# Understanding a Fast Algorithm to Obtain Shapley Values for Tree Models

**Hugh Chen** hughchen@cs.washington.edu
*Paul G. Allen School of CSE*
*University of Washington*
*Seattle, WA*

**Scott Lundberg** scottmlundberg@gmail.com
*Microsoft Research*
*Redmond, WA*

**Su-In Lee** suinlee@cs.washington.edu
*Paul G. Allen School of CSE*
*University of Washington*
*Seattle, WA*

## Abstract

Shapley values have been adapted to explain machine learning algorithms; however, calculating them is NP-hard. By restricting our machine learning model class to trees, we showed that they can be calculated exactly in linear time and evaluated them experimentally in Lundberg et al. (2020). However, the underlying algorithms are hard to understand. Because explainable methods can easily be mis-used due to lack of understanding, this article focuses on better understanding Shapley values and our algorithm for explaining trees.

## 1. Introduction

Nowadays, machine learning (ML) is widespread. In this field, one of the most popular machine learning model types is tree-based models. As evidence, a recent survey of data scientists and researchers found that tree models were both the second and third most popular class of method, beaten only by Logistic Regression (Figure 1). Although small tree models can be interpretable (Rudin, 2019), most tree models are generally large and hard for humans to interpret.

In order to explain these models, we use Shapley values - a unique game-theoretic solution for spreading credit between features. First, we discuss SHAP values, an extension of Shapley values to machine learning models (Section 2). However, exactly computing SHAP values for an arbitrary model is NP-hard (Matsui and Matsui, 2001), but by focusing on explaining tree models, it is possible to compute them exactly in linear time (Section 3).

*What is the goal of this article?*

Given that there is a long history of model explanations going awry when users do not understand what an explanation means (e.g., p-values for linear models (Schervish, 1996)), it is critical to have a broadly accessible explanation of SHAP values and how they are obtained for tree models to ensure that they are not misused. To this end, we aim to provide an easy to understand answer to two questions: 1.) What are SHAP values?

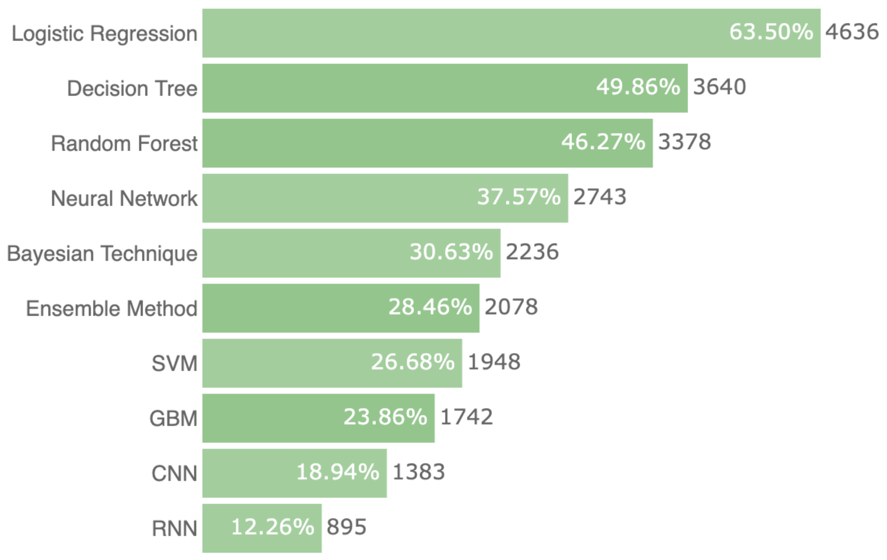


Figure 1: The most popular data science methods according to a 2017 Kaggle survey (based on a total of 7,301 responses).

2.) How can we obtain SHAP values for trees? In particular, we focus on explaining an algorithm to explain trees in the popular SHAP package called Interventional Tree Explainer (the algorithm was first described and empirically evaluated in Lundberg et al. (2020)).

## 2. Defining the feature attribution problem

In this section, we describe Shapley values and a few versions that have been used to explain machine learning models. Then, we use an example with a linear model to justify a specific extension of the Shapley values. With this formulation, we show how obtaining the SHAP values reduces to an average of "single reference SHAP values" that can be thought of as explanations with respect to a single foreground sample (explicand, or sample being explained) and a single background sample (baseline, or sample the explicand is compared to).

