definition 3 (PB Function)**.** *A* Pseudo Boolean function *is a function of the form* $F(x_1, \ldots, x_h) : \{0,1\}^h \to \mathbb{R}$.

Definition 4 (Positive and Negative Literal; Cube; WDNF)**.** *A* literal *is a Boolean variable* x_i *(*positive literal*) or its negation* $\neg x_i$ *(*negated literal*). A* cube *is a conjunction of literals. A* Weighted Disjunctive Normal Form (WDNF) *formula represents a PB function as:*

$$F(x_1, \ldots, x_h) = \sum_{k=1}^{n} w_k \cdot c_k(x_1, \ldots, x_h),$$

where each c_k *is a cube and* $w_k \in \mathbb{R}$ *is its weight.*

Next, we define the linearity property over WDNF:

Definition 5 (Linearity)**.** *A function g over WDNF formulas satisfies* linearity *if its value on a WDNF formula* $F = \sum_{k=1}^{n} w_k \cdot c_k$ *can be expressed as the weighted sum of its values on the individual cubes:*

$$g(F) = \sum_{k=1}^{n} w_k \cdot g(c_k)$$

Using this pseudo-Boolean terminology (Defs. 3 to 5), we first define the cube matrix M_{cubes}. This matrix will later be used to define the M matrices used by WOODELF. Given Boolean variables $x_1, \ldots, x_D$ their M_{cubes} is a $2^D \times 2^D$ matrix that can be constructed as follows:

1. The boolean representation of the row index is treated as a list of negated literal indicators. If the i-th most significant bit is 0 add the literal $\neg x_i$ to the cube.
2. The boolean representation of the column index is treated as a list of positive literal indicators. If the i-th most significant bit is 0 add the literal x_i to the cube.
3. Remove cubes that have both the positive and the negated literal of the same variable as they are unsatisfiable.

Def. 6 formalize these rule:

Definition 6. *Let $x_1, \ldots, x_D$ be Boolean variables. For any $a, b \in \{0,1\}^D$, the cubes matrix M_{cubes} is defined as follows (Notation: $a \vee b$ is a bitwise OR and a_i is the i most significant bit of a):*

$$M_{cubes}[a][b] \;=\; \Big(\bigwedge_{i:\, a_i=0} \neg x_i \;\wedge \bigwedge_{i:\, b_i=0} x_i \quad \textit{if } a \vee b = 2^D - 1 \;\; \textit{else } \emptyset \Big) \tag{2}$$

By construction, M_{cubes} includes all possible satisfiable cubes over the Boolean variables $x_1, \ldots, x_D$. The M_{cubes} for $D = 2$ and $D = 3$ are visualized in Fig. 6 and Fig. 7, respectively.

We can now define the WOODELF M matrix.

Definition 7 (The WOODELF M matrix)**.** *The WOODELF M matrix is constructed by applying a linear function on a cube matrix. For a cube matrix M_{cubes}, a function v that satisfies linearity and any $a, b \in \{0,1\}^D$, a WOODELF M matrix can be constructed as follows:*

$$M[a][b] \;=\; v(M_{cubes}[a][b])$$

The results and algorithms presented in the Strassen-like multiplication section (Sect. 3) apply to any M matrix that satisfies Def. 7.

B VECTORIZED STRASSEN-LIKE MATRIX-VECTOR MULTIPLICATION

This section provides intuition and further details about the *StrassenLikeMult* algorithm presented in Fig. 5.

In Sect. 3, we propose an efficient matrix–vector multiplication algorithm that exploits the structure of the WOODELF matrices. By leveraging the identities $M_1 = 0$ and $M_4 = M_2 + M_3$, the computation is reduced from four sub-matrix multiplications to two at each level of the recursion (see Fig. 4).

Alg. 2 presents a naive recursive implementation of this idea. While conceptually simple, it incurs significant overhead as the depth D increases, leading to 2^D independent recursive calls. In contrast, our vectorized implementation processes all nodes at each recursion depth simultaneously using vectorized operations. As illustrated in Fig. 8, the vectorized computation can be viewed as two phases corresponding to the recursive downward and upward pass:

- Downward Pass: We perform the $v_1 + v_2$ additions for every recursive depth in a single vectorized step using bit-masks and shifts.
- Upward Pass: After the core multiplication, propagate the results upward by aggregating the $M_2 v_2$ and $M_3(v_1 + v_2)$ components using similar shift-based operations.

Fig. 8 compares the recursive and vectorized executions side by side and highlights how each vectorized operation corresponds to a specific recursion depth. Although the theoretical complexity remains $O(2^D D)$, the vectorized implementation is significantly faster in practice. It replaces an exponential number of recursive calls with only $O(D)$ vectorized operations, which can be efficiently implemented using SIMD (Developers, 2019; 2024).

Algorithm 2 Recursive $M \cdot v$ Strassen-Like Multiplication

function MvRecursive(M, v)
  if M and v are scalars **then**
    return $M \cdot v$
  ▷ Split v to halves and M to quarters.
  $v_1, v_2 = v$
  $M_1, M_2, M_3, M_4 = M$
  ▷ Assume $M_1 = 0$ and $M_2 + M_3 = M_4$.
  ▷ Downward Pass:
  $r_1 = \text{MvRecursive}(M_2, v2)$
  $r_2 = \text{MvRecursive}(M_3, v1 + v2)$
  ▷ Upward Pass:
  $r = \textit{np.concat}([r_1, r_1 + r_2])$
  return r

