Examining SHAP

Bernhard Jaeger
University of Tübingen
Tübingen, Germany
bernhard.jaeger@student.uni-tuebingen.de

***Abstract*—Understanding the decisions made by machine learning models can be challenging due to their black box nature. We investigate the additive feature attribution method SHAP (SHapley Additive exPlanations) that addresses this problem [2]. While SHAP can be used to explain the decisions of any model, the generic KernelSHAP algorithm for it can be intractable for larger datasets. For tree models there exists a specialized algorithm (Tree SHAP) that can compute exact SHAP values in polynomial time [1]. We use the SHAP framework to investigate the relationship between Decision Trees and Hoeffding Trees [3]. Theory tells us that a Hoeffding Tree should converge towards a Decision Tree when trained. In our experiments however we could not observe convergence. The reason for this remains unclear.**

***Index Terms*—Tree SHAP, Kernel SHAP, Decision Tree, Hoeffding Tree**

I. Introduction

This . This and [3] document is a model and instructions for LaTeX. Please observe the conference page limits.

II. Additive Feature Attribution methods

Modern models of Machine Learning have become increasingly complex making it hard to explain their decisions. To address this challenge we can go one layer of abstraction further and make a model of the model which we want to be simple enough to be interpretable again. In [2] they coined the term explanation model for this approach of using an interpretable approximation of the original model.
When viewed trough this lens multiple of the current explanation methods can be categorized in the class of **Additive Feature Attribution methods**. In [2] they defined this as methods having an explanation model that is a linear function of binary variables:

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

Where g is the explanation model, $z' \in \{0,1\}^M, \phi \in \mathbb{R}$.
Local methods like this first approximate their model $f(x)$ using *simplified inputs* x' where $x = h_x(x')$.
The objective is then to ensure $g(z') \approx f(h_x(z'))$ whenever $z' \approx x'$.
ϕ_i is then called the **effect** of feature i and summing up all effects approximates the output of the model $f(x)$ we wish to explain. Note that these *explanation models* are now easily interpretable. We can for a prediction $f(x)$ given input x say what influence or effect each component x_i had on the final decision.
Methods falling in the category of Additive Feature Attribution methods include LIME [4], DeepLIFT [5], layer-wise relevance propagation [6], Quantitive Input Influence [7], Shapley sampling values [8] and SHAP [2].

Additive Feature Attribution methods can have 3 desirable theoretical properties [2]:

Local Accuracy:

$$f(x) = g(x') whenx = h_x(x') and\phi_0 = f(h_x(0)) \tag{2}$$

The intuition being that the explanation model should match the original model for the simplified input (notice $x' = z'$ here).

Missingness:

$$x'_i = 0 \rightarrow \phi_i = 0 \tag{3}$$

The intuition is that feature missing in the original input may have no impact even if they are present in the simplified version. (Notice that even if $x' = 0$, z' does not necessarily have to be 0).

Consistency:
$f_x(z') := f(h_x(z'))$, and let $z' \setminus i$ be $z'_i = 0$
For any two models f and f', if:

$$f'_x(z') - f'_x(z' \setminus i) \geq f_x(z') - f_x(z' \setminus i) \tag{4}$$

for all inputs $z' \in 0, 1^M$.
Meaning that the model changes in a way that some simplified input's contribution becomes larger.
Then: $\phi_i(f', x) \geq \phi_i(f, x)$
Meaning that it's inputs attribution should not become smaller.

Now it turns out that the only possible features for an Additive Feature attribution method that satisfies all 3 properties are the Shapley values defined as:

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!}[f_x(z') - f_x(z' \setminus i)] \tag{5}$$

$z' \subseteq x'$ are all z' vectors where the non-zero entries are a subset of the non-zero entries in x'.
$|z'|$ represents the number of non-zero entries in z'.

This means that at least in theory Additive Feature Attribution methods using Shapley Values as effects are to be preferred over other methods. In practice methods using shapely values can be intractable for larger sets.
In the next section we will introduce the SHAP framework which tries to unify these measures of feature importance.