### 2.1. Back to basics with Shapley values

Shapley values are a method to spread credit among players in a "coalitional game". We can define the players to be a set $N = \{1, \cdots, d\}$. Then, the coalitional game is a function that maps the power set of the players to a scalar value:

$$v(S) : \mathcal{P}(N) \rightarrow \mathbb{R}^1 \tag{1}$$

To make these concepts more concrete, we can imagine a company that makes a profit $v(S)$ that is determined by what combination of individuals they employ $S$. Furthermore, let's assume we know $v(S)$ for all possible combinations of employees. Then, the Shapley values assign credit to an individual $i$ by taking a weighted average of how much the profit increases when $i$ works with a group $S$ versus when he does not work with $S$. Repeating this for all possible subsets $S$ gives us the Shapley Values:

$$\overbrace{\phi_i(v)}^{\text{Shapley value of } i} = \sum_{S \subseteq N \setminus \{i\}} \underbrace{\frac{|S|!(|N|-|S|-1)!}{|N|!}}_{\text{Weight } W(|S|,|N|)} (\overbrace{v(S \cup \{i\}) - v(S)}^{\text{Profit individual } i \text{ adds}}) \tag{2}$$

Shapley values are an excellent way to give credit to individuals in a coalitional game. In fact, they are known to be a unique solution to spreading credit as defined by several desirable properties[1] (Young, 1985):

- Local Accuracy/Efficiency: The sum of Shapley values for all employees adds up to the profit with all employees minus the profit with no employees:

$$\sum_{i \in N} \phi_i(v) = v(N) - v(\{\}) \tag{3}$$

- Consistency/Monotonicity: If an employee $i$ always increases company $v_1$'s profit more than they would company $v_2$ for all possible teams of employees, then $i$'s attribution for $v_1$ should be greater than or equal to their attribution in $v_2$:

$$v_1(S \cup i) - v_1(S) \geq v_2(S \cup i) - v_2(S) \forall S \implies \phi_i(v_1) \geq \phi_i(v_2) \tag{4}$$

- Missingness: Employees $i$ that don't help or hurt the company's profit must have no attribution.

$$v(S \cup i) = v(S) \forall S \implies \phi_i(v) = 0 \tag{5}$$

## 2.2. Adapting Shapley values to explain machine learning models

Although Shapley values are an optimal solution for allocating credit among players in coalitional games, our goal is actually to allocate credit among features ($x := \{x_1, \cdots, x_d\} \in \mathbb{R}^d$) in a machine learning model ($f(x)$). This begs the question: how exactly do coalitional games differ to machine learning models?

Well, a machine learning model with binary features $x_i \in (0,1)$[2] would in fact be a coalitional game. In this setting, the best way to assign credit to features is to just use the Shapley values where we define a new set function where features are 1 or 0 depending on if they are in the set:

$$v(S) := f(z^S), \text{ where } z_i^S = 1 \text{ if } i \in S, 0 \text{ otherwise} \tag{6}$$

However, most machine learning models actually have continuous features. In order to explain a machine learning model with continuous features, we now have to define both the presence and absence of features[3].

1. Which hold for any coalitional game $v(S)$.
2. Where 1 indicates presence of a feature and 0 indicates absence of a feature.
3. More accurately, we have to define a new set function $v(S)$ related to the model's prediction $f(x)$ where the features in $S$ are present and the remaining features are absent.

First, to address the presence of a feature, our goal is to obtain *local feature attributions*[4]: a vector of importance for each feature of a model prediction for a specific sample $x^f$ (explicand[5]). So if feature $i$ is present, we can simply set that feature to be from the explicand ($x_i^f$). The next step is to address the absence of a feature which can be done in many ways.

## 2.3. Masking missing features

One straightforward way to address the absence of a feature is to define a baseline $x^b$. In this case, if feature $i$ is absent, we can simply set that feature to be $x_i^b$. Then the new set function is:

$$v(S) = f(h^S), \text{ where } h_i^S = x_i^f \text{ if } i \in S,\ x_i^b \text{ otherwise} \tag{7}$$

This specific adaptation of Shapley values to machine learning models is called Baseline Shapley, and has been shown to be a unique solution to attribution methods with a single baseline based on cost-sharing literature (Sundararajan and Najmi, 2019). However, the choice of the baseline is a complicated one, and many different baselines have been considered including an all-zeros baseline, an average across features, a uniformly distributed baseline, and more ((Sundararajan et al., 2017), (Shrikumar et al., 2017), (Fong and Vedaldi, 2017), (Sturmfels et al., 2020))[6].

a.) Model and sample being explained

| | | | |
|---|---|---|---|
| Linear model | $\beta_1 = 2$ | $\beta_2 = -1$ | $\beta_3 = 10$ |
| Explicand | $x_1^f = 70$ | $x_2^f = 135$ | $x_3^f = 0$ |

b.) Zero baseline

| | | | |
|---|---|---|---|
| Baseline | $x_1^b = 0$ | $x_2^b = 0$ | $x_3^b = 0$ |
| Attribution | $\phi_1 = 140$ | $\phi_2 = -135$ | $\phi_3 = 0$ |

c.) Average baseline

| | | | |
|---|---|---|---|
| Baseline | $x_1^b = 70$ | $x_2^b = 135$ | $x_3^b = 0.5$ |
| Attribution | $\phi_1 = 0$ | $\phi_2 = 0$ | $\phi_3 = -5$ |

Figure 2: Example illustrating the downsides of a single baseline for Shapley values. Feature 1 corresponds to age (inches), feature 2 is weight (pounds), and feature 3 is gender (0 is male, 1 is female). The first baseline is an all-zero baseline. The second is an average feature value baseline. See Appendix Section 6.1 for the derivation of the SHAP values for a linear model for masking.

