Understanding TreeSHAP - A complete tutorial

Keryann MASSIN

December 5, 2023

Abstract

With the development of machine learning techniques, predictive algorithms have become more and more complex, to the point where their outputs are often unintelligible for unaware users. This observation led to the development of interpretability theory, of which SHAP values stand as one of the most powerful tools. The literature is already flourishing on SHAP values but the explanations it provides on some of its main concepts and algorithms are a bit abstruse. In this article, we will not introduce new theoretical content but will focus our efforts on explaining what are SHAP values, and how can one compute them in the specific case of tree-based models using TreeSHAP (whose in-depth mathematical basements are recalled here). We make the distinction between an interventional and an observational approach and develop the latter.

The reader may find the notebook that implements the algorithm discussed here on github, at: https://github.com/Noomkwah/UnderstandingTreeSHAP

Contents

1 Introduction

Artifical intelligence has become intensively used in a variety of fields, and understanding the machine learning model's behaviors behind it can often be crucial to the business. Imagine the following situation: you are a data scientist and your last big task was to help doctors at your local hospital classify new patients according to their risk of heart attack, in order to better manage the hospital resources. You have completed your task successfully, providing an efficient AI-powered classifier, but the head doctor would like you to explain how the predictions are made. You cannot just say 'go and do my studies', so how do you meet his demand? This situation is an example of a case where being able to *interpret* a model's prediction can be useful.

The best explanation of a model is the model itself. However, for complex models, it can be interesting to obtain simpler explanations. Let f be the prediction function of a model (that we want to explain) and g be the prediction function of the explanatory model. We will focus on *local models*, which explain $f(x)$ given x. The goal is not to explain f in all generality, but rather to explain how the different coordinates of x (the features) have influenced the calculation of $f(x)$. For example, saying 'weight is an important explanatory factor for the risk of heart attack' is not a local explanation (it is global), but saying 'for Mr. Bertrand, a weight of 93kg contributed to increasing his risk of cardiac arrest by 50%' is a local explanation.

This paper is meant to be a tutorial for neophytes who wish to interpret their models outputs. In section 2 we define SHAP values [1] and see why they stand as a good candidate for explanation purposes. Section 3 is then dedicaded to explaining in full details how they may be computed using the famous TreeSHAP algorithm [2] in the context of a tree-based model. The reader may find all the proofs of cited theorems and propositions in annexe A. The whole discussion below is backed by the example of heart attack probability prediction mentioned above. We assume the predictive model we wish to explain is the decision tree of figure 1, that takes $x =$ (age, weight, height, bodyfat percentage) as an input and outputs $f(x)$, a probability of heart attack.

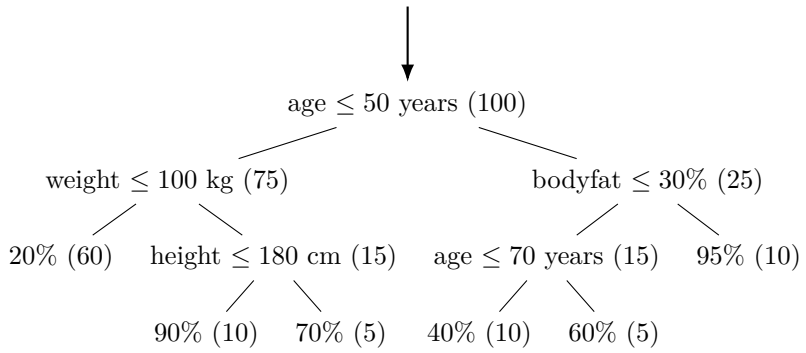

Figure 1: Decision tree with 4 features

2 First Considerations

2.1 Mathematical Formalism

Most machine/deep learning algorithms take in fairly complex data (high-dimensional vectors, images, etc.). Hence, we introduce the notion of a *simplified input*, denoted as x', such that $x = h_x(x')$ for a certain function h_x (which is relative to x). For instance, if x represents the vector containing Mr. Bertrand's age, weight, name, and gender ($x = (56, 93, \text{Gérard}, \text{M})$), we could have $x' = (1, 1, 1, 1)$ and h_x mapping $z' \in \{0, 1\}^4$ to $h_x(z') = z$, where z contains only the coordinates of the features for which z' had a coordinate of 1. Thus: $h_x((1, 0, 1, 0)) = (56, \text{Gérard})$. The goal of local methods is to ensure that if $z' \approx x'$, then $g(z') \approx f(h_x(z'))$.

Definition 1. An **additive feature attribution method** presents an explanatory prediction function g that is linear and a function of binary variables:

$$g(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z'_i \tag{1}$$

where $z' \in \{0, 1\}^M$, and M is the number of features in the simplified input z', and $\phi_i \in \mathbb{R}$.

Definition 2. The **Shapley values** characterize the importance of each feature for linear models in the presence of multicollinearity. To calculate them, the idea is to train a model f_S on a subset $S \subseteq F$ of features, then train a new model $f_{S\cup\{i\}}$ on $S \cup \{i\}$. The difference $f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S)$ gives the importance of feature i for the prediction $f(x)$ in configuration S. To obtain the overall importance of feature i, a weighted average is taken over all configurations:

$$\phi_i(f, x) = \sum_{S\subseteq F\setminus\{i\}} \frac{|S|!(|F| - |S| - 1)!}{|F|!} \cdot \left[f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S)\right] \tag{2}$$

Here, h_x maps 1 and 0 to the original space, where 1 indicates inclusion of the feature in the model, and 0 indicates absence. Note $\phi_0 = f_\emptyset(\emptyset)$ as the 'baseline' value provided by the model in the absence of information.

Intuition To understand the weight $\frac{|S|!(|F|-|S|-1)!}{|F|!}$, consider the problem differently. Imagine the following scenario: our M features are waiting outside the 'door' of the model. Each time a feature goes through the door, the model's prediction changes based on the information obtained from this new feature. The marginal contribution of a new feature i to a set of features S is $f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S)$, and this contribution is invariant to the order in which the $|S|$ features of S are considered and the order in which the remaining $(|F| - |S| - 1)$ features are considered afterward. There are $|S|!(|F| - |S| - 1)!$ such orders. So ultimately, denoting Π as the set of permutations of the $|F|$ features, we have:

$$\begin{aligned}
\phi_i(f, x) &= \frac{1}{|\Pi|} \sum_{\pi\in\Pi} f_{\text{before}(\pi,i)\cup\{i\}}(x_{\text{before}(\pi,i)\cup\{i\}}) - f_{\text{before}(\pi,i)}(x_{\text{before}(\pi,i)}) \\
&= \frac{1}{|F|!} \sum_{\pi\in\Pi} \sum_{S=\text{before}(\pi,i)} \left[f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S)\right] \\
&= \sum_{S\subseteq F\setminus\{i\}} \frac{|S|!(|F| - |S| - 1)!}{|F|!} \cdot \left[f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S)\right]
\end{aligned}$$

where $\text{before}(\pi, i)$ corresponds to the subset of features appearing before i in permutation π.

It turns out that the class of additive feature attribution methods includes only one solution possessing 'good properties', known as the *SHAP value* (Shapley Additive exPlanation in English). These three good properties are as follows:

Property 1. (Local accuracy)

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^{M} \phi_i x'_i \tag{3}$$

For $x = h_x(x')$, the explanatory model is equal to the initial model.