III. SHAP values

SHAP (SHapley Additive exPlanations) values [2] are obtained by solving equation 5 using a conditional expectation function of the original model:

$$f(h_x(z')) = E[f(z)|z_S] \tag{6}$$

S is the set of non-zero indexes in z'. The mapping function is therefore implicitly defined as:

$$h_x(z') = z_S \tag{7}$$

z_S has missing values for features not in the set S.
One problem with this approach is that in general models don't can't necessarily handle features that are just missing.
To address this issue $f(z_S)$ can be approximated with $E[f(z)|z_S]$.
The intuition being that SHAP values calculate the attribution (or effect) of each feature using the change in expectation of the model prediction when you condition on that feature. The feature $\phi_0 = E[f(z)]$ is the base value or the prediction the model would make if it had no observation. The following ϕ_i are then calculated by conditioning on some input features and averaging over all possible orderings of them. As hinted already earlier the exact computation of SHAP values can be challenging as it requires to calculate combinatory many expectations. Assuming feature independence (equation 10) and model linearity (equation 11) can make the computation much easier.

$$\begin{aligned} f(h_x(z')) &= E[f(z)|z_S] && (8)\\ &= E_{z_{\overline{S}}|z_S}[f(z)] && (9)\\ &\approx E_{z_{\overline{S}}}[f(z)] && (10)\\ &\approx f([z_S, E[z_{\overline{S}}]]) && (11)\end{aligned}$$

It allows you to These are however strong assumptions as most modern models are non-linear and features are in general not independent of each other (e.g. a pixel value only has a meaningful interpretation when put into the context of the surrounding pixels).
In the following subsections we will discuss algorithms for computing SHAP values.

A. Kernel SHAP (Linear Lime + Shapely Value)

Kernel SHAP [2] is the only method presented here that can be applied to any model. Linear Lime is a method that minimizes the following objective function:

$$\xi = \underset{g \in G}{\operatorname{argmin}}\, L(f, g, \pi_{x'}) + \Omega(g) \tag{12}$$

Lime is a local linear explanation model that locally approximates the model around a given prediction. Kernel SHAP now chooses Ω, π and L in a way that recovers the Shapely Values (and therefore guarantees the properties missingness, local accuracy and consistency). This is possible because LIME is a additive feature attribution method. This lead to the following equations:

$$\Omega(g) = 0 \tag{13}$$

$$\pi_{x'}(z') = \frac{(M-1)}{(M choose |z'|)|z'|(M - |z'|)1} \tag{14}$$

$$L(f, g, \pi_{x'}) = \sum_{z' \in Z} [f(h_x(z')) - g(z')]^2 \pi_{x'}(z') \tag{15}$$

Because the loss is quadratic and $g(z')$ is assumed to be linear we can solve equation 12 using linear regression (and calculate the shapely values using weighted linear regression). The input mapping used by LIME is equivalent to equation 11 and we can therefore use Kernel SHAP on every model. The simplified input mapping $h_x(z')$ that LIME uses is equivalent to equation 11 which leads to estimation of the SHAP values.
To summarize Kernel SHAP is a model agnostic algorithm that under the (strong) assumptions model linearity and feature independence approximates the SHAP values. The time complexity of the algorithm is exponential in the number of input features M : $O(M2^M)$ [2].
An additional problem that arises in practice is that most models can not handle arbitrary patterns of missing inputs at inference time (which is a requirement to compute the Shapley values). Implementation of the algorithm therefore make further approximations by replacing missing features with random values from a background dataset provided by the user (e.g. the training set). Given these severe flaws one can question whether the explanations provided by Kernel SHAP can be trusted in a realistic setting.
Better algorithms for calculating SHAP values for specific model types exist. Some of which will be covered in the following subsections:

B. Linear SHAP

For a linear model $f(x) = \sum_{j=1}^{M} w_j x_j + b$ you can approximate the SHAP values by only assuming feature independence (equation 10). The features will then be:

$$\phi_0(f, x) = b\phi_i(f, x) = w_j(x_j - E[x_j]) \tag{16}$$

Notice that the difference to just looking at the weights is that Linear SHAP takes the difference to the baseline into account.