4. As opposed to a *global feature attribution* that aims to assign per-feature importance for the model as a whole.
5. The sample being explained.
6. For an in-depth discussion of this concept and a comparison of several baselines for Aumann-Shapley values applied to explaining images see (Sturmfels et al., 2020).

One criticism of Baseline Shapley is that the choice of baseline is very influential to the resulting feature attributions. In Figure 2, we can see that the choice of baseline heavily influences the resulting attribution value.

Figure 2b is the all-zeros baseline. In this case, it appears that height and weight are quite important, however being male is not important at all. This is a bit counter-intuitive, because relative to being female, being male reduces the prediction for this individual. Put another way, we can consider a new model where $x_3$ is 0 for female, rather than 1 for female. Then, an equivalent model would be $y = 2x_1 - x_2 - 10x_3 + 10$. In this case, using the same zero-baseline gives an attribution value of -10 instead of 0 for being male for the exact same model. Since the meaning of zero is often arbitrary for different variables, selecting it as your baseline can result in misleading attributions.

Alternatively, we could use a mean-baseline as in Figure 2c. Although the mean of a binary variable does not have a natural interpretation it fixes the Gender problem. However, for Height and Weight, we can see that the all-mean baseline introduces bias and effectively forces zero attribution to individuals with average heights and weights. This is undesirable because their height and weight do indeed influence the model's prediction.

To avoid the bias of using a single *background sample* (baseline) we can instead use many different baselines. In the next section we will describe two approaches to incorporate *background distributions* to our set functions $v(S)$ for a much better definition of missingness.

## 2.4. Imputing missing features by conditioning

One natural approach to incorporate a background distribution to the set function is with a conditional expectation. Instead of simply replacing "missing" features with a fixed value, we condition on the set of features that are "present" as if we know them and use those to guess at the "missing" features. If we define $D$ to be the background (underlying) distribution our samples are drawn from, the value of the game is:

$$v(S) = \mathbb{E}_D[f(x)|x_S] \tag{8}$$

One caveat is that getting this conditional expectation for actual data is very difficult. Furthermore, even if you do manage to do so, the resulting explanations can end up having undesirable characteristics (more on this later). Because our goal is just to explain the model itself, an arguably more natural approach is to use causal inference's *interventional conditional expectation*:

$$v(S) = \mathbb{E}_D[f(x)|\text{do}(x_S)] \tag{9}$$

The *do* notation is causal inference's *do*-operator (Pearl, 2009). In words, we break the dependence between the features in $S$ and the remaining features, which is analogous to *intervening* on the remaining features. The motivation behind this decision comes from (Janzing et al., 2019) which is also very close to Random Baseline Shapley in (Sundararajan and Najmi, 2019). Additionally, this is exactly the assumption made by Kernel Explainer (Lundberg and Lee, 2017) and Sampling Explainer from the SHAP package.

In general, computing the conditional expectation SHAP value is difficult; however, for a multivariate normal distribution the conditional expectation is easy to calculate. In addition, for a linear function the conditional expectation of the function equals the function applied to the conditional expectation ($E[f(x)|x_S] = f(E[x|x_S])$[7]).

a.) Independent variables.

$$\beta = \begin{bmatrix}1\\2\\3\end{bmatrix} \quad x^f = \begin{bmatrix}1\\1\\1\end{bmatrix} \quad \Sigma = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} \quad \Longrightarrow \quad \phi^{CE} = \begin{bmatrix}1\\2\\3\end{bmatrix} \quad \phi^{ICE} = \begin{bmatrix}1\\2\\3\end{bmatrix}$$

b.) Pair correlation between $x_2$ and $x_3$.

$$\beta = \begin{bmatrix}1\\2\\3\end{bmatrix} \quad x^f = \begin{bmatrix}1\\1\\1\end{bmatrix} \quad \Sigma = \begin{bmatrix}1 & 0 & 0\\0 & 1 & .99\\0 & .99 & 1\end{bmatrix} \quad \Longrightarrow \quad \phi^{CE} = \begin{bmatrix}1\\2.495\\2.505\end{bmatrix} \quad \phi^{ICE} = \begin{bmatrix}1\\2\\3\end{bmatrix}$$

c.) $x_3$ not used by model (independent variables).

$$\beta = \begin{bmatrix}1\\2\\0\end{bmatrix} \quad x^f = \begin{bmatrix}1\\1\\1\end{bmatrix} \quad \Sigma = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} \quad \Longrightarrow \quad \phi^{CE} = \begin{bmatrix}1\\2\\0\end{bmatrix} \quad \phi^{ICE} = \begin{bmatrix}1\\2\\0\end{bmatrix}$$

d.) $x_3$ not used by model (pair correlation).

$$\beta = \begin{bmatrix}1\\2\\0\end{bmatrix} \quad x^f = \begin{bmatrix}1\\1\\1\end{bmatrix} \quad \Sigma = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0.99\\0 & 0.99 & 1\end{bmatrix} \quad \Longrightarrow \quad \phi^{CE} = \begin{bmatrix}1\\1.01\\0.99\end{bmatrix} \quad \phi^{ICE} = \begin{bmatrix}1\\2\\0\end{bmatrix}$$

Figure 3: Comparing two versions of SHAP values: conditional expectation ($\phi^{CE}$) and interventional conditional expectation ($\phi^{ICE}$). We make two simplifying assumptions: 1.) The function is linear ($y = \beta^T x$) and 2.) the data is multivariate normally distributed ($x \sim \mathcal{N}_3(0, \Sigma)$). See Appendix Section 6.2 for the derivation of the SHAP values for ICE and CE.