Property 2. (Missingness)

$$x'_i = 0 \quad \Rightarrow \quad \phi_i = 0 \tag{4}$$

Missing features should have no impact on the prediction.

Property 3. (Consistency) Let $f_x(z') = f(h_x(z'))$, and let $z'\backslash i$ denote setting $z'_i = 0$. If f and f' are two models, and

$$f'_x(z') - f'_x(z'\backslash i) \geq f_x(z') - f_x(z'\backslash i) \tag{5}$$

for every simplified input z', then $\phi_i(f', x) \geq \phi_i(f, x)$. In other words, if the contribution of feature i is always greater for model f' than it is for model f, then the SHAP value of i will be greater in model f' than in model f.

These properties are, upon reflection, entirely reasonable: transitioning from f to g significantly simplifies the model (which was the intention), yet one would still desire to maintain a certain level of accuracy (Property 1); assigning non-zero importance to features that aren't even present is unwarranted (Property 2); obtaining a measure of importance that is comparable across models is a significant advantage (Property 3).

Theorem 1. There exists only one method of additive feature attribution g that satisfies Properties 1, 2, and 3. For this g, we have:

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{(|z'| - 1)!(M - |z'|)!}{M!} \Big(f_x(z') - f_x(z'\backslash i) \Big) \tag{6}$$

where $z' \subseteq x'$ means $(\forall i,\ x'_i = 1 \ \Rightarrow \ z'_i = 1)$.

Note. It's worth mentioning that the weight in Equation 6 differs from that in Equation 2. This stems from the following observation: in Equation 6, we remove a feature i from the set z', while we add this feature to S in Equation 2. In the end, both definitions are equivalent.

2.2 Observational Expectation versus Interventional Expectation

Most machine learning models do not handle missing values in their input data. Thus, estimating $f_x(z')$ when $z' \neq x'$ (for instance, giving $(56, \text{Gérard})$ instead of $(56, 93, \text{Gérard}, \text{M})$ to our algorithm) becomes challenging, leading to potential errors. To resolve this issue, a commonly used method involves retaining a reference value for each feature and filling in missing data with the reference value. This baseline is denoted as x^b, and typically, setting $x^b = (45, 70, \text{Julie}, \text{F})$, we would have:

$$(56, \text{Gérard}) \rightarrow (56, 70, \text{Gérard}, \text{F}) \tag{7}$$

Formally, this equates to considering $h_x^{x^b}$ such that:

$$h_x^{x^b}(z') = z \quad \text{with} \quad z_i = \begin{cases} x_i & \text{if } z'_i = 1 \\ x_i^b & \text{otherwise} \end{cases} \tag{8}$$

Initially, there is no specific reason to choose a particular baseline x^b, making the final SHAP values significantly dependent on the chosen x^b. In fact, an ideal approach would consider h_x as follows:

$$f_x(z') = f(h_x(z')) = \mathbb{E}[f(z) \mid z_S] \tag{9}$$

where S is the set of non-zero coefficients in z, i.e., the set of i such that $z'_i = 1$. The expectation here is taken over the prior distribution of z, encompassing all features. In other words, we compute:

$$f_x(z') = \mathbb{E}[f(z) \mid z_S] = \int f(z) d\mu_{z_{\overline{S}}|z_S} \tag{10}$$

In practice, we rarely know the underlying distribution of our data (which is why in machine learning, we train a model that is supposed to 'learn' this distribution). Therefore, obtaining the

quantity defined earlier accurately is challenging. However, we can approximate it using our data. The idea is to take several different x^b directly from our training database (the entire database or a subset D) and calculate:

$$\frac{1}{|D|}\sum_{x^b \in D} f\left(h_x^{x^b}(z')\right) \underset{|D|\to\infty}{\to} \int f(z)d\mu \tag{11}$$

By doing so, we 'break' the potential dependencies between the features of $\overline{S}$ and S, hence the notation $d\mu \neq d\mu_{z_{\overline{S}}|z_S}$. Generally, we denote $\mathbb{E}[f(z) \mid \text{do}(z_S)]$ instead of $\mathbb{E}[f(z) \mid z_S]$ for this kind of 'breaking'. We refer to observational expectation for $\mathbb{E}[f(z) \mid z_S]$ and interventional expectation for $\mathbb{E}[f(z) \mid \text{do}(z_S)]$. To better understand the difference between the two, let us take an example. Suppose we want to calculate $f_x((56, \text{Gérard}))$.

Observational Expectation We aim to compute:

$$f_x((56, \text{Gérard})) = \mathbb{E}[f(z) \mid z_{\text{age}} = 56, z_{\text{name}} = \text{Gérard}] \tag{12}$$

This translates to:

$$\begin{aligned} f_x((56, \text{Gérard})) &= \mathbb{E}[f(56, z_{\text{weight}}, \text{Gérard}, z_{\text{gender}}) \mid z_{\text{age}} = 56, z_{\text{prénom}} = \text{Gérard}] \\ &= \mathbb{E}[f(56, z_{\text{weight}}, \text{Gérard}, \text{M}) \mid z_{\text{age}} = 56, z_{\text{name}} = \text{Gérard}] \end{aligned}$$

because gender is not independent of the first name. In fact, the probability of being male when named Gérard is close to (let us say equal to) 1. Similarly, weight is not independent of the first name, as men tend to weigh more than women on average, and the first name is not independent of gender. Weight is also correlated with age: an infant weighs less than an adult. Denoting $\mathbb{Q}$ as the distribution of z_{weight} given $z_{\text{age}} = 56, z_{\text{name}} = \text{Gérard}$, we finally get:

$$f_x((56, \text{Gérard})) = \int f(56, z_{\text{weight}}, \text{Gérard}, \text{M}), \text{d}\mathbb{Q} \tag{13}$$

Interventional Expectation Now, we aim to calculate:

$$f_x((56, \text{Gérard})) = \mathbb{E}[f(z) \mid \text{do}(z_{\text{age}} = 56, z_{\text{name}} = \text{Gérard})] \tag{14}$$

Here, we disregard the previously mentioned dependencies. Everything happens as if we had 'forced' the values of z_{age} and z_{name} and removed dependencies on other variables. Therefore, we can independently calculate integrals as follows:

$$\begin{aligned} f_x((56, \text{Gérard})) = \mathbb{E}[f(z) \mid \text{do}(z_S)] &= \int f((56, a, \text{Gérard}, b))da\, db \\ &= \mathbb{P}(b = \text{M}) \int f((56, a, \text{Gérard}, \text{M}))da + \mathbb{P}(b = \text{F}) \int f((56, a, \text{Gérard}, \text{F}))da \\ &= \int \mathbb{P}(b = \text{M}) f((56, z_{\text{weight}}, \text{Gérard}, \text{M})) + \mathbb{P}(b = \text{F}) f((56, z_{\text{weight}}, \text{Gérard}, \text{F}))\, \text{d}\mathbb{Q}' \end{aligned}$$

Here, $\mathbb{Q}' \neq \mathbb{Q}$ denotes the distribution of z_{weight}.