C. Deep SHAP

Deep SHAP [2] is an algorithm that combines shapely values with DeepLIFT [5]. It combines SHAP values that are analytically computed for smaller components of the network and combines them into SHAP values for the whole network. Doing so we are linearising the non linear components of the network and the way in which we do this is uniquely specified by the Shapley values. To calculate the expectation needed for SHAP we again have to assume that the input features are independent and that the deep models is linear. Notice that the second assumption is always violated as the point of adding depth to a neural network is that it becomes non-linear. Further details on the algorithm are omitted here an can be found in [2].

D. Tree SHAP

The previously discussed algorithms were all approximations of the SHAP values. The Tree SHAP [1] algorithm is able to compute **exact** SHAP values in **polynomial** time complexity: $O(TLD^2)$. T here is the number of trees in the ensemble, L is the maximum number of leaves in any of the trees and D is the maximum depth of any of these trees. This means that the algorithm is tractable even for high dimensional data (which Kernel Shap due to the exponential time complexity is not).

Global feature importance of trees can also be computed with methods based on Gain [9], Split Count [10] or Permutations [11] and locally using Sabbas [12]. However these methods are inconsistent, meaning features can become more important while the attribution decreases. One can relate this intuitively to the definition of consistency above (but the methods here are not necessarily additive feature attribution methods hence the lack of formality).

Tree SHAP does not have this problem as it uses (exact) Shapely values and therefore guarantees the properties local accuracy, missingness and consistency. The intuition of the algorithm is that it is recursively tracking the proportion of all possible subsets flowing down into the leaves of the tree. Doing so the algorithm requires polynomial memory in the order $O(D^2 + M)$. The algorithm is applicable to trees and sums of trees (which uses the fact that SHAP values of two functions added together is the sum of their individual SHAP values).

Note that Tree SHAP does **not** require the additional assumptions model linearity and feature independence that we had to make earlier. The algorithmic description in pseudo code can be found in [1] and an open source implementation is available at [13].

Now that we have a tool to precisely calculate SHAP values we can both globally for an entire dataset and locally on an individual sample inspect the importance of each feature as learned by the tree model gain insight into the model behaviour and potentially even the underlying problem if the model is good. This analysis can even be further expanded be calculating SHAP interaction values covered next.

E. SHAP interaction values

Using the more modern Shapely interaction index [14] one can calculate interaction effects between features and store them in a matrix [1]. It's entries represent the impact of pairs of features on the model behaviour. The Shapely Interaction Index is defined as follows:

$$\phi_{i,j} = \sum_{S \subseteq N \setminus \{i,j\}} \frac{|S|!(M - |S| - 2)!}{2(M-1)!} \nabla_{ij}(S), i \neq j \quad (17)$$

$$\nabla_{ij}(S) = f_X(S \cup \{i,j\}) - f_X(S \cup \{i\}) - f_X(S \cup \{j\}) + f_X(S) \quad (18)$$

Due to it's similarity with the classical Shapely values the Shapely interaction values can be computed by using the Tree Shap algorithm twice for each feature leading to a runtime of $O(TMLD^2)$. These SHAP interaction values now allow us to distinguish between the main effect of a feature and it's interaction effect with other features. The main effect being defined as:

$$\phi_{i,i} = \phi_i - \sum_{j \neq i} \phi_{i,j} \quad (19)$$

F. Supervised Clustering

Another interesting application of the individual feature attributions obtained by SHAP is called Supervised Clustering [1]. The idea is relatively straight forward. Instead of clustering features directly in an unsupervised fashion you first train an algorithm on the data using labels. You then calculate the corresponding SHAP values and then cluster on the feature attributions. With traditional clustering you have the problem that differences in the units of the features can have a large impact. With supervised clustering you don't have that problem because all attributions have the same unit as the models output. Fluctuations in the features value are also only considered if they are relevant for our (supervised) model.