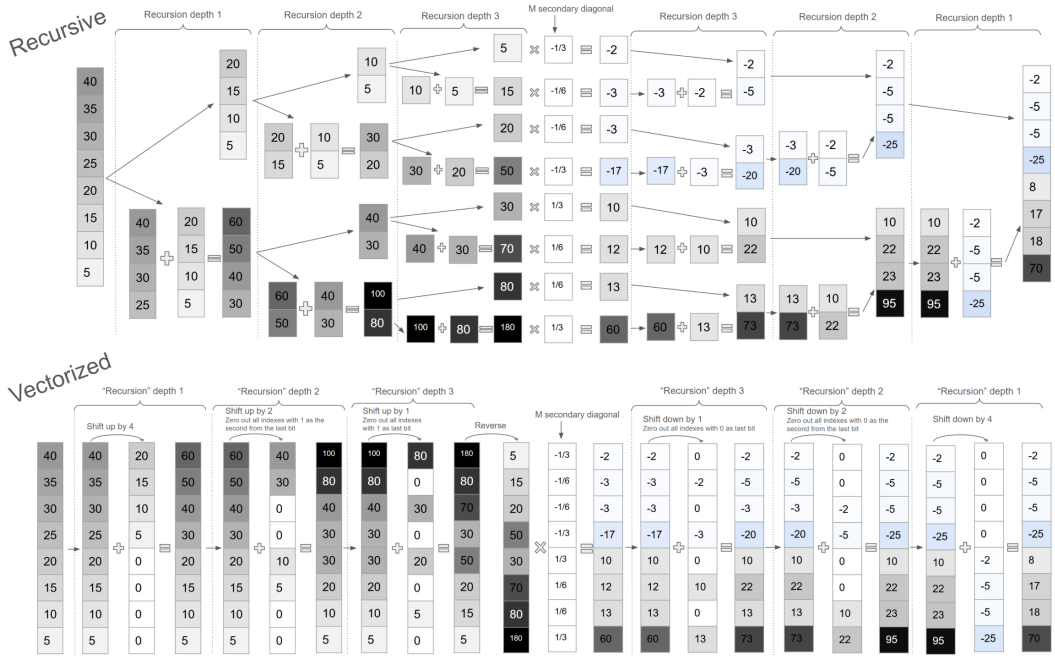


Figure 8: Recursive and vectorized multiplication, as formalized by Alg. 2 and Fig. 5, applied to a simple example. The figure illustrates how the vectorized approach performs all computations at a given recursion depth using just two steps: one operating on the input vector during the downward pass, and one operating on the resulting vector during the upward propagation.

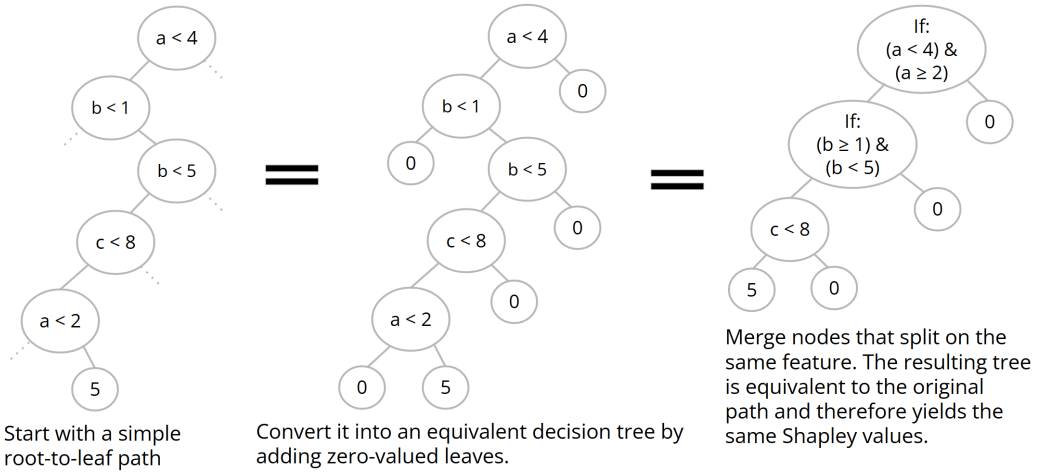


Figure 9: Mathematical justification for merging all same-feature splits into a single constraint.

C Pseudo Codes

C.1 ComputeMs Pseudo Code

Algorithm 3 Compute and cache all possible Ms

```
function ComputeMs(D)
    Ms = {}
    for k in 1, ..., D do
        d = CubesInDiagonal([1, ..., k])
        for index in d do
            cube = d[index]
            for feature, value in v(cube) do
                Ms[k][feature][index] = value
    return Ms
```

C.2 DecisionPatternsGenerator Pseudo Code

The *DecisionPatternsGenerator* algorithm is discussed in Sect. 4. It returns a dictionary P that maps each leaf to the unique features decision patterns of its consumers, where $P[l_j][c_i]$ stores the pattern of consumer c_i at leaf l_j. The same procedure can be applied to a background dataset, computing the pattern of each background row b_i at leaf l_j.

Alg. 4 presents the *DecisionPatternsGenerator* algorithm pseudo code. We use the following notation: $n.l$ and $n.r$ denote the left and right child of the node n, respectively. For two integers a, b, the operation $a\&b$ denotes the bit-wise AND of their Boolean representation.

The *DecisionPatternsGenerator* algorithm replaces the *CalcDecisionPatterns* algorithm used in WOODELF and significantly improves its space complexity. For reference, the pseudocode of *CalcDecisionPatterns* is also provided in this section (see Alg. 5). Note that *CalcDecisionPatterns* computes decision patterns (Def. 1), whereas *DecisionPatternsGenerator* computes unique-feature decision patterns (Def. 2).

C.3 The WoodelfHD Algorithm

The original WOODELF algorithm executes the four logical steps described in Sect. 2 independently for each tree, storing in memory for every leaf its corresponding f vectors, M matrices, and s vectors. At peak memory usage, the algorithm maintains D matrices of size 3^D and $2D$ vectors of size 2^D per leaf, resulting in a space complexity of $O(L\,3^D D)$.