Which expectation to use? The example above highlights the differences between these two expectations. Neither is inherently better than the other, as each has its drawbacks and qualities. They both led to the design of an eponym algorithm, that is *Interventional TreeSHAP* and *Observational TreeSHAP*.
Resources are already available online to explain how the Interventional TreeSHAP algorithm is conceived, therefore will the following discussions only cover the Observational TreeSHAP whereabouts.

3 Observational TreeSHAP

3.1 Context and founding theorem

The important definitions and the mathematical formalism has been set, so we can now work on our initial question: how to explain the output of a model? Recalling equation 2, the naive way to calculate SHAP values consists in independently evaluating each difference $f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S)$ over all subsets S of features, and taking the weighted average to return the SHAP value of th e feature i. This method presents the advantage to be model-agnostic, which means one does not need to know how the model f works to calculate the SHAP values. However, such a method is massively time-consuming because there are 2^K subsets S (where K is the number of features) and that for each subset, we need to evaluate $f_S(x_S)$ using either the Interventional or the Observational paradigm. As said before, we specifically focus on the Observational Expectation method applied to decision trees such as the one in figure 1.

How to compute the Observational Expectation? For the sake of conformity with the notations commonly used in the literature, we will represent a decision tree $\mathcal{A}$ by six vectors: $\mathcal{A} = (v, a, b, t, r, d)$, where

- v represents the vector of nodes: if the node is internal, it has no value; we set $v_i =$ 'internal'.
- a (resp. b) are the indices of the left (resp. right) nodes for each internal node.
- t contains the decision threshold (e.g., $x_5 \leq 7 \Rightarrow t_i = 7$) for each internal node.
- d contains the index of the feature on which the decision is based (e.g., $x_5 \leq 7 \Rightarrow d_i = 5$) for each internal node.
- r contains, for each node, the number of data points from the training set that have passed through this node.

To calculate $\mathbb{E}[f(z) \mid z_S]$, we shall proceed recursively: at a node n, if $n_{\text{feature}} \in S$ then we follow the only edge which satisfies the condition on n_{feature}, otherwise we calculate the weighted average of the expectations calculated in the two lower paths. In Python, the computation of $\mathbb{E}[f(z) \mid z_S]$ is written as follows:

```
def observationalExpectation(tree, x, S):
  v,a,b,t,r,d = tree
  def rec(j):
    if v[j] != 'internal':
      return v[j]
    else :
      if d[j] in S:
        return rec(a[j]) if (x[d[j]] <= t[j]) else rec(b[j])
      else :
        return (rec(a[j])*r[a[j]] + rec(b[j])*r[b[j]])/r[j]
  return rec(1) # We start at the root of the tree
```

If we use the `observationalExpectation` function to compute each expectation and then directly calculate the SHAP values using their definition, the algorithmic complexity becomes exponential. Hence, a more clever approach is needed. From the previous algorithm, we can derive an explicit expression for $\mathbb{E}[f(z) \mid z_S]$. Indeed, let $v_1, \ldots, v_K$ denote the values of the K leaves of our decision tree. A leaf v_k contributes to $\mathbb{E}[f(z) \mid z_S]$ whenever it is reached, meaning that each edge of the path P_k (leading from the root to leaf v_k) is traversed. For an edge to be traversed, either:

- the feature d_j on which the condition for traversing that edge $j \in P_k$ is based is in S (i.e., $d_j \in S$) and satisfies the condition (i.e., $x_{d_j} \leq t_{k,j}$ if it's a left edge or $x_{d_j} > t_{k,j}$ if it's a right edge, where $t_{k,j}$ is the threshold of the condition). We denote $x_{d_j} \in T_{k,j}$ to indicate that x_{d_j} satisfies this condition, and D_k is the set of features d_j on which at least one condition of the path P_k depends.
- or $d_j \notin S$, and in this case, the edge is traversed, but we keep track of the weight of this edge $R_{k,j}$.

Ultimately, we have:

$$\mathbb{E}[f(z) \mid z_S] = \sum_{k=1}^{K} \Bigg[\underbrace{\prod_{\substack{j \in P_k \\ \text{such that } d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}}}_{\text{known feature } d_j} \cdot \underbrace{\prod_{\substack{j \in P_k \\ \text{such that } d_j \notin S}} R_{k,j}}_{\text{unknown feature } d_j} \Bigg] v_k \tag{15}$$

We denote $\omega_{k,S}$ as the coefficient in front of v_k in this summation.

From equation 15, we can derive an interesting formula for the SHAP values (theorem 1 in [3]), given by the theorem below.

Theorem 2. The SHAP value of a feature i can be obtained through the following equation:

$$\phi_i(f,x) = \sum_{\{k \mid i \in D_k\}} \Bigg[\sum_{h=0}^{|D_k|-1} \frac{h!(|D_k|-h-1)!}{|D_k|!} \sum_{\substack{S \subseteq D_k \setminus \{i\} \\ |S|=h}} \left(\prod_{\substack{j \in P_k \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus (S \cup \{i\})}} R_{k,j} \right) \Bigg]$$
$$\cdot \left(\prod_{\substack{j \in P_k \\ d_j = i}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} - \prod_{\substack{j \in P_k \\ d_j = i}} R_{k,j} \right) v_k$$

Based on Theorem 2, we will detail the original TreeSHAP algorithm, which made the `shap` package famous. This algorithm is essentially a clever way of "traversing" the tree simultaneously for all subsets S and keeping track of the number of subsets that end up at the same leaf. However, due to the weight $\frac{|S|!(|F|-|S|-1)!}{|F|!}$ depending on the size of S, this information must also be retained.

3.2 Function UNWIND

let us start at the end. Suppose that at a leaf node v_k (which lies at the end of path P_k), we have all the necessary information to calculate the contribution of this path to the SHAP values. According to Theorem 2, this contribution for the SHAP value $\phi_i(f,x)$ is equal to:

$$\sum_{h=0}^{|D_k|-1} \frac{h!(|D_k|-h-1)!}{|D_k|!} \sum_{\substack{S \subseteq D_k \setminus \{i\} \\ |S|=h}} \left(\prod_{\substack{j \in P_k \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus (S \cup \{i\})}} R_{k,j} \right) \cdot \left(\prod_{\substack{j \in P_k \\ d_j = i}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} - \prod_{\substack{j \in P_k \\ d_j = i}} R_{k,j} \right) v_k \tag{16}$$

By iteratively adding the contributions of each path $P_1, \dots P_K$ to SHAP values initialized to 0, we finally obtain the values $\phi_i(f,x)$. Therefore, we need to be able to compute this term at a leaf node.

Proposition 1. It turns out that from the expressions:

$$w_{h,\text{wound}} = \frac{h!(|D_k|-h)!}{(|D_k|+1)!} \sum_{\substack{S \subseteq D_k \\ |S|=h}} \left(\prod_{\substack{j \in P_k \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus S}} R_{k,j} \right) \tag{17}$$

for $h = 0, 1, \dots, |D_k|$ and the following terms $o_{k,i}$ (with o for 'ones') and $z_{k,i}$ (with z for 'zeros'):

$$o_{k,i} = \prod_{\substack{j \in P_k \\ d_j = i}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \quad \text{and} \quad z_{k,i} = \prod_{\substack{j \in P_k \\ d_j = i}} R_{k,j} \tag{18}$$

it is possible to compute the quantity:

$$w_{s,\text{unwound}} = \frac{s!(|D_k|-s-1)!}{|D_k|!} \sum_{\substack{S \subseteq D_k \setminus \{i\} \\ |S|=s}} \left(\prod_{\substack{j \in P_k \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus (S \cup \{i\})}} R_{k,j} \right) \tag{19}$$

for $s = 0, 1, \dots, |D_k| - 1$.