In Figure 3, we highlight tradeoffs between the conditional expectation and the interventional conditional expectation for a linear model. Comparing 3a to 3b, we can see that the correlation between variables will cause the CE SHAP values to be split between correlated variables. Although this may be desirable at times[8], it can feel somewhat unnatural as in 3d, where feature $x_4$ is as important as feature $x_3$ despite not being used in the model.

Although both CE SHAP values and ICE SHAP values are meaningful, we will focus on ICE SHAP values moving forward because they are tractable to compute and explain what the model is doing, rather than how features in the data relate to each other.

## 2.5. Single reference SHAP values

To compute $\phi_i(f, x^f)$ we evaluate the interventional conditional expectation. However, this depends on a *background distribution* $D$ that the foreground sample will be compared

7. Note that this is not generally true. Only for affine functions.
8. For example, if we are interested in detecting whether a model is secretly using a confounding variable, we may want CE SHAP values.

to. One natural definition of the background distribution is a uniform distribution over a population sample. For instance, in machine learning, you could assign equal probability to every sample in your training set. With this background distribution, we can re-write the SHAP value as an average of *single reference SHAP values* (which are analogous to masking missing features Section 2.3) (Chen et al., 2019):

$$\phi_i(f, x^f) = \frac{1}{|D|} \sum_{x^b \in D} \phi_i(f, x^f, x^b) \tag{10}$$

This is in part because the interventional conditional expectation has a very natural definition when the background distribution is a single sample $x^b$ ($D_{x^b}$):

$$\mathbb{E}_{D_{x^b}}[f(x^f)|\text{do}(x_S)] = h^S, \text{ where } h_i^S = x_i^f \text{ if } i \in S,\ x_i^b \text{ otherwise} \tag{11}$$

In words, the interventional conditional expectation for a single baseline is the model prediction for a hybrid sample where the features in $S$ are from the explicand and the remaining features are from baseline. This is as if *we intervene on features in the foreground sample with features from the background sample.* This should look very familiar, as it is equivalent to obtaining the SHAP value by treating a sample in the background distribution as a baseline and masking as in Section 2.3).

## 3. An algorithm to explain trees quickly and exactly

Now our goal is to tackle the simpler problem of obtaining single reference SHAP values $\phi_i(f, x^f, x^b)$ that are attributions for a single foreground sample and background sample.

### 3.1. Brute force

Based on the proof in Section 2.5, the brute force approach would be to compute the following:

$$\phi_i(f, x^f, x^b) = \sum_{S \subseteq N \setminus \{i\}} \underbrace{W(|S|, |N|)}_{W} \underbrace{f(h^{S\cup\{i\}})}_{\text{Pos term}} - \underbrace{f(h^S)}_{\text{Neg term}} )$$

The algorithm is then fairly straightforward:

```
def brute_force(array xf, array xb, tree T, array N):
    phi = [0]*len(xf)
    for each feature i in N:
        for each set S in powerset(setminus(N,i)):
        hs  = [xf[j] for j in N if j in S else xb[j]] # Hybrid samples
        hsi = [xf[j] for j in N if j in union(S,i) else xb[j]]
        fxs  = T.predict(hs) # Predictions
        fxsi = T.predict(hsi)
        W = (len(S)!*(len(N)-len(S)-1)!)/(len(N)!) # Weight
        phi[i] += W*(fxsi-fxs) # Calculate phi contribution
```

If we assume the computational cost of computing the weight $W$ is constant[9], then the complexity of the brute force method is the number of terms in the summation multiplied by the cost of making a prediction (on the order of the depth of the tree). This gives $O((\text{tree depth}) \times 2^d)$.

Then, in order to compute $\phi_i(f, x^f, x^b)$ for all features $i$, we have to re-run the entire algorithm $d$ times, giving us an overall complexity of

$$O(d \times (\text{tree depth}) \times 2^d) \tag{12}$$

An exponential computational complexity is bad; however, if we *constrain $f(x)$ to be a tree-based model* (e.g., XGBoost, decision trees, random forests, etc.), then we can come up with a polynomial time algorithm to compute $\phi_i(f, x^f, x^b)$ exactly. Why is this the case? Well, even for explaining a single feature, the brute force algorithm may consider a particular path multiple times. However, to compute the SHAP value for a single feature, it turns out that we only need to consider each path once. This insight leads us to the naive algorithm in the following Section (3.2).

## 3.2. Naive Algorithm

Before we get into the algorithm, we first describe a theorem that is the basis for this naive implementation.