Algorithm 4 Generate the UFDP for all leaves

function DECISIONPATTERNSGENERATOR(T, C)
  $P \leftarrow \{r_T : (0)_{\forall c \in C}\}$
  $F \leftarrow \{r_T : []\}$ ▷ Map nodes to their path features
  for n in DFS(T) **do**:
    if n is a leaf **then**
      yield $P[n]$
    else
      ▷ Update F
      if $n.feature \notin F[n]$ **then**
        $F[n.l] = F[n] + [n.feature]$
        $F[n.r] = F[n] + [n.feature]$
      else
        $F[n.l] = F[n]$
        $F[n.r] = F[n]$
      ▷ Update P with $n.l$'s and $n.r$'s UFDPs
      $s = n.\textit{split}(C)$
      if $n.feature \notin F[n]$ **then**
        $P[n.l] = (P[n] << 1) + s$
        $P[n.r] = (P[n] << 1) + \neg s$
      else
        $i = F[n].get_index(n.feature)$
        $f_{bit} = 2^{|F[n]|-1-i}$
        $mask = 2^{|F[n]|} - 1 - f_{bit}$
        $P[n.l] = P[n]\ \&\ (mask + f_{bit} \cdot s)$
        $P[n.r] = P[n]\ \&\ (mask + f_{bit} \cdot \neg s)$
      ▷ When possible, clean up the RAM
    if $n.parent.l \neq n$ **then**
      ▷ We can clean all the patterns of my left neighbor
      $P.\textit{pop}(n.parent.l)$
      $\textit{PopAllChildren}(P, n.parent.l)$

Algorithm 5 (Nadel & Wettenstein, 2026, Alg. 1) Mapping Each Leaf to Its Decision Patterns

function CALCDECISIONPATTERNS(T, C)
  $P_{leaves} \leftarrow \{\}$
  $P_{all} \leftarrow \{r_T : (0)_{\forall c \in C}\}$
  for n in BFS(T) **do**:
    if n is a leaf **then**
      $P_{leaves}[n] = P_{all}[n]$
    else
      $P_{all}[n.l] = (P_{all}[n] << 1) + n.\textit{split}(C)$
      $P_{all}[n.r] = (P_{all}[n] << 1) + \neg n.\textit{split}(C)$
  return P_{leaves}

In contrast, WOODELFHD processes each leaf independently. At any point, it keeps the $2D$ vectors associated with the leaf currently being processed. Combined with the observation that only the secondary diagonal of M needs to be computed and stored, this reduces the space complexity from $O(L3^D D)$ to $O(2^D D)$.

The pseudocode for WOODELFHD is given in Alg. 6. The algorithm takes as input a decision tree T, a consumer dataset C, a background dataset B, and a linear function v such as the Shapley value function. It computes and returns the v-values (e.g., Shapley values) for all consumers.

Algorithm 6 WOODELFHD algorithm

function WOODELFFORHIGHDEPTH(T, C, B, v)
  $values = \{\}$
  $cpg = DecisionPatternsGenerator(T, C)$
  $bpg = DecisionPatternsGenerator(T, B)$
  $M_{cache} =$ ***ComputeMs***$(T.depth)$
  for $l \in L_T$ **do** ▷ For each leaf l
    if $|B| > 0$ **then** ▷ Step 1, Background f
      $p_b = bpg.next()$
      $f = p_b.$***value_counts***$(normalize = True)$
    else ▷ Step 1, Path Dependent f
      Compute f using l and Formula 1
    ▷ Step 2, fetch M from cache
    $path = root_to_leaf_path(T, l)$
    $pf = unique((n.$***feature***$)_{\forall n \in path})$
    $M_{diag} = M_{cache}[\,|pf|\,]$
    $s = \{\}$ ▷ Step 3, compute s
    for $i, feature$ in $enumerate(pf)$ **do**
      $a =$ ***StrassenLikeMult***$(M_{diag}[i+1], f, T.Depth)$
      $s[feature] = l.weight \cdot a$
    $p_c = cpg.next()$ ▷ Step 4, the final values
    for $feature$ in s **do**
      $values[feature]$ += $s[feature][\,p_c\,]$
  return $values$

D WOODELFHD COMPLEXITY ON A GENERIC CONTRIBUTION FUNCTION

Importantly, WOODELF is not limited to Shapley values. It applies to any function v that operates on a WDNF formula (a sum of weighted cubes) and satisfies linearity. This includes Banzhaf values, Shapley interaction values, and other probabilistic values. In this section, we analyze the complexity of WOODELFHD for any linear contribution function v over WDNF.

The complexity depends on the cost of evaluating v over cubes defined on variables $\{x_1, \ldots, x_D\}$, where D is the tree depth. Let $O(v_c^{avg})$ denote the average cost of evaluating v on a single cube, and $O(v_s)$ the number of unique feature subsets returned across all cubes. For example, for Shapley values, each variable is considered separately, so $O(v_c^{avg}) = O(v_s) = O(D)$.

The resulting complexities of WOODELF are:

$$\text{Path-Dependent WOODELF: } O(nTLv_s + TL3^D v_c^{avg})$$

$$\text{Background WOODELF: } O(mTL + nTLv_s + TL3^D v_c^{avg})$$

WOODELFHD reduces the matrix-vector multiplication complexity from 3^D to $2^D D$ and evaluates v only on the secondary diagonal of the M_{cubes} matrix (see Def. 6), i.e., only on cubes of length D. Let $O(v_c^{davg})$ be the average cost of evaluating v on such cubes, and $O(v_{ds})$ the number of unique feature subsets returned. The Background WOODELFHD algorithm consists of four main components:

1. Computing the background UFDPs and the corresponding f vectors. This is the component that depends on the number of background samples m. In Path-Dependent SHAP, background UFDPs are not computed and the cost of constructing the f vectors is dominated by component 3 below. Therefore, this component appears only in Background SHAP and not in Path-Dependent SHAP.
2. Computing the consumer UFDPs and extracting the final values from the s vectors. This is the component that depends on the number of explained samples n.
3. Computing the matrix-vector products $s = M \cdot f$ for each leaf. WOODELFHD reduces this component from a dependence on 3^D to a dependence on $2^D D$.