The motivation behind introducing Proposition 1 is that the expression in Equation 16 is exactly equal to $\sum_{s=0}^{|D_k|-1} w_{s,\text{unwound}} \cdot (o_{k,i} - z_{k,i}) \cdot v_k$. Its proof can be find in annexe A and gives the wanted relationships:

- if $o_{k,i} = 0$, then $w_{h,\text{unwound}} = w_{h,\text{wound}} \cdot \frac{|D_k|+1}{|D_k|-h} \cdot \frac{1}{z_{k,i}}$ for $h = 0, 1, \ldots, |D_k| - 1$;
- otherwise (when $o_{k,i} = 1$), $w_{h,\text{unwound}} = \omega_{h+1} \cdot \frac{|D_k|+1}{h+1} \cdot \frac{1}{o_{k,i}}$ with

$$\omega_h = \begin{cases} w_{|D_k|,\text{wound}} & \text{if } h = |D_k| \\ w_{h,\text{wound}} - \frac{|D_k|-h}{|D_k|+1} \cdot w_{h,\text{unwound}} \cdot z_{k,i} & \text{if } 0 \leq h \leq |D_k| - 1 \end{cases} \tag{20}$$

Based on this demonstration, we can devise a function that computes the quantity $w_{s,\text{unwound}}$ for $s = 0, 1, \ldots, |D_k| - 1$:

```
def UNWIND_weights(weights_wound, ones_ki, zeros_ki):
  Dk = len(weights_wound) - 1
  weights_unwound = np.zeros(Dk)

  # First case
  if ones_ki == 0:
    for h in range(0, Dk):
      weights_unwound[h] = (Dk + 1)/(Dk - h) * weights_wound[h] / zeros_ki

  # Second case
  else:
    omega = np.zeros(Dk+1)
    omega[Dk] = weights_wound[Dk]

    for h in range(Dk-1, -1, -1): # h = Dk-1, Dk-2, ..., 1, 0
      weights_unwound[h] = omega[h+1] * (Dk+1)/(h+1) * 1/ones_ki
      omega[h] = weights_wound[h] - (Dk-h)/(Dk+1) * weights_unwound[h] * zeros_ki

  return list(weights_unwound)
```

Thanks to the `UNWIND_weights` function, we're now able to compute the total contribution of a path P_k to the SHAP values once we have reached the leaf v_k. The next step is figuring out how to efficiently record the expressions $w_{h,\text{wound}}$, $o_{k,i}$ and $z_{k,i}$ during the tree traversal. This is the focus of the following section.

Note. Later on, there will be a (very) slight modification to the `UNWIND_weights` expression to create the more generally applicable `UNWIND` function.

3.3 Function EXTEND

To be able to store the values $w_{h,\text{wound}}$, $o_{k,i}$, and $z_{k,i}$ at the leaf level, we will build these quantities gradually during the recursive traversal of the tree.

To understand how to proceed, let us take an example. First, consider a simplification of the tree in Figure 1, represented by Figure 2. In this tree, the edges have different colors only to visually distinguish them from each other. They are also numbered according to their destination node. For instance, the cyan-colored edge is edge number 2 because it leads to node 2 (which is a leaf, by the way). This tree is simpler than the one in Figure 1 because it is shallower, and also because no feature appears twice in the same path (unlike the 'age' feature in the tree of Figure 1). As we will see, this will be helpful. let us again consider Gérard with $x = (56 \text{ years}, 93 \text{ kg}, 187\text{cm}, 37\%)$.

Consider an edge $e \in P_k$, and let P be the portion of the path P_k starting from the root and ending at edge e (excluding it). The depth of the tree up to the node from which edge e emerges is δ. Finally, let D be the set of distinct features encountered on P. For instance, for the path P_2 leading to the leaf '50%' in Figure 2, we have:
In Figure 3, let e be the last edge (which goes from node 4 to node 5). Upon reaching node 4, let us denote P as the path traveled. We have $\delta = 1$, and $D = \{\text{age}\}$. Note that the first edge does not have

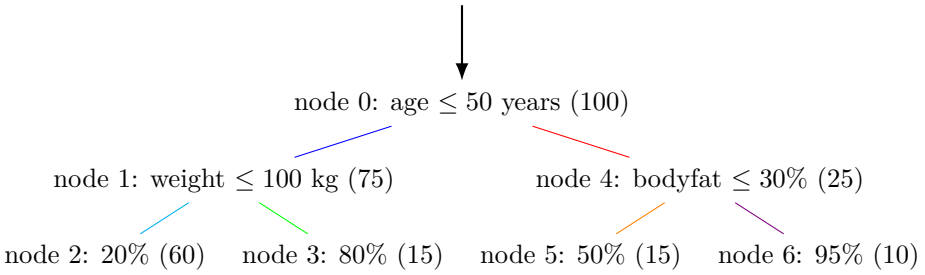

Figure 2: Decision tree with 3 features

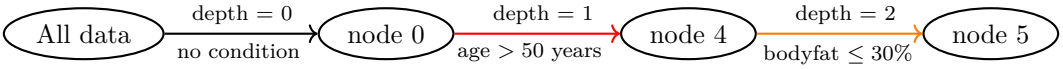

Figure 3: Example of a path from the root to a leaf

any conditions, so $|D| = \delta$ because the same feature is not encountered twice.

Every time we encounter a new node, it means we have traversed a new edge e. Three pieces of information are then available:

- the feature d_e on which the condition of edge e is based;
- whether or not this condition is satisfied by x, indicated by $\mathbf{1}_{\{x_{d_e} \in T_{k,e}\}}$;
- the proportion of training data $R_{k,e}$ that has passed through this edge e.

Our goal is to obtain, at the leaf level, the expressions $w_{h,\text{wound}}$, $o_{k,h}$, and $z_{k,h}$ for $h = 0, 1, \ldots, |D_k|$, and for this, we will rely on the following proposition.

Proposition 2. For $h = 0, 1, \ldots, |D|$, denote:

$$w_{h,P} = \frac{h!(|D| - h)!}{(|D| + 1)!} \sum_{\substack{S \subseteq D \\ |S| = h}} \left(\prod_{\substack{j \in P \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P \\ d_j \in D \setminus S}} R_{k,j} \right) \tag{21}$$

and:

$$o_{h,P} = \prod_{\substack{j \in P \\ d_j = d_h}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \quad \text{and} \quad z_{h,P} = \prod_{\substack{j \in P_k \\ d_j = d_h}} R_{k,j} \tag{22}$$

Then, knowing d_e, $\mathbf{1}_{\{x_{d_e} \in T_{k,e}\}}$ and $R_{k,e}$, it is possible to deduce $w_{h,P \cup \{e\}}, o_{h,P \cup \{e\}}$ and $z_{h,P \cup \{e\}}$ for $h = 0, 1, \ldots, |D \cup \{d_e\}|$.