**Theorem 1** *To calculate $\phi_i(f, x^f, x^b)$, we can calculate attributions for each path from the root to each leaf. For a given path $P$, we define $N_P$ to be the unique features in the path and $S_P$ to be the unique features in the path that came from $x^f$. Finally, define $v$ to be the value of the path's leaf. Then, the attribution of the path is:*

$$\phi_i^P(f, x^f, x^b) = \begin{cases} 0 & \text{if } i \notin N_P \\ W(|S_P| - 1, |N_P|) \times v & \text{if } i \in S_P \\ -W(|S_P|, |N_P|) \times v & o.w. \end{cases} \tag{13}$$

**Proof** (Sketch) Treat each path $P$ in the tree from the root to each leaf as a separate model $f^P(x)$ that returns the value of the leaf if that path is traversed by $x$ or zero otherwise. This results in (# leaf nodes) models that operate on disjoint parts of the input space, implying that our original model is equal to the summation of all of these path models:

$$f(x) = \sum_P f^P(x) \tag{14}$$

By the additivity of SHAP values,

$$\phi_i(f, x^f, x^b) = \sum_P \phi_i(f^P, x^f, x^b) \tag{15}$$

Then, we can simply calculate $\phi_i$ for each path model. Since the path model is zero everywhere except for the associated path, we arrive to the solution in Theorem 1. ■

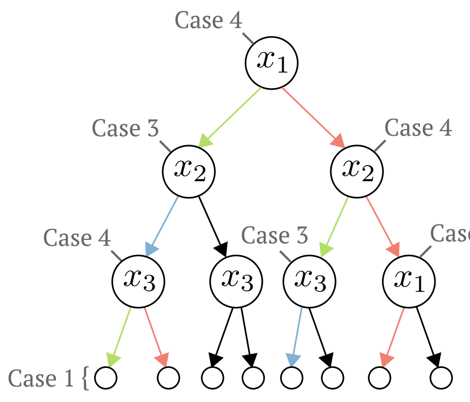


Figure 4: Example to illustrate cases for calculating the attribution $\phi_i(f,x)$ by growing the lists $S_P$ and $N_P$. Green paths are associated with $x^f$ and red paths are $x^b$.

Then the goal of the naive algorithm is to obtain $N_P$ and $S_P$ for each path by recursively traversing the tree. We will start by explaining the algorithm via an example:

In the naive algorithm, we maintain lists $N_P$ and $S_P$ as we traverse the tree. At each internal node (Cases 2-4) we update the lists and then pass them to the node's children. At the leaf nodes (Case 1), we calculate the attribution for each path. In Figure 4, we see four possible cases:

- Case 1: $n$ is a leaf. Return the attribution in Theorem 1 based on $N_P$ and $S_P$.
- Case 2: The feature has been encountered already ($n_{feature} \in N_P$). Depending on if we split on $x^f$ or $x^b$, we compare either $x^f_{n_{feature}}$ or $x^b_{n_{feature}}$ to $n_{threshold}$ and go down the appropriate child. Pass down $N_P$ and $S_P$ without modifications because we did not add a new feature.
- Case 3: Both $x^f$ and $x^b$ are on the same side of $n$'s split. Pass down $N_P$ and $S_P$ without modifications because relative to $x^f$ and $x^b$ it's as if this node doesn't exist.
- Case 4: $x^f$ and $x^b$ go to different children. Add $n_{feature}$ to both $N_P$ and $S_P$ and pass both lists to the $x^f$ child. Only add $n_{feature}$ to $N_P$ and pass both lists to the $x^b$ child.

Then the recursive algorithm is:

```
def naive(array xf, array xb, tree T):
    phi = [0]*len(xf)
    def recurse(node n, list np, list sp):
        # Case 1: Leaf
        if n.is_leaf: [Theorem 1]
            for i in N:
            if i in sp:
                phi[i] += W(len(sp)−1,len(np))
```

9. Which it will be if we memoize $k!$ for $k = 1, \cdots, d$.

```
        elif:
            phi[i] -= W(len(sp),len(np))
    # Find children associated with xf and xb
    xf_child = n.left if xf[n.feat] < n.thres else n.right
    xb_child = n.left if xb[n.feat] < n.thres else n.right
    // Case 2: Feature encountered before
    if n.feat in np:
        if n.feat in sp:
            return(recurse(xf_child,np,sp))
        else:
            return(recurse(xb_child,np,sp))
    # Case 3: xf and xb go the same way
    if xf_child == xb_child:
        return(recurse(xf_child,np,sp))
    # Case 4: xf and xb don't go the same way
    if not xf_child != xb_child:
        f_phi = recurse(xf_child,np+[n.feat],sp+[n.feat])
        b_phi = recurse(xb_child,np+[n.feat],sp)
    recurse(n=T.root,sp=[],np=[])
```

Then, we examine the computational complexity to compute $\phi_i(f, x^f, x^b)$ for all features $i$ using the naive algorithm.