4. Applying v to the secondary diagonals of the M_{cubes} matrices. This component depends on $O(v_c^{d_{\text{avg}}})$. In practice, this cost is greatly reduced by the cache constructed in ***ComputeMs***. Using this cache, the computation is performed once for each depth from 1 to D, rather than once for every leaf in the ensemble. We assume that the computation for matrices of depth D dominates this cost. This is reasonable because the diagonal size doubles with each increase in depth, so the secondary diagonal at depth D contains more cubes than all smaller depths combined. For Shapley and Shapley interaction values, this component is dominated by component 3 above, and therefore does not appear in the final complexity.

Combining these components gives the following time complexities:

$$\text{Path-Dependent WOODELFHD: } O(nTLv_{ds} + TL2^D D\, v_{ds} + 2^D v_c^{davg})$$

$$\text{Background WOODELFHD: } O(mTL + nTLv_{ds} + TL2^D D\, v_{ds} + 2^D v_c^{davg})$$

For Shapley values, $O(v_{ds}) = O(v_c^{davg}) = O(D)$, and for pairwise Shapley interaction values, $O(v_{ds}) = O(v_c^{davg}) = O(D^2)$. Substituting these gives the complexities reported in Table 1.

The complexities above assume that the cache in ***ComputeMs*** can be used. This requires v to be invariant to feature identities, that is, v must satisfy symmetry. This is usually the case: for example, all semivalues (Dubey et al., 1981) and all Cardinal Interaction Indexes (Grabisch & Roubens, 1999) are symmetric. WOODELFHD also supports nonsymmetric value functions. In this case, the current caching mechanism can not be used, and ***ComputeMs*** must be applied to all tree paths.

For nonsymmetric v, where the cache can not be used, the resulting time complexities are:

$$\text{Path-Dependent WOODELFHD: } O(nTLv_{ds} + TL2^D D\, v_{ds} + TL2^D v_c^{davg})$$

$$\text{Background WOODELFHD: } O(mTL + nTLv_{ds} + TL2^D D\, v_{ds} + TL2^D v_c^{davg})$$

Let s_{out} denote the output size for a single explained instance. For semivalues (including Shapley and Banzhaf), $s_{out} = f$, while for pairwise interactions $s_{out} = f^2$. The space complexities of WOODELFHD are:

$$\text{Path-Dependent WOODELFHD: } O(2^D v_{ds} + n(s_{out} + D))$$

$$\text{Background WOODELFHD: } O(2^D v_{ds} + n(s_{out} + D) + mD)$$

E ADDITIONAL EXPERIMENTAL RESULTS

This section presents additional experimental results. We evaluate our approach on several well-known large datasets: the KDD-99 intrusion detection dataset[2] (see related studies (Pfahringer, 2000; Tavallaee et al., 2009; Agalit & Khamlichi, 2024)), the HIGGS particle physics dataset[3], and the California Housing dataset[4] (Pace & Barry, 1997). Table 4 summarizes the datasets, including train/test sizes and the number of features after preprocessing.

As in Sect. 5, CPU experiments were run on the Google Colab high-memory CPU runtime, and GPU experiments were run on the T4 runtime. We trained LightGBM models with varying depths, at most 2024 leaves, and `min_data_in_leaf = 500`. For California Housing, which has a smaller training set, we used `min_data_in_leaf = 5`.

We compare the runtime of our approach on CPU and GPU to other SHAP methods, including the SHAP package, PLTREESHAP, FASTTREESHAP v2, FASTTREESHAP v1, and the original WOODELF algorithm. We analyze performance for both Background and Path-Dependent SHAP. Results are in Table 5 and Table 6.

We now discuss the main experimental findings.

[2]The KDD-99 dataset: `https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`

[3]The HIGGS dataset: `https://archive.ics.uci.edu/ml/datasets/HIGGS`

[4]The California Housing dataset: `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_california_housing.html`

Table 4: Details of the datasets used.

Name	Train Size	Test Size	Features
IEEE-CIS	472 432	118 108	397
HIGGS	8 800 000	2 200 000	28
KDD-99	4 898 431	2 984 154	121
California Housing	16 512	4128	8

Table 5: **Background (BG) SHAP results on additional datasets**. See Table 3 caption for explanation on the table layout. Runtime estimation details: entries marked with * were measured on 10 trees and extrapolated (multiplied by 10), entries marked with ** were measured on a single tree and extrapolated (multiplied by 100). Running times for SHAP were measured using a background dataset of size 10 and extrapolated (multiplied by $0.1 \cdot$ *train_size*).