Note that Proposition 2 concerns adding a new feature to the path, whereas Proposition 1 considered deletion. This proposition is intriguing because starting from an empty path and iteratively adding all edges during the traversal of P_k, we ultimately obtain the expressions of $w_{h,\text{wound}}$, $o_{k,h}$, and $z_{k,h}$ desired at the leaf level.

Building upon Proposition 2 and its proof (which may again be found in annexe A), we can explicitly define a function that computes the quantities $w_{h,P \cup \{e\}}, o_{h,P \cup \{e\}}$, and $z_{h,P \cup \{e\}}$ for $h = 0, 1, \ldots, |D \cup \{d_e\}|$.

```
def EXTEND(features_P, ones_P, zeros_P, weights_P, feature_e, one_e, zero_e):

  # We save the encountered feature, the condition and the proportion
  features_Pe = features_P + [feature_e]
  ones_Pe = ones_P + [one_e]
  zeros_Pe = zeros_P + [zero_e]
```

```
  D = len(weights_P)-1 # Number of distinct features met uptill now

  # First case: weights_P is empty, we leave 'All Data' node
  if not weights_P:
    weights_Pe = [1]

  # Second case: weights_P is not empty
  else:
    weights_P_ = weights_P + [0] # We add w[D+1 , P] = 0

    # We compute the new weights
    weights_Pe = np.zeros(D+2)
    weights_Pe[0] = (D+1)/(D+2) * weights_P_[0] * zero_e
    for h in range(0, D+1):
      weights_Pe[h+1] += (h+1)/(D+2) * weights_P_[h] * one_e
      weights_Pe[h+1] += (D-h)/(D+2) * weights_P_[h+1] * zero_e

  return features_Pe, ones_Pe, zeros_Pe, list(weights_Pe)
```

With the function `EXTEND`, we are able to track our progress deeper into the tree. We have all the tools at our disposal now to present the TreeSHAP algorithm itself.

3.4 The TreeSHAP Algorithm

We can now present the TreeSHAP algorithm itself. Our goal is to compute the SHAP values based on Theorem 2 and the functions UNWIND and EXTEND. It's a recursive algorithm, so the following explanations consider an arbitrary node `node` of the tree, but we will subsequently apply the algorithm to the root of the tree.

Firstly, when arriving at a new node, we know the distinct `features_` encountered, we know the weights `weights_` from Equation 17, and the `ones_` and `zeros_` from Equation 18. These pieces of information pertain only to the previous node, so it's necessary to record the edge that has just been traversed: from this edge, we retain the `feature_e` on which the traversal condition of the edge depends, whether or not this condition is met by x (through `one_e`), and the portion `zero_e` of training data that has passed through this edge. This is the utility of the `EXTEND` function:

```
args = (features_, ones_, zeros_, weights_, feature_e, one_e, zero_e)
features_, ones_, zeros_, weights_ = EXTEND(*args)
```

At this point, there are two possibilities: either the node is internal or it's a leaf.

At a leaf level, Proposition 1 enables us to calculate the contributions from the root to this leaf via the function `UNWIND_weights`:

```
if node.is_leaf: # Case 1: the node is a leaf
  for i in range(1, len(weights_)):
    phi[features_[i]] += sum(UNWIND_weights(weights_, ones_[i], zeros_[i])) * (ones_[i
                                          ] - zeros_[i]) * node.value
```

At a new internal node, we need to determine, for each of the two child nodes, the quantities `feature_e`, `one_e`, and `zero_e`, which will be necessary during the recursive call on the subsequent nodes. We start by determining for which child node we have `one_e = 1` (the 'hot child') and for which we have `one_e = 0` (the 'cold child'), then we record the aforementioned quantities:

```
if x[node.feature] <= node.threshold:
  hot_child, cold_child = node.left_child, node.right_child
else:
  cold_child, hot_child = node.left_child, node.right_child

feature_e = node.feature # Identical for both hot and cold children
hot_one_e = 1 # By definition of the hot child
cold_one_e = 0
hot_zero_e = hot_child.n_samples / node.n_samples
cold_zero_e = cold_child.n_samples / node.n_samples
```

Considering Figure 2, it might seem like there is nothing to add, and we could proceed directly to the recursive call for all previous operations on the 'hot' and 'cold' nodes. But that would overlook a crucial condition for using the `EXTEND` function, which will be called at the beginning of these two recursive calls: all features encountered on the path must be distinct. In Figure 1, we can demonstrate a path on which the feature 'age' appears twice (see Figure 4). To manage this situation, note

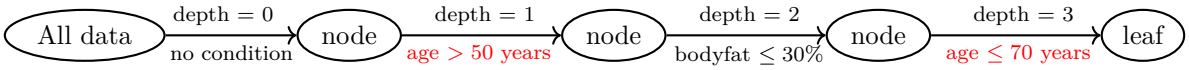

Figure 4: A path with feature 'age' appearing two times

that the function `UNWIND_weights` does more than just calculate weights to sum up (the term `sum(UNWIND_weights(...))`) for computing SHAP values; it goes further: it precisely calculates what `weights_` would have been if the feature i had not been encountered. Simply observe equations 19 and 17 to realize this. This means that we can handle a previously encountered feature by applying `UNWIND_weights` with i as the feature in question, allowing a recursive call of `EXTEND` that complies with its assumptions. However, two problems arise that need resolution:

- When applying `UNWIND_weights`, a feature and its associated data are removed from `weights_`. Therefore, it's crucial not to forget to also remove the data related to this feature from the lists `features_`, `ones_`, and `zeros_`;
- The `hot_zero_e` and `cold_zero_e` weights must be adjusted to account for the fact that we have already encountered feature i.

The first problem can be addressed by creating a function `UNWIND`, primarily based on `UNWIND_weights`, but that also update the lists `features_`, `ones_`, and `zeros_`:

```
def UNWIND(features_wound, ones_wound, zeros_wound, weights_wound, i):

  ## UNWIND_weights ##
  weights_unwound = UNWIND_weights(weights_wound, ones_wound[i], zeros_wound[i])

  ## UNWIND features_wound, ones_wound, zeros_wound ##
  features_unwound = list(np.delete(features_wound, i))
  ones_unwound = list(np.delete(ones_wound, i))
  zeros_unwound = list(np.delete(zeros_wound, i))

  return features_unwound, ones_unwound, zeros_unwound, weights_unwound
```

For the second problem, the first time we encountered feature i (let us say at edge ε), we recorded $\mathbf{1}_{\{x_i \in T_{k,\varepsilon}\}}$ and $R_{k,\varepsilon}$. When encountering feature i a second time, at edge e this time, we would typically record $\mathbf{1}_{\{x_i \in T_{k,e}\}}$ and $R_{k,e}$. In conclusion, here we will record neither of these two propositions but rather their product:

$$\mathbf{1}_{\{x_i \in T_{k,\varepsilon}\}} \cdot \mathbf{1}_{\{x_i \in T_{k,e}\}} \quad \text{and} \quad R_{k,\varepsilon} \cdot R_{k,e} \tag{23}$$