First of all, the worst case complexity for each internal node is based on Cases 2-4. In the worst case, we need to check whether the current node's feature has been encountered previously by iterating through $S_P$ and $N_P$. Since these lists are of length (tree depth) in the worst case, each node incurs a linear $O$(tree depth) cost.

Next, for the leaf nodes, we compute the contributions for each feature (of which there are $d$). Then, computing the contributions for each feature requires checking whether the feature is in $S_P$. This means that each leaf node incurs a cost of $O((\text{tree depth}) \times d)$ cost.

Putting this together, we get an overall cost of:

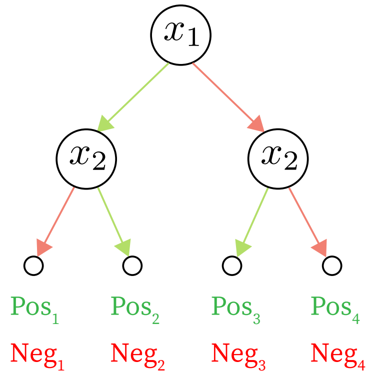
$$O((\#\text{ internal nodes}) \times (\text{tree depth})) + O((\#\text{ leaf nodes}) \times (\text{tree depth}) \times d)$$

In the next section we present two observations that allow us to compute $\phi_i(f, x^f, x^b)$ for all features $i$ in $O(\#\text{ nodes})$ time.

## 3.3. Dynamic Programming Algorithm

To improve the computational complexity of the straightforward naive algorithm, we can make two observations.

The first observation is that we can actually get rid of the multiplicative (tree depth) factor by focusing on what the lists $S_P$ and $N_P$ are used for. The first use is to check if a feature has been previously encountered. By replacing the lists with boolean arrays, we can check if a given feature has been encountered by a constant time access into the arrays. This means the internal nodes now incur a constant cost. The second use of $S_P$ and $N_P$ is to calculate the sizes of $|S|$ and $|N|$ at the leaves. By instead maintaining integers that



| | |
|---|---|
| $\phi_1(f, x^f, x^b)$ | $\text{Pos}_1 + \text{Pos}_2 + \text{Neg}_3 + \text{Neg}_4$ |
| $\phi_2(f, x^f, x^b)$ | $\text{Neg}_1 + \text{Pos}_2 + \text{Pos}_3 + \text{Neg}_4$ |

Figure 5: Example to illustrate collapsibility for features. Green paths are associated with $x^f$ and red paths are $x^b$.

keep track of these values, the leaves no longer have to iterate through lists. This gets rid of the multiplicative (tree depth) factor. Then, the new computational complexity is:

$$O((\#\text{ internal nodes}) + O((\#\text{ leaf nodes}) \times d) \tag{16}$$

The second observation is that we can compute the attributions for all features simultaneously as we traverse the tree by passing Pos and Neg attributions to parent nodes. Before describing the algorithm in more detail, we first present a figure that illustrates why passing up the attributions is sufficient.

We can first observe that for each leaf, according to Theorem 1, there are only two possible values needed to compute the SHAP values (Pos and Neg). Then, in Figure 5, based on the attributions for $x_1$ we see that these Pos and Neg terms can be grouped by the left and right subtrees below $x_1$. To generalize, we make the following observation:

**Observation** *In order to compute the attribution for any feature $i$ it is sufficient to consider the paths that correspond to each Case 4 node's children.*

*This means we can focus on Case 4 nodes $n$, where we know that one child is associated with $x^f$ child and one child is associated with $x^b$. Then, the attribution to the node's feature is:*

$$\sum_{\text{paths } P \text{ under } x^f \text{ child}} \text{Pos}_P + \sum_{\text{paths } P \text{ under } x^b \text{ child}} \text{Neg}_P \tag{17}$$

*For Case 2 and 3 nodes, there is no need to do anything, because it is as if they are not in the model.*

This observation suggests that we can always add the Pos and Neg terms at a given node and pass them up to the parent. This information is sufficient to calculate the attributions for each upstream feature. This aggregation of the Pos and Neg terms is the dynamic programming observation that allows each upstream node to only need a constant number of operations to compute its feature's attribution.

Using this observation, we devise an algorithm that computes the attributions for all features simultaneously (getting rid of that final multiplicative $d$ factor for the leaf nodes):

```
def dynamic(array xf, array xb, tree T):
    phi = [0]*len(xf)
    def recurse(node n, int nc, int sc, array fseen, array bseen):
        # Case 1: Leaf
        if n.is_leaf:
            if sc == 0: return((0,0))
            else: return((n.value*W(sc,nc-1),-n.value*W(sc,nc)))
        # Find children associated with xf and xb
        xf_child = n.left if xf[n.feat] < n.thres else n.right
        xb_child = n.left if xb[n.feat] < n.thres else n.right
        # Case 2: Feature encountered before
        if fseen[n.feat] > 0:
            return(recurse(xf_child,nc,sc,fseen,bseen))
        if bseen[n.feat] > 0:
            return(recurse(xb_child,nc,sc,fseen,bseen))
        # Case 3: xf and xb go the same way
        if xf_child == xb_child:
            return(recurse(xb_child,nc,sc,fseen,bseen))
        # Case 4: xf and xb don't go the same way
        if xf_child != xb_child:
            fseen[n.feat] += 1
            posf,negf = recurse(xf_child,nc+1,sc+1,fseen,bseen)
            fseen[n.feat] -= 1; bseen[n.feat] += 1
            posb,negb = recurse(xb_child,nc+1,sc  ,fseen,bseen)
            bseen[n.feat] -= 1
            phi[n.feat] += posf+negb
            return((posf+posb,negf+negb))
        recurse(n=T.root,0,0,[0]*len(xf),[0]*len(xf))
```