G. SHAP plots

Now there are multiple ways to visualize the explanations generated by the SHAP framework [1]. The most traditional would be a bar chart where the global importance of each feature is plotted as absolute value. To plot the effects of a single example one can plot for example on a line how much each feature pushed the prediction of the model away from the baseline.

Additionally you can create *SHAP summary plots* which add more detail to the global view. In these features are again sorted by importance each individual attribution is then plotted as a dot plotted horizontally where the x axis show the SHAP value (effect). If multiple dots have the same SHAP value they get stacked creating an effect similar to violin plots. The actual feature value is then encoded through color.

SHAP dependence plots are another visual tool where the SHAP value of a feature is plotted axis against it's feature value on the x-axis. The individual points of the test set are then displayed like a scatter plot. An additional value of a different feature can be encoded in the color of the dots.

IV. Decision and Hoeffding Trees

V. Experiments

A. Setup

The dataset we used for our experiments was the UCI Adults dataset [15]. The objective for this dataset is to predict from demographic data if the person is going to make more than 50.000$ a year (or not). During preprocessing we replaced missing values (”?”) with the most frequent attribute of the respective column. People who make less than 50k a year were labelled with 0 and the other once with 1. Because the sklearn decision tree implementation does not support mixed type (categorical and numeric) input data we converted categorical values to numerical ones using the sklearn label encoder. This is likely going to lead to worse performance of the model. However since we were not interested in building high accuracy models this should not matter.

As models we used the DecisionTreeClassifier from sklearn and HoeffdingTreeClassifier from scikit-multiflow. For the split criterion we used gini for both models. All the other hyper-parameters did unfortunately not have an overlap between the two libraries. Notably we set the min_sample_split parameter of the decision tree to 1000 (which stopped the model from over-fitting) and the leaf_prediction of the Hoeffding Tree to “mc” (because we think that it coincides best with the decision tree).

B. Experiments

We then trained both trees on the entire training set (consisting of 32561 data samples). Additionally we trained another Hoeffding tree by feeding it each sample individually. By plotting the train and test accuracy every 5000 samples we observed that the Hoeffding tree trained on individual samples converged to the Hoeffding tree trained on all data at once. The resulting trees both had the exact (16 digits) same training and test accuracy suggesting that the exact same tree structure has emerged.

Somewhat surprisingly the Decision Tree showed a difference in accuracy compared to the Hoeffding tree (in the configuration described here about 3 percent). Theory tells us that Hoeffding trees should converge towards decision trees given enough data. We also tried different combinations of hyper parameters (as said before they are quite different between the two libraries) but could not find a setting with similar performance. Notably the decision tree heavily overfitted when min_sample_split was set to the default value of 2 (almost 0 training error). Increasing it substantially regularized the model until the train and test performance were similar. The Hoeffding tree does not have this parameter and never showed overfitting behaviour.

We then tried to further investigate this behaviour by using the Tree SHAP algorithm to explain what the models look for.

Unfortunately it turned out that the current implementation of the Tree SHAP algorithm [16] is not compatible with the scikit-multiflow library (or any other library implementing Hoeffding trees). This is not a theoretical problem (Tree SHAP works on every tree model), the existing open source libraries are simply incompatible right now.

We therefore used the model agnostic Kernel SHAP algorithm instead to explain the Hoeffding Tree. Since the Kernel SHAP as discussed before is really slow we had to restrict the analysis to a background dataset of 50 samples from the training set and calculated the SHAP values for 50 samples of the test set. To stay comparable we used this subset for both algorithms.

Additionally for the Kernel SHAP we set the attribute l1 regularisation to “num_features(10)” as suggested by the library.