Finally, we can write a few lines of code that check whether the feature of the descendant edge has already been encountered or not, and act accordingly:

```
feature_e_indexes = np.where(features_==feature_e)[0]
if len(feature_e_indexes) > 0: # the feature has already been encountered
  i = feature_e_indexes[0]
  hot_one_e *= ones_[i]
  hot_zero_e *= zeros_[i]
  cold_zero_e *= zeros_[i]

# UNWIND to delete feature_e
args = (features_, ones_, zeros_, weights_, i)
features_, ones_, zeros_, weights_ = UNWIND(*args)
```

We do not need to specify `cold_one_e *= ones_[i]` because `cold_one_e = 0` in any case.

let us put it all together, without forgetting the final recursive calls:

```
def treeSHAP(tree, x):
  phi = np.zeros(len(x)) # We initialize the SHAP values at 0

  def RECURSE(node, features_, ones_, zeros_, weights_, feature_e, one_e, zero_e):

    args = (features_, ones_, zeros_, weights_, feature_e, one_e, zero_e)
    features_, ones_, zeros_, weights_ = EXTEND(*args)

    if node.is_leaf: # Case 1: the node is a leaf
      for i in range(1, len(weights_)):
        phi[features_[i]] += sum(UNWIND_weights(weights_, ones_[i], zeros_[i])) * (
                                          ones_[i] - zeros_[i]) * node.value

    else: # Case 2: the node is internal
      # We establish which child is hot and which one is cold
      if x[node.feature] <= node.threshold:
        hot_child, cold_child = node.left_child, node.right_child
      else:
        cold_child, hot_child = node.left_child, node.right_child

      feature_e = node.feature # Identical for both hot and cold children
      hot_one_e = 1 # By definition of the hot child
      cold_one_e = 0
      hot_zero_e = hot_child.n_samples / node.n_samples
      cold_zero_e = cold_child.n_samples / node.n_samples

      # We check whether feature_e has already been encountered
      feature_e_indexes = np.where(features_==feature_e)[0]
      if len(feature_e_indexes) > 0: # the feature has already been encountered
        i = feature_e_indexes[0]
        hot_one_e *= ones_[i]
        hot_zero_e *= zeros_[i]
        cold_zero_e *= zeros_[i]

        # UNWIND to delete feature_e
        args = (features_, ones_, zeros_, weights_, i)
        features_, ones_, zeros_, weights_ = UNWIND(*args)

      # Final recursive calls
      RECURSE(hot_child, features_, ones_, zeros_, weights_, feature_e, hot_one_e,
                                          hot_zero_e)
      RECURSE(cold_child, features_, ones_, zeros_, weights_, feature_e, cold_one_e,
                                          cold_zero_e)

  # We call RECURSE on the root of the tree,
  # by setting feature_e = -1 to indicate there is no feature on the first edge
  RECURSE(tree.root, [], [], [], [], -1, 1, 1)

  return phi
```

Here is finally the code of the TreeSHAP algorithm as firstly described by the authors.

A Annexe - Proofs

A.1 Proof of theorem 2

Proof of theorem 2. By inserting the expression $\mathbb{E}[f(z) \mid z_S] = \sum_{k=1}^{K} \omega_{k,S} v_k$ into the definition of the SHAP value, we get:

$$\phi_i(f,x) = \sum_{S \subseteq F \backslash \{i\}} \frac{|S|!(|F|-|S|-1)!}{|F|!} \sum_{k=1}^{K} (\omega_{k,S\cup\{i\}} - \omega_{k,S}) v_k \tag{24}$$

We notice that $\omega_{k,S\cup\{i\}} = \omega_{k,S}$ when $i \notin D_k$, i.e., when feature i is not used on path P_k. Thus, we can restrict the sum over k to only those k such that $i \in D_k$:

$$\phi_i(f,x) = \sum_{k|i\in D_k} \sum_{S \subseteq F \backslash \{i\}} \frac{|S|!(|F|-|S|-1)!}{|F|!} (\omega_{k,S\cup\{i\}} - \omega_{k,S}) \cdot v_k \tag{25}$$

Given that $F\backslash\{i\} = (F\backslash\{i,\ell\}) \cup \{S \cup \{\ell\} \mid S \in F\backslash\{i,\ell\}\}$, we have:

$$\begin{aligned}\phi_i(f,x) = \sum_{k|i\in D_k} \Bigg[& \sum_{S \subseteq F \backslash \{i,\ell\}} \frac{|S|!(|F|-|S|-1)!}{|F|!} (\omega_{k,S\cup\{i\}} - \omega_{k,S}) \cdot v_k \\ & + \sum_{S \subseteq F \backslash \{i,\ell\}} \frac{(|S|+1)!(|F|-|S|-2)!}{|F|!} (\omega_{k,S\cup\{i,\ell\}} - \omega_{k,S\cup\{\ell\}}) \cdot v_k \Bigg]\end{aligned}$$

Similarly to before, if $\ell \notin D_k$ and $\ell \neq i$, then for all $S \subseteq F\backslash\{\ell,i\}$ we have $\omega_{k,S\cup\{\ell,i\}} = \omega_{k,S\cup\{i\}}$ et $\omega_{k,S\cup\{\ell\}} = \omega_{k,S}$. Hence, for those ℓ:

$$\omega_{k,S\cup\{\ell,i\}} - \omega_{k,S\cup\{\ell\}} = \omega_{k,S\cup\{i\}} - \omega_{k,S} \tag{26}$$

Moreover:

$$\frac{|S|!(|F|-|S|-1)!}{|F|!} + \frac{(|S|+1)!(|F|-|S|-2)!}{|F|!} = \frac{|S|!(|F|-|S|-2)!}{(|F|-1)!} \tag{27}$$

Hence:

$$\phi_i(f,x) = \sum_{k|i\in D_k} \left[\sum_{S \subseteq F \backslash \{i,\ell\}} \frac{|S|!(|F|-|S|-2)!}{(|F|-1)!} (\omega_{k,S\cup\{i\}} - \omega_{k,S}) \right] \cdot v_k \tag{28}$$

he only difference between equations 25 and 28 lies in the transition from $F\backslash\{i\}$ to $F\backslash\{i,\ell\}$ and from $|F|$ to $|F|-1$. By repeating these steps for all $\ell \in F\backslash D_k$, we eventually obtain:

$$\phi_i(f,x) = \sum_{k|i\in D_k} \left[\sum_{S \subseteq D_k \backslash \{i\}} \frac{|S|!(|D_k|-|S|-1)!}{|D_k|!} (\omega_{k,S\cup\{i\}} - \omega_{k,S}) \right] \cdot v_k \tag{29}$$