Now each node now only requires a constant amount of work. The computational complexity to compute $\phi_i(f, x^f, x^b)$ for all features $i$ with this algorithm is:

$$O(\#\text{ nodes}) \tag{18}$$

## 3.4. Final considerations

**Background distribution:** Finally, our original goal was to compute $\phi_i(f, x^f)$. In order to do so, we compute $\phi_i(f, x^f, x^b)$ for many references $x^b$, resulting in a run time of:

$$O(|D| \times (\#\text{ nodes})) \tag{19}$$

where $|D|$ is the number of samples in the background distribution. In practice, using a fixed number of about 100 to 1000 references works well.

**Further optimization:** Explaining the tree for a specific foreground and background sample requires knowing where all hybrid samples go in the tree. Therefore we achieve the best possible complexity of $O(\#\text{ nodes})$. However, it may be possible to compute the attributions for many background samples simultaneously in sub-linear time in order to reduce the $O(|D| \times (\#\text{ nodes}))$ cost.

**Ensembles of trees:** Many tree based methods are ensembles (e.g., random forests, gradient boosting trees). In order to compute that attributions for these types of models, we can simply leverage the additivity of SHAP values and explain each tree and sum the attributions for each tree. This means that to explain a gradient boosting tree model, the computational complexity is:

$$O((\#\text{ trees}) \times |D| \times (\#\text{ nodes})) \tag{20}$$

## 4. Related Work

### 4.1. Methods in the SHAP package

It should be noted that there are a number of alternative methods that aim to approximate SHAP values. A few of these methods include: *Sampling Explainer*, *Kernel Explainer*, and *Path Dependent Tree Explainer*. If you are explaining tree-based models, it may not be clear which one you should use. In this article we briefly overview a few of these the methods and compare them to ITE.

First, there are two model agnostic explanation methods in the SHAP package. The first is *Sampling Explainer* which is an implementation of Interactions-based Method for Explanation (IME) (Kononenko et al., 2010). This approach is based on sampling from all possible sets in order to estimate ICE SHAP values. The second is *Kernel Explainer* which is an extension of Local Interpretable Model-agnostic Explanations (LIME) (Ribeiro et al., 2016) to estimate ICE SHAP values. Both *Sampling Explainer* and *Kernel Explainer* are sampling based approaches that will converge to the same SHAP values ITE obtains. However, ITE is much faster in practice because it leverages the structure of the tree.

Second, in the SHAP package there is a method named *Path Dependent Tree Explainer (PDTE)* that is meant to obtain SHAP values for tree models specifically. *PDTE* approximates the interventional conditional expectation based on how many training samples went down paths in the tree, whereas ITE computes it exactly. The computational complexity of *PDTE* is $O((\#\text{ leaf nodes}) \times (\text{tree depth})^2)$. In practice, *PDTE* can be faster than ITE, although it may depend on the number of references or the tree depth. The tradeoff is that the attribution values estimated by *PDTE* are biased and do not give the ICE SHAP values.

### 4.2. A few other methods to explain trees

Two pre-existing methods to explain trees include *Gain (Gini Importance)* (Breiman et al., 1984) and *Saabas* (Saabas). Both methods consider a single ordering of the features (as opposed to all possible orderings) specified by the tree they are explaining. Note that for an infinite ensemble of fully developed totally randomized trees[10] and an infinitely large training

10. Where a fully developed totally randomized tree is a decision tree where each node $n$ is partitioned using a variable $x_i$ picked uniformly at random among those not yet used upstream of $n$, each node $n$ has one child for each possible value of $x_i$, and the construction of these trees is complete only when all variables have been used.

sample, *Gain* is the mutual information between covariates and the outcome (Louppe, 2014) and Saabas gives the SHAP values (Lundberg et al., 2020). However, in practice trees are constructed greedily and both methods fail to satisfy the consistency axiom because they only consider a single ordering of the features.

### 4.3. Empirical Evaluation

In this article we do not empirically evaluate these methods, because evaluating interpretability methods can difficult due to the subjective nature of an explanation. Instead, we identify a way of assigning credit that comes equipped with a set of desirable properties (i.e., Shapley values) and show how to compute them exactly for trees with a tractable algorithm. This means that the credit allocation we obtain comes built in with many of the desirable properties that ablation tests aim to measure (e.g., consistency). That being said, Lundberg et al. (2020) provides an in-depth empirical comparison of many of these methods using a variety of ablation tests.