Dataset	Task	Algorithm	d=6	d=9	d=12	d=15	d=18	d=21
HIGGS	BG SHAP	SHAP	396 yrs**	1131 yrs**	2416 yrs**	2892 yrs**	2969 yrs**	2970 yrs**
		PLTREESHAP	85 min*	50 hrs*	33 days**	X	X	X
		WOODELF	3.7 min	62 min*	X	X	X	X
		WOODELFHD	8 min	87 min*	7 hrs**	9 hrs**	11 hrs**	14 hrs**
		WHD GPU	1.7 min	15 min*	65 min**	67 min**	82 min**	3.7 hrs**
	BG SHAP IV n=10 000	PLTREESHAP	19 min*	6 hrs*	5 days**	X	X	X
		WOODELF	3.5 min	12 hrs*	X	X	X	X
		WOODELFHD	3.9 min	48 min*	3.6 hrs**	4.2 hrs**	7 hrs**	X
		WHD GPU	89 sec	13 min*	58 min**	64 min**	2.3 hrs**	X
KDD-99	BG SHAP	SHAP	1.3 yrs	1.9 yrs	2.9 yrs	3.9 yrs	5.1 yrs	6.1 yrs
		PLTREESHAP	55 min*	7 hrs*	6 hrs**	14 hrs**	X	X
		WOODELF	2.7 min	10 min	2.2 hrs**	12 hrs**	X	X
		WOODELFHD	2.5 min	9 min	20 min	41 min	75 min	118 min*
		WHD GPU	40 sec	104 sec	3.6 min	6.4 min	11 min	29 min*
	BG SHAP IV n=10 000	PLTREESHAP	3.6 min*	10 min*	11 min**	X	X	X
		WOODELF	8.3 sec	106 sec	2.6 hrs*	X	X	X
		WOODELFHD	9.2 sec	36 sec	159 sec	16.2 min	5.3 hrs*	X
		WHD GPU	30 sec	83 sec	173 sec	5.3 min	16 min*	X
California Housing	BG SHAP	SHAP	6 hrs	17 hrs	31 hrs	43 hrs	54 hrs	63 hrs
		PLTREESHAP	14 sec	119 sec	9 min	33 min**	X	X
		WOODELF	3 sec	20 min*	24 hrs**	19 days**	X	X
		WOODELFHD	3 sec	16 sec	38 sec	63 sec	83 sec	100 sec
		WHD GPU	22 sec	84 sec	3.5 min	6 min	8 min	9 min
	BG SHAP IV n=4128	PLTREESHAP	18 sec	2.5 min	17 min*	X	X	X
		WOODELF	2.4 sec	17 min	31 min**	X	X	X
		WOODELFHD	2.2 sec	11 sec	27 sec	45 sec	60 sec	70 sec
		WHD GPU	19 sec	91 sec	3.9 min	6 min	8 min	10 min

- **IEEE-CIS.** For Background SHAP, where WOODELFHD has state of the art complexity, our method consistently outperforms the other approaches (see Sect. 5). For Path-Dependent SHAP, the results are mixed. For Shapley values, WOODELFHD is the fastest method up to depth 15. At depth 18 and above, the 2^D component becomes dominant and SHAP becomes faster. For Path-Dependent Shapley interaction values, WOODELFHD outperforms the other approaches on all supported depths (depths 6–18).
- **KDD-99.** The results are similar to those on IEEE-CIS. For Background SHAP, WOODELFHD is consistently faster than the other methods. For Path-Dependent SHAP, WOODELFHD is the best-performing method up to depth 18 for Shapley values, and up to depth 15 for Shapley interaction values.
- **HIGGS.** With 11 million rows, this dataset simulates using SHAP on big data. The task is harder and the runtimes are longer. These large datasets are exactly where GPU implementations shine. On this dataset, WOODELFHD on GPU is $2.45\times$–$9.75\times$ faster than WOODELFHD on CPU. It also significantly outperforms all CPU baselines, except for Path-Dependent Shapley interaction values, which are computed on a sample of size 10 000.

Table 6: **Path-Dependent (PD) SHAP results**. See Table 5 caption for explanation on the table layout and estimation methodology. Here, the SHAP algorithm is run on the full test set.