The proof is almost complete. We let $W(|S|,|D_k|) := \frac{|S|!(|D_k|-|S|-1)!}{|D_k|!}$ for simplicity. The only remaining step is to reintroduce the expression of $\omega_{k,S}$:

$$\begin{aligned}
\phi_i(f,x) &= \sum_{k|i\in D_k}\left[\sum_{S\subseteq D_k\setminus\{i\}} W(|S|,|D_k|)\left(\prod_{\substack{j\in P_k\\ d_j\in S\cup\{i\}}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}}\prod_{\substack{j\in P_k\\ d_j\notin S\cup\{i\}}}R_{k,j} - \prod_{\substack{j\in P_k\\ d_j\in S}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}}\prod_{\substack{j\in P_k\\ d_j\notin S}}R_{k,j}\right)\right]v_k \\
&= \sum_{k|i\in D_k}\left[\sum_{S\subseteq D_k\setminus\{i\}} W(|S|,|D_k|)\cdot\left(\prod_{\substack{j\in P_k\\ d_j\in S}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}}\prod_{\substack{j\in P_k\\ d_j\notin S\cup\{i\}}}R_{k,j}\right)\right. \\
&\quad \left.\cdot\left(\prod_{\substack{j\in P_k\\ d_j=i}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}} - \prod_{\substack{j\in P_k\\ d_j=i}}R_{k,j}\right)\right]v_k \\
&= \sum_{k|i\in D_k}\left[\sum_{h=0}^{|D_k|-1}\frac{h!(|D_k|-h-1)!}{|D_k|!}\sum_{\substack{S\subseteq D_k\setminus\{i\}\\ |S|=h}}\left(\prod_{\substack{j\in P_k\\ d_j\in S}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}}\prod_{\substack{j\in P_k\\ d_j\notin S\cup\{i\}}}R_{k,j}\right)\right] \\
&\quad \cdot\left(\prod_{\substack{j\in P_k\\ d_j=i}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}} - \prod_{\substack{j\in P_k\\ d_j=i}}R_{k,j}\right)v_k
\end{aligned}$$

This achieves the proof. □

A.2 Proof of proposition 1

Proof of proposition 1. We will distinguish two cases, depending on whether $o_{k,i}=0$ or $o_{k,i}=1$.

First case: $o_{k,i}=0$ In the expression for $w_{h,\text{wound}}$, note that:

$$\prod_{\substack{j\in P_k\\ d_j\in S}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}} = \prod_{\gamma\in S}\prod_{\substack{j\in P_k\\ d_j=\gamma}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}} \tag{30}$$

Thus, for any S such that $i\in S$, $o_{k,i}=0$, and the above product is null. Therefore, we can write:

$$w_{h,\text{wound}} = \frac{h!(|D_k|-h)!}{(|D_k|+1)!}\sum_{\substack{S\subseteq D_k\setminus\{i\}\\ |S|=h}}\left(\prod_{\substack{j\in P_k\\ d_j\in S}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}}\prod_{\substack{j\in P_k\\ d_j\in D_k\setminus S}}R_{k,j}\right) \tag{31}$$

As soon as $i\notin S$, we have $i\in D_k\setminus S$, so the expression can be rewritten as:

$$w_{h,\text{wound}} = \frac{|D_k|-h}{|D_k|+1}\cdot\frac{h!(|D_k|-h-1)!}{|D_k|!}\sum_{\substack{S\subseteq D_k\setminus\{i\}\\ |S|=h}}\left(\prod_{\substack{j\in P_k\\ d_j\in S}}\mathbf{1}_{\{x_{d_j}\in T_{k,j}\}}\prod_{\substack{j\in P_k\\ d_j\in D_k\setminus(S\cup\{i\})}}R_{k,j}\right)\cdot\prod_{\substack{j\in P_k\\ d_j=i}}R_{k,j} \tag{32}$$

i.e $w_{h,\text{wound}} = \frac{|D_k|-h}{|D_k|+1}\cdot w_{h,\text{unwound}}\cdot z_{k,i}$ for $h=0,1,\ldots,|D_k|-1$.

Second case: $o_{k,i}=1$ let us begin by introducing an auxiliary sequence, denoted by (ω_h). We define it as:

$$\omega_h = \begin{cases} w_{|D_k|,\text{wound}} & \text{if } h=|D_k| \\ w_{h,\text{wound}} - \frac{|D_k|-h}{|D_k|+1}\cdot w_{h,\text{unwound}}\cdot z_{k,i} & \text{if } 0\le h\le |D_k|-1 \end{cases} \tag{33}$$

Now, let us prove that $w_{h,\text{unwound}} = \omega_{h+1} \cdot \frac{|D_k|+1}{h+1} \cdot \frac{1}{o_{k,i}}$ for $h = 0, 1, \ldots, |D_k| - 1$.
For $h = |D_k| - 1$, we have:

$$\omega_{|D_k|} \cdot \frac{|D_k|+1}{|D_k|} \cdot \frac{1}{o_{k,i}} = w_{|D_k|,\text{wound}} \cdot \frac{|D_k|+1}{|D_k|} \cdot \frac{1}{o_{k,i}} = \frac{1}{|D_k|+1} \prod_{j \in P_k} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \cdot \frac{|D_k|+1}{|D_k|} \cdot \frac{1}{o_{k,i}}$$
$$= \frac{1}{|D_k|} \prod_{\substack{j \in P_k \\ d_j \neq i}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} = w_{h,\text{unwound}}$$

Assuming the equality above holds for $|D_k| - 1, \ldots, h+1$, let us show it is also true for h.

$$\omega_{h+1} \cdot \frac{|D_k|+1}{h+1} \cdot \frac{1}{o_{k,i}} = \left(w_{h+1,\text{wound}} - \frac{|D_k|-h-1}{|D_k|+1} \cdot w_{h+1,\text{unwound}} \cdot z_{k,i} \right) \frac{|D_k|+1}{h+1} \cdot \frac{1}{o_{k,i}} \tag{34}$$

Expanding the expression in parentheses above and factoring by $\frac{(h+1)!(|D_k|-h-1)!}{(|D_k|+1)!}$, the remaining term is

$$\sum_{\substack{S \subseteq D_k \\ |S|=h+1}} \left(\prod_{\substack{j \in P_k \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus S}} R_{k,j} \right) - \sum_{\substack{S \subseteq D_k \setminus \{i\} \\ |S|=h+1}} \left(\prod_{\substack{j \in P_k \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus (S \cup \{i\})}} R_{k,j} \right) \cdot z_{k,i}$$

or:

$$\sum_{\substack{S \subseteq D_k \\ |S|=h+1}} \left(\prod_{\substack{j \in P_k \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus S}} R_{k,j} \right) - \sum_{\substack{S \subseteq D_k \setminus \{i\} \\ |S|=h+1}} \left(\prod_{\substack{j \in P_k \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus S}} R_{k,j} \right)$$

The only difference between these two sums lies in their indexing. On one side, we sum over all $S \subseteq D_k$, and on the other side over $S \subseteq D_k \setminus \{i\}$. Therefore, we have:

$$\sum_{\substack{S \subseteq D_k \\ |S|=h+1 \\ i \in S}} \left(\prod_{\substack{j \in P_k \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus S}} R_{k,j} \right)$$

Thus, taking into account the weights $\frac{(h+1)!(|D_k|-h-1)!}{(|D_k|+1)!}$ and $\frac{|D_k|+1}{h+1} \cdot \frac{1}{o_{k,i}}$, we get:

$$\omega_{h+1} \cdot \frac{|D_k|+1}{h+1} \cdot \frac{1}{o_{k,i}} = \frac{h!(|D_k|-h-1)!}{|D_k|!} \sum_{\substack{S \subseteq D_k \\ |S|=h+1 \\ i \in S}} \left(\prod_{\substack{j \in P_k \\ d_j \in S \setminus \{i\}}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus S}} R_{k,j} \right)$$