We benchmarked the performance of both algorithms using the ipython notebook command %%time. Our hardware setup can be found in section VII

Tree SHAP was able to calculate the SHAP values for the Decision Tree in only 15ms wall time.

Even with this small subset of the data Kernel SHAP still needed 4 minutes and 49 seconds wall time. For the full test set the algorithm seemed to be intractable.

Using the calculated SHAP values we investigated the models overall using summary bar plots which can be seen in figure 1 and figure 2. An individual prediction from the test set is visualized in figure 3 and figure 4.

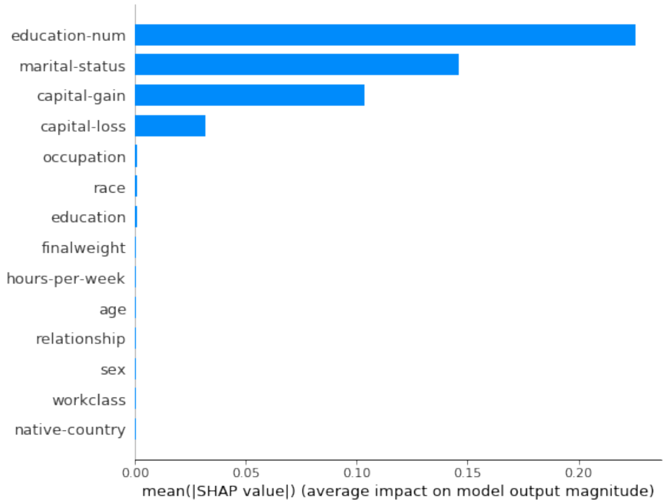

Fig. 1. Summary plot of a Hoeffding Tree explained with Kernel SHAP using 50 samples

One can clearly see that the two models look for different attributes when making their predictions and even on the attributes they agree upon (like education-num) the magnitude of their importance varies.

Explaining the observation that the Hoeffding tree doesn’t necessarily converge to a decision tree in practice remains an open question. Some hypothesis we have are: Differences in

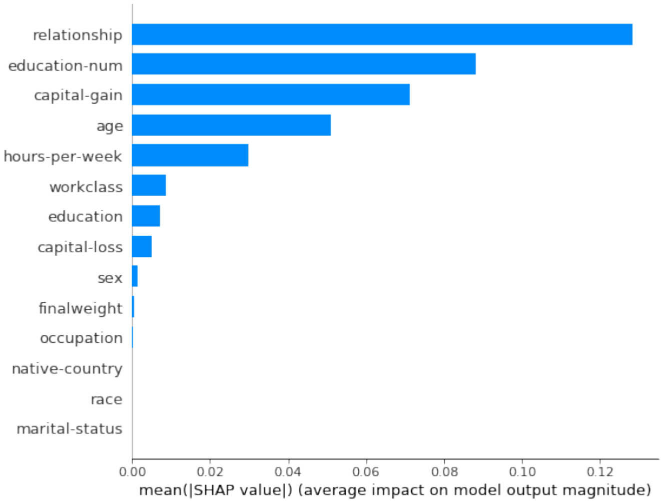

Fig. 2. Summary plot of a Decision Tree explained with Tree SHAP using 50 samples

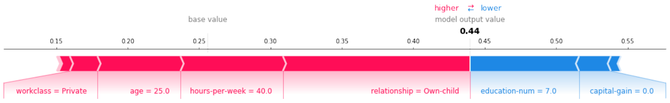

Fig. 3. Force plot of a Decision Tree explained with Tree SHAP using 50 samples

library implementation, Hoeffding trees assume a stationary distribution (it is unclear if this dataset is stationary), convergence guarantees are only asymptotic so the convergence could be too slow to be observed here.

VI. Conclusion

VII. Reproducibility Considerations

A. Software Libraries:

The software library versions we used (read out using pip –freeze) are documented in table I.
All experiments were conducted on a single personal computer. The hardware specs can be found in table II.