Dataset	Task	Algorithm	d=6	d=9	d=12	d=15	d=18	d=21
IEEE-CIS	PD SHAP	SHAP	20 sec	81 sec	3.6 min	7 min	12 min	17 min
		FastTreeSHAP v1	61 sec	4 min	11 min	22 min	53 min*	83 min*
		FastTreeSHAP v2	41 sec	2.4 min	5.5 min	10 min	26 min*	51 min*
		WoodelF	6 sec	59 sec	56 min*	34 hrs**	X	X
		WoodelFHD	7 sec	27 sec	67 sec	3.3 min	15 min	3 hrs*
		WoodelFHD GPU	15 sec	39 sec	91 sec	2.8 min	5.6 min	41 min*
	PD SHAP IV n=10 000	SHAP	3.7 min	30 min	168 min*	3.2 hrs**	7 hrs**	15 hrs**
		FastTreeSHAP v1	4.5 min	31 min	115 min	7 hrs*	7 hrs**	15 hrs**
		WoodelF	2.4 sec	85 sec	2.4 hrs*	X	X	X
		WoodelFHD	2.6 sec	12 sec	57 sec	17 min*	2.8 hrs*	X
		WoodelFHD GPU	13 sec	45 sec	112 sec	6 min*	24 min*	X
HIGGS	PD SHAP	SHAP	9 min*	1.7 hrs**	10 hrs**	11 hrs**	11 hrs**	12 hrs**
		FastTreeSHAP v1	18 min	3 hrs**	16 hrs**	18 hrs**	18 hrs**	19 hrs**
		FastTreeSHAP v2	8 min	118 min*	15 hrs**	18 hrs**	19 hrs**	X
		WoodelF	159 sec	52 min*	X	X	X	X
		WoodelFHD	3.3 min	37 min*	3.2 hrs**	3.9 hrs**	4.8 hrs**	7.1 hrs**
		WoodelFHD GPU	28 sec	4.4 min*	22 min**	24 min**	40 min**	2.9 hrs**
	PD SHAP IV n=10 000	SHAP	4.3 min	56 min*	5 hrs**	5 hrs**	5 hrs**	5 hrs**
		FastTreeSHAP v1	115 sec	25 min	2.5 hrs	2.4 hrs*	2 hrs**	3.3 hrs**
		WoodelF	5 sec	40 min*	X	X	X	X
		WoodelFHD	5 sec	50 sec	5.5 min	11 min	18 min*	X
		WoodelFHD GPU	25 sec	3.5 min	17 min	25 min	30 min*	X
KDD-99	PD SHAP	SHAP	8 min	30 min	62 min*	128 min*	82 min**	111 min**
		FastTreeSHAP v1	23 min	86 min	2.9 hrs*	5.7 hrs*	3.9 hrs**	5.2 hrs**
		FastTreeSHAP v2	12 min	28 min	62 min	2.1 hrs	3.6 hrs*	X
		WoodelF	13 sec	85 sec	62 min*	35.9 hrs**	X	X
		WoodelFHD	16 sec	51 sec	122 sec	301 sec	25 min	5.4 hrs*
		WoodelFHD GPU	16 sec	51 sec	113 sec	3.5 min	6 min	23 min*
	PD SHAP IV n=10 000	SHAP	2.6 min*	13 min*	38 min*	87 min*	164 min*	280 min*
		FastTreeSHAP v1	3.1 min	13 min	37 min	56 min*	42 min**	59 min**
		WoodelF	8.3 sec	106 sec	2.6 hrs*	X	X	X
		WoodelFHD	9.2 sec	36 sec	159 sec	16.2 min	5.3 hrs*	X
		WoodelFHD GPU	14 sec	43 sec	99 sec	201 sec	14 min*	X
California Housing	PD SHAP	SHAP	0.7 sec	2.8 sec	10 sec	17 sec	24 sec	29 sec
		FastTreeSHAP v1	1.9 sec	11 sec	33 sec	54 sec	73 sec	88 sec
		FastTreeSHAP v2	0.9 sec	5 sec	18 sec	34 sec	46 sec	X
		WoodelF	2.6 sec	14 min	12 days**	18 days**	X	X
		WoodelFHD	1.5 sec	7 sec	18 sec	29 sec	38 sec	46 sec
		WoodelFHD GPU	17 sec	78 sec	3.3 min	6 min	7 min	9 min
	PD SHAP IV n=4128	SHAP	22 sec	115 sec	9 min*	15 min**	18 min**	18 min**
		FastTreeSHAP v1	18 sec	85 sec	3.7 min	6 min	8 min	9 min
		WoodelF	1.7 sec	19 min	34 hrs*	X	X	X
		WoodelFHD	1.8 sec	9 sec	23 sec	38 sec	51 sec	62 sec
		WoodelFHD GPU	17 sec	83 sec	3.5 min	5.9 min	8 min	10 min

- **California Housing.** This dataset is interesting for two reasons. First, it is small, with roughly 20 000 rows. Second, and more importantly, it has only 8 features. A closer look at WoodelFHD shows that its complexity depends on the length of the Unique Features Decision Pattern (UFDP). In the 2^D component, D can be replaced by the maximum number of bits in the longest UFDP. This quantity is bounded by the tree depth, but also by the number of features in the dataset. Since California Housing has only 8 features, the longest possible UFDP has length at most 8, regardless of the tree depth. As a result, WoodelFHD supports all tested depths on this dataset. In additional experiments, we successfully ran WoodelFHD on this dataset with depths up to 60. This also leads to practical runtimes: for example, Background SHAP at depth 60 takes only 162 seconds. As shown in Table 5 and Table 6, WoodelFHD significantly outperforms the other approaches for Background SHAP and Path-Dependent SHAP IV. In Path-Dependent SHAP, the SHAP package is the fastest approach.

Overall, WOODELFHD is consistently faster for Background SHAP, and outperforms the other Path-Dependent approaches at depths below 18. On very large datasets and deep trees, GPU computation can provide substantial speedups, while in other cases the CPU version may be preferable. Finally, on datasets with few features relative to the tree depth, WOODELFHD can efficiently support very deep trees.

E.1 MEMORY USAGE

We evaluate the memory usage of WOODELFHD on the IEEE-CIS dataset under the same model and environment settings as in the experiments presented in Table 3. The base memory consumption of the system, including the operating system and data loading, is approximately 10GB. We therefore report the additional (delta) memory used by the algorithm itself.

Table 7: Peak additional memory usage (in MB) of WOODELFHD on IEEE-CIS across tree depths.

Task	d=6	d=9	d=12	d=15	d=18	d=21
Background SHAP (MB)	0.4	0.1	3.2	55.0	487.1	5284.1
Background SHAP IV n=10 000 (MB)	0.0	39.4	462.1	1102.8	3396.2	X

The results follow the expected space complexity of WOODELFHD, which depends on $O(2^D \cdot D)$. At small depths, memory usage is negligible, as the 2^D term remains small. In particular, usage at depth 15 for Background SHAP is still negligible, and even at depth 18 the memory requirements remain moderate. Only at higher depths does the exponential factor become dominant, leading to a sharp increase in memory usage, with runs exceeding 50GB at depth 24 for Background SHAP and at depth 21 for Background SHAP interaction values.

E.2 EMPIRICAL CORRECTNESS VERIFICATION

To validate the correctness of WOODELFHD, we conducted two experiments on the first 1000 consumers of the IEEE-CIS dataset, using the first 80 rows as the background data. First, we compared the outputs of WOODELFHD with those of the SHAP Python package on a depth 9 LightGBM model. We evaluated Background SHAP, Path-Dependent SHAP, and Path-Dependent SHAP interaction values (note that SHAP does not support Background SHAP interaction values). The results showed strong agreement, with all differences below 10^{-5}.