## 5. Conclusion

In this paper, we presented an end to end description of an algorithm we developed and previously experimentally validated in Lundberg et al. (2020). We described Shapley values and the ways to define them for machine learning methods and analyzed a few tradeoffs for linear models. Finally, we moved on to discuss their computation for trees moving from exponential time to a low-order polynomial time algorithm.

## References

Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. Classification and regression trees, 1984.

Hugh Chen, Scott Lundberg, and Su-In Lee. Explaining models by propagating shapley values of local components. *arXiv preprint arXiv:1911.11888*, 2019.

Ruth C Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3429–3437, 2017.

Dominik Janzing, Lenon Minorics, and Patrick Blöbaum. Feature relevance quantification in explainable ai: A causality problem. *arXiv preprint arXiv:1910.13413*, 2019.

Igor Kononenko et al. An efficient explanation of individual classifications using game theory. *Journal of Machine Learning Research*, 11(Jan):1–18, 2010.

Gilles Louppe. Understanding random forests: From theory to practice. *arXiv preprint arXiv:1407.7502*, 2014.

Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in neural information processing systems*, pages 4765–4774, 2017.

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(1): 56–67, 2020.

Yasuko Matsui and Tomomi Matsui. Np-completeness for calculating power indices of weighted majority games. *Theoretical Computer Science*, 263(1-2):305–310, 2001.

Judea Pearl. *Causality*. Cambridge university press, 2009.

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.

Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, 2019.

Ando Saabas. andosa/treeinterpreter. URL https://github.com/andosa/treeinterpreter.

Mark J Schervish. P values: what they are and what they are not. *The American Statistician*, 50(3):203–206, 1996.

Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3145–3153. JMLR. org, 2017.

Pascal Sturmfels, Scott Lundberg, and Su-In Lee. Visualizing the impact of feature attribution baselines. *Distill*, 5(1):e22, 2020.

Mukund Sundararajan and Amir Najmi. The many shapley values for model explanation. *arXiv preprint arXiv:1908.08474*, 2019.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3319–3328. JMLR. org, 2017.

H Peyton Young. Monotonic solutions of cooperative games. *International Journal of Game Theory*, 14(2):65–72, 1985.

## 6. Appendix

### 6.1. Masking missing features

**Proof** For masking (and Figure 2), the SHAP values for a linear model are easy to compute:

$$\phi_i(f, x^f, x^b) = \sum_{S \in C} W(|S|, |N|)(f(h^{S \cup i}) - f(h^S)) \tag{21}$$

$$= \sum_{S \in C} W(|S|, |N|)(\beta h^{S \cup i} - \beta h^S) \tag{22}$$

$$= \sum_{S \in C} W(|S|, |N|)\beta_i(x_i^f - x_i^b) \tag{23}$$

$$= \beta_i(x_i^f - x_i^b) \tag{24}$$

Note that $\sum_{S \in C} W(|S|, |N|) = 1$. ■

### 6.2. ICE and CE

**Proof** Computing SHAP values for a linear model is much easier than for other model classes [11]. This is how we compute them for Figure 3.

First, to compute Interventional Conditional Expectation (ICE) SHAP values for a linear model, we can start with:

$$\phi_i(f, x) = \sum_{S \in C} W(|S|, |N|)(\mathbb{E}_D[f(x)|do(x_{S \cup \{i\}})] - \mathbb{E}_D[f(x)|do(x_S)]) \tag{25}$$

$$= \sum_{S \in C} W(|S|, |N|)(\mathbb{E}_D[\beta x|do(x_{S \cup \{i\}})] - \mathbb{E}_D[\beta x|do(x_S)]) \tag{26}$$

$$= \sum_{S \in C} W(|S|, |N|)\beta(\mathbb{E}_D[x|do(x_{S \cup \{i\}})] - \mathbb{E}_D[x|do(x_S)]) \tag{27}$$

$$= \sum_{S \in C} W(|S|, |N|)\beta_i x_i^f \tag{28}$$

$$= \beta_i x_i^f \tag{29}$$

Note that $\sum_{S \in C} W(|S|, |N|) = 1$. ■

**Proof** Second, to compute Conditional Expectation (CE) SHAP values for a linear model, we assume the background distribution $D$ is multivariate normal with zero mean and covariance $C$, we can start with:

$$\phi_i(f, x) = \sum_{S \in C} W(|S|, |N|)(\mathbb{E}_D[f(x)|x_{S \cup \{i\}}] - \mathbb{E}_D[f(x)|x_S]) \tag{30}$$

$$= \sum_{S \in C} W(|S|, |N|)(\mathbb{E}_D[\beta x|x_{S \cup \{i\}}] - \mathbb{E}_D[\beta x|x_S]) \tag{31}$$

$$= \sum_{S \in C} W(|S|, |N|)\beta(\mathbb{E}_D[x|x_{S \cup \{i\}}] - \mathbb{E}_D[x|x_S]) \tag{32}$$

11. Note that we assume features have zero mean in the following calculations.

Here, we know that $\mathbb{E}_D[x|x_S] = C_{N\setminus S,S}C_{S,S}^{-1}x_S$. ■