Let $S' = S \setminus \{i\}$, so that

$$\Big(S \subseteq D_k,\ |S| = h+1,\ i \in S \Big) \quad \Leftrightarrow \quad \Big(S' \subseteq D_k \setminus \{i\},\ |S'| = h \Big) \tag{35}$$

This substitution allows us to conclude:

$$\omega_{h+1} \cdot \frac{|D_k|+1}{h+1} \cdot \frac{1}{o_{k,i}} = \frac{h!(|D_k|-h-1)!}{|D_k|!} \sum_{\substack{S' \subseteq D_k \setminus \{i\} \\ |S'|=h}} \left(\prod_{\substack{j \in P_k \\ d_j \in S'}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P_k \\ d_j \in D_k \setminus (S' \cup \{i\})}} R_{k,j} \right) = w_{h,\text{unwound}}$$

□

A.3 Proof of proposition 2

Proof of proposition 2. First, if no edge has been traversed yet, we are still on the 'All data' node, as in the example shown in Figure 3. Therefore, we traverse edge $e = 0$ leading to node 0. As there is no condition on this edge, let $d_e = -1$. Since there is no condition, the edge is always traversed, so

$\mathbf{1}_{\{x_{d_e} \in T_{k,e}\}} = 1$ and $R_{k,e} = 1$. After this traversal, $|D| = 0$, and we obtain $w_{0,\{e\}} = 1$, $o_{0,\{e\}} = 1$, $z_{0,\{e\}} = 1$.

Moving forward, let us assume we at least traversed one edge.
Since the feature d_e has not been encountered yet, we have:

$$o_{|D|+1,P\cup\{e\}} = \mathbf{1}_{\{x_{d_e} \in T_{k,e}\}} \quad \text{and} \quad z_{|D|+1,P\cup\{e\}} = R_{k,e} \tag{36}$$

Other values remain unchanged:

$$o_{h,P\cup\{e\}} = o_{h,P} \quad \text{and} \quad z_{h,P\cup\{e\}} = z_{h,P} \quad \text{for} \quad h \in \{0, 1, \ldots, |D|\} \tag{37}$$

Now, we aim to calculate the expression for $h = 0, 1, \ldots, |D| + 1$:

$$w_{h,P\cup\{e\}} = \frac{h!(|D| + 1 - h)!}{(|D| + 2)!} \sum_{\substack{S \subseteq D\cup\{d_e\} \\ |S|=h}} \left(\prod_{\substack{j \in P\cup\{e\} \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P\cup\{e\} \\ d_j \in D\cup\{d_e\}\setminus S}} R_{k,j} \right) \tag{38}$$

Let $w_{|D|+1,P} = 0$. we will actually demonstrate that:

$$w_{h+1,P\cup\{e\}} = \frac{h+1}{|D|+2} \cdot w_{h,P} \cdot \mathbf{1}_{\{x_{d_e} \in T_{k,e}\}} + \frac{|D| - h}{|D|+2} \cdot w_{h+1,P} \cdot R_{k,e} \tag{39}$$

for $h = -1, 0, 1, \ldots, |D|$.

The case $h = -1$ is trivial:

$$w_{0,P\cup\{e\}} = \frac{1}{|D|+2} \prod_{\substack{j \in P\cup\{e\} \\ d_j \in D\cup\{d_e\}}} R_{k,j} = \frac{|D|+1}{|D|+2} \cdot w_{0,P} \cdot R_{k,e} \tag{40}$$

let us assume $h \geq 0$. We have:

$$w_{h+1,P\cup\{e\}} = \frac{(h+1)!(|D| - h)!}{(|D| + 2)!} \sum_{\substack{S \subseteq D\cup\{d_e\} \\ |S|=h+1}} \left(\prod_{\substack{j \in P\cup\{e\} \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P\cup\{e\} \\ d_j \in D\cup\{d_e\}\setminus S}} R_{k,j} \right)$$

We can split the sum into two based on whether $d_e \in S$ or $d_e \notin S$, resulting in (ignoring the weight):

$$\sum_{\substack{S \subseteq D\cup\{d_e\} \\ |S|=h+1 \\ d_e \in S}} \left(\prod_{\substack{j \in P\cup\{e\} \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P\cup\{e\} \\ d_j \in D\cup\{d_e\}\setminus S}} R_{k,j} \right) + \sum_{\substack{S \subseteq D\cup\{d_e\} \\ |S|=h+1 \\ d_e \notin S}} \left(\prod_{\substack{j \in P\cup\{e\} \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P\cup\{e\} \\ d_j \in D\cup\{d_e\}\setminus S}} R_{k,j} \right)$$

let us handle these two terms separately for clarity.
On one side:

$$\begin{aligned}
& \frac{(h+1)!(|D| - h)!}{(|D| + 2)!} \sum_{\substack{S \subseteq D\cup\{d_e\} \\ |S|=h+1 \\ d_e \in S}} \left(\prod_{\substack{j \in P\cup\{e\} \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P\cup\{e\} \\ d_j \in D\cup\{d_e\}\setminus S}} k, j \right) \\
= \quad & \frac{h+1}{|D|+2} \cdot \frac{h!(|D| - h)!}{(|D|+1)!} \sum_{\substack{S \subseteq D \\ |S|=h}} \left(\prod_{\substack{j \in P \\ d_j \in S}} \mathbf{1}_{\{x_{d_j} \in T_{k,j}\}} \prod_{\substack{j \in P \\ d_j \in D\setminus S}} R_{k,j} \right) \cdot \mathbf{1}_{\{x_{d_e} \in T_{k,e}\}} \\
= \quad & \frac{h+1}{|D|+2} \cdot w_{h,P} \cdot \mathbf{1}_{\{x_{d_e} \in T_{k,e}\}}
\end{aligned}$$

On the other:

$$\begin{aligned}
& \frac{(h+1)!(|D|-h)!}{(|D|+2)!} \sum_{\substack{S\subseteq D\cup\{d_e\}\\ |S|=h+1\\ d_e\notin S}} \left(\prod_{\substack{j\in P\cup\{e\}\\ d_j\in S}} \mathbf{1}_{\{x_{d_j}\in T_{k,j}\}} \prod_{\substack{j\in P\cup\{e\}\\ d_j\in D\cup\{d_e\}\setminus S}} R_{k,j} \right) \\
= \quad & \frac{|D|-h}{|D|+2} \cdot \frac{(h+1)!(|D|-h-1)!}{(|D|+1)!} \sum_{\substack{S\subseteq D\\ |S|=h+1}} \left(\prod_{\substack{j\in P\\ d_j\in S}} \mathbf{1}_{\{x_{d_j}\in T_{k,j}\}} \prod_{\substack{j\in P\\ d_j\in D\setminus S}} R_{k,j} \right) \cdot R_{k,e} \\
= \quad & \frac{|D|-h}{|D|+2} \cdot w_{h+1,P} \cdot R_{k,e}
\end{aligned}$$

This last equality concludes the proof. □

References

[1] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions. 2017.

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

[3] Jilei Yang. Fast treeshap: Accelerating SHAP value computation for trees. *CoRR*, abs/2109.09847, 2021.