Second, we compared WOODELFHD with the original WOODELF implementation. We ran this test on XGBoost to also validate correctness on an additional ensemble type. Specifically, we trained a depth-6 XGBoost model and evaluated Background SHAP, Path-Dependent SHAP, Path-Dependent SHAP IV, and Background SHAP IV. Again, all differences were below 10^{-5}.

F INTERACTIVE EXPERIMENTAL REPORT

In the supplementary material, we provide four HTML reports with experiments similar to those discussed above, along with additional results. The report helps researchers build intuition about the runtimes of the different methods, view plots of runtime across tree depths, and filter experiments into subsets (e.g., high-depth trees or interaction value computations).

We include four reports:

1. **The main_experiment report**: These results are very similar to those in Table 3, Table 5, and Table 6. The report uses the four datasets discussed above (IEEE-CIS, KDD-99, HIGGS, and California Housing) and compares the methods on Background SHAP, Background SHAP IV, Path-Dependent SHAP and Path-Dependent SHAP IV. This report does not include WOODELFHD GPU.
2. **The main_experiment_with_GPU report**: Similar to `main_experiment`, but includes WOODELFHD GPU.
3. **The higgs_shap_vs_treegrad_experiment**: This experiment visits the TreeGrad (Li et al., 2026) algorithm and shows that, although it has better theoretical complexity than Tree-

SHAP, it performs worse in practice, likely due to implementation factors. We compare both approaches on the HIGGS dataset.

4. **The woodelfhd_vs_original_low_depths_experiment**: This experiment focuses on WOODELF vs. WOODELFHD at low depths. As seen in Table 3, WOODELF can be faster at small depths. Here, we extend this analysis by training models with depths 1–10 on the IEEE-CIS dataset and comparing the two methods. WOODELF is faster up to depth 7, although the differences for depths 3–7 are relatively small (less than 10% on average).

The experiments in these reports were run on Google Colab (CPU and T4 high-memory machines). They were executed separately from those shown in Table 3, Table 5, and Table 6, so the reported runtimes may differ slightly. Additionally, the evaluation settings are not identical; for example, some experiments that use 10 trees in Table 5 are based on a single tree in the `main_experiment` report.

We do not provide the code used to generate these reports, as it is more complex and difficult to anonymize. We plan to release the HTML report framework as a separate open-source project in the camera-ready version, along with the notebooks used to generate the reports. These reports are intended to help readers interact with the results and explore them in a visually intuitive way. All claims in the paper rely only on the results in Table 3, Table 5, and Table 6, which are reproducible using the code and notebooks provided in the supplementary material.

G HYPERPARAMETER TUNING AND THE DEPTH OF TUNED MODELS

The experiments of Sect. 5 use fixed tree depths to study how runtime scales with depth. This appendix asks the complementary questions: when the depth is left to a hyperparameter search rather than prescribed, what depth do tuned models actually reach? Are these models supported by WOODELFHD?

Throughout this appendix we write D_u for the maximum UFDP length (see Def. 2) over all root-to-leaf paths of an ensemble. It is D_u, not the path length D, that is the exponent in the complexities of Tables 1 and 2.

Protocol. For each of five ensemble families and each of the four datasets of Sect. 5 and Appendix E. The ensemble families are: XGBoost (Chen & Guestrin, 2016), LightGBM (Ke et al., 2017), HistGradientBoosting (Pedregosa et al., 2011), RandomForest (Breiman, 2001) and Extra-Trees (Geurts et al., 2006). We draw 20 configurations by seeded random search over grids of five options per hyperparameter, and select on a validation split by R^2 for regression and ROC-AUC for classification. Ten of the 20 draws pin the depth parameter to the library's "no limit" configuration and ten draw a finite `max_depth`; every other hyperparameter is drawn from the same grid in both halves, so the two halves differ only in the depth constraint. The train/test split of each dataset is the one used in Sect. 5 and Appendix E; the validation split is carved out of the training half alone, so nothing in the search selects on test data. Overall, we trained 400 models. The notebook is provided in the supplementary material.

Tables 8 and 9 report the selected configuration of each (dataset, model) pair.

The depth-unlimited half supplies 13 of the 20 selected configurations. This is direct evidence for the practice that motivates this work: left to itself, a hyperparameter search often prefers to grow trees deep.

The margins, however, are small. Excluding California Housing, removing the depth limit improves the median test score by only 0.004, with a maximum gain of 0.026 (Extra Trees on IEEE-CIS). California Housing is the main exception, where the two bagging methods gain 0.030 and 0.087 in R^2. Thus, to achieve fast explainability, practitioners can often cap tree depth with only a small accuracy loss.

We run WOODELFHD on every tuned model it supports, that is every model with $D_u \leq 21$, computing Background SHAP for $n = 1000$ test rows and using the entire training split as the background dataset. The WHD column of Table 8 reports the runtimes, and marks with "X" the five models WOODELFHD does not support: three of them are tuned IEEE-CIS models, and two of them are on HIGGS. Extra Trees and HistGB models on HIGGS have $D_u \leq 21$ but when running WOODELFHD

Table 8: The configuration selected out of 20 per cell, ranked within each dataset by the validation metric. The metric is R^2 on California Housing and ROC-AUC elsewhere; the KDD Cup 99 validation scores are 1.0000 only after rounding. D is the longest root-to-leaf path of the fitted ensemble, D_u the largest number of unique features on such a path, and T the number of trees built. WHD is the WOODELFHD runtime for Background SHAP on $n = 1000$ test rows, using the whole training split as the background dataset: $m = 13\,209$ on California Housing, $64\,000$ on IEEE-CIS, and $160\,000$ on HIGGS and KDD Cup 99. "X" marks the five models with $D_u > 21$ that WOODELFHD does not support.

Dataset	Model	Depth mode	D	D_u	T	Val.	Test	WHD (sec)
California Housing	LIGHTGBM	unlimited	36	5	300	0.8502	0.8517	30
	HistGB	unlimited	36	8	800	0.8485	0.8483	30
	XGBoost	limited (6)	6	4	800	0.8441	0.8424	8
	Random Forest	unlimited	33	8	50	0.8054	0.7940	33
	Extra Trees	unlimited	41	8	300	0.8040	0.7921	365
HIGGS	XGBoost	unlimited	47	25	500	0.8224	0.8219	X
	HistGB	unlimited	35	20	500	0.8220	0.8227	614
	LIGHTGBM	unlimited	19	15	800	0.8194	0.8202	47
	Random Forest	unlimited	44	23	300	0.8083	0.8090	X
	Extra Trees	limited (24)	24	19	300	0.7898	0.7903	1219
IEEE-CIS	LIGHTGBM	limited (10)	10	10	300	0.9165	0.8943	23
	HistGB	unlimited	22	21	800	0.9146	0.8941	190
	XGBoost	unlimited	41	35	500	0.9132	0.8932	X
	Random Forest	unlimited	74	60	150	0.8800	0.8538	X
	Extra Trees	unlimited	64	58	300	0.8732	0.8623	X
KDD Cup 99	Extra Trees	limited (16)	16	16	150	1.0000	0.9910	22
	HistGB	limited (10)	10	10	500	1.0000	0.9781	11
	LIGHTGBM	limited (7)	7	7	200	1.0000	0.9820	4
	Random Forest	unlimited	23	20	50	1.0000	0.9734	97
	XGBoost	limited (10)	10	10	300	1.0000	0.9795	3

on 50GB RAM machine the run crashed on out of memory. Overall, WOODELFHD supports 15 of the 20 tuned models, where the original WOODELF supports only 10 (WOODELF supports only models with $D_u \leq 15$).

Table 9: The configuration selected in each cell, exactly as sampled. `max_depth` values of `0`, `-1` and `None` are the respective libraries' "no limit" sentinels, and `max_samples=None` disables bootstrapping for Extra Trees.

Dataset	Model	Selected hyperparameters
Cal. Housing	XGBoost	`max_depth=6, n_estimators=800, learning_rate=0.03, subsample=1.0, colsample_bytree=0.6, min_child_weight=50, reg_lambda=5.0`
	LightGBM	`max_depth=-1, num_leaves=255, n_estimators=300, learning_rate=0.03, min_child_samples=10, subsample=0.9, colsample_bytree=0.6, reg_lambda=10.0`
	HistGB	`max_depth=None, max_leaf_nodes=255, max_iter=800, learning_rate=0.03, min_samples_leaf=100, l2_regularization=100.0, max_features=0.8`
	Random Forest	`max_depth=None, n_estimators=50, max_features=0.25, min_samples_leaf=2, min_samples_split=10, max_samples=0.9`
	Extra Trees	`max_depth=None, n_estimators=300, max_features=0.5, min_samples_leaf=2, min_samples_split=2, max_samples=0.8`
HIGGS	XGBoost	`max_depth=0, n_estimators=500, learning_rate=0.03, subsample=0.8, colsample_bytree=1.0, min_child_weight=10, reg_lambda=5.0`
	LightGBM	`max_depth=-1, num_leaves=63, n_estimators=800, learning_rate=0.05, min_child_samples=10, subsample=0.9, colsample_bytree=0.6, reg_lambda=10.0`
	HistGB	`max_depth=None, max_leaf_nodes=255, max_iter=500, learning_rate=0.1, min_samples_leaf=50, l2_regularization=100.0, max_features=1.0`
	Random Forest	`max_depth=None, n_estimators=300, max_features=0.25, min_samples_leaf=5, min_samples_split=10, max_samples=0.9`
	Extra Trees	`max_depth=24, n_estimators=300, max_features=0.5, min_samples_leaf=10, min_samples_split=200, max_samples=None`
IEEE-CIS	XGBoost	`max_depth=0, n_estimators=500, learning_rate=0.05, subsample=0.9, colsample_bytree=0.9, min_child_weight=1, reg_lambda=5.0`
	LightGBM	`max_depth=10, num_leaves=127, n_estimators=300, learning_rate=0.1, min_child_samples=50, subsample=0.8, colsample_bytree=1.0, reg_lambda=10.0`
	HistGB	`max_depth=None, max_leaf_nodes=31, max_iter=800, learning_rate=0.05, min_samples_leaf=100, l2_regularization=10.0, max_features=0.8`
	Random Forest	`max_depth=None, n_estimators=150, max_features=0.5, min_samples_leaf=2, min_samples_split=10, max_samples=0.7`
	Extra Trees	`max_depth=None, n_estimators=300, max_features=0.5, min_samples_leaf=10, min_samples_split=2, max_samples=0.9`
KDD Cup 99	XGBoost	`max_depth=10, n_estimators=300, learning_rate=0.1, subsample=0.6, colsample_bytree=1.0, min_child_weight=1, reg_lambda=50.0`
	LightGBM	`max_depth=7, num_leaves=127, n_estimators=200, learning_rate=0.1, min_child_samples=5, subsample=0.6, colsample_bytree=0.6, reg_lambda=0.1`
	HistGB	`max_depth=10, max_leaf_nodes=255, max_iter=500, learning_rate=0.3, min_samples_leaf=100, l2_regularization=10.0, max_features=0.4`
	Random Forest	`max_depth=None, n_estimators=50, max_features='sqrt', min_samples_leaf=2, min_samples_split=10, max_samples=0.8`
	Extra Trees	`max_depth=16, n_estimators=150, max_features=0.75, min_samples_leaf=1, min_samples_split=50, max_samples=0.9`