Acknowledgment

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression “one of us (R. B. G.) thanks . . .”. Instead, try “R. B. G. thanks. . .”. Put sponsor acknowledgments in the unnumbered footnote on the first page.

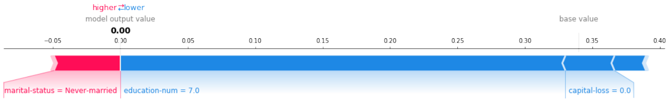

Fig. 4. Force plot of a Hoeffding Tree explained with Kernel SHAP using 50 samples

TABLE I
Software Library Versions

Software library	**Version**
Python	3.7.4
anaconda-client	1.7.2
anaconda-navigator	1.9.7
anaconda-project	0.8.3
conda	4.7.12
conda-build	3.18.9
conda-package-handling	1.6.
0 conda-verify	3.4.2
ipykernel	5.1.2
ipython	7.8.0
ipython-genutils	0.2.0
jupyter	1.0.0
jupyter-client	5.3.3
jupyter-console	6.0.0
jupyter-core	4.5.0
jupyterlab	1.1.4
jupyterlab-server	1.0.6
matplotlib	3.1.1
notebook	6.0.1
numpy	1.16.5
pandas	0.25.1
scikit-image	0.15.0
scikit-learn	0.21.3
scikit-multiflow	0.5.0
scipy	1.3.1
shap	0.35.0
sklearn	0.0
torch	1.5.0+cu101
torchvision	0.6.0+cu101

TABLE II
Hardware Specifications:

Component	**Model**
Processor	Intel Core i7-6700 CPU
RAM	16 GB Corsair Vengeance LPX DDR4-2400
GPU	NVIDIA GeForce GTX 980
Motherboard	ASUS Z170 Pro Gaming
Operating system	Microsoft Windows 10 Education N
Hard Drive	SSD Crucial MX200

References

[1] Scott M. Lundberg, Gabriel G. Erion, Su-In Lee: Consistent Individualized Feature Attribution for Tree Ensembles. CoRR abs/1802.03888 (2018).

[2] Scott M. Lundberg, Su-In Lee: A Unified Approach to Interpreting Model Predictions. NIPS 2017: 4765-4774.

[3] Pedro M. Domingos, Geoff Hulten: Mining high-speed data streams. KDD 2000: 71-80

[4] Marco Túlio Ribeiro, Sameer Singh, Carlos Guestrin: ”Why Should I Trust You?”: Explaining the Predictions of Any Classifier. KDD 2016: 1135-1144.

[5] Avanti Shrikumar, Peyton Greenside, Anshul Kundaje: Learning Important Features Through Propagating Activation Differences. ICML 2017: 3145-3153.

[6] Alexander Binder, Grégoire Montavon, Sebastian Bach, Klaus-Robert Müller, Wojciech Samek: Layer-wise Relevance Propagation for Neural Networks with Local Renormalization Layers. CoRR abs/1604.00825 (2016).

[7] Anupam Datta, Shayak Sen, Yair Zick: Algorithmic Transparency via Quantitative Input Influence: Theory and Experiments with Learning Systems. IEEE Symposium on Security and Privacy 2016: 598-617.

[8] Erik Strumbelj, Igor Kononenko: Explaining prediction models and individual predictions with feature contributions. Knowl. Inf. Syst. 41(3): 647-665 (2014).

[9] Leo Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone: Classification and Regression Trees. Wadsworth 1984, ISBN 0-534-98053-8.

[10] Tianqi Chen, Carlos Guestrin: XGBoost: A Scalable Tree Boosting System. KDD 2016: 785-794.

[11] L Auret, C Aldrich: Empirical comparison of tree ensemble variable importance measures. Chemometrics and Intelligent Laboratory Systems, 2011 - 105, 2(2011), 157-170

[12] Ando Sabbas. 2014. Interpreting random forests. http://blog.datadive.net/interpreting-random-forests/ Accessed: 2020-06-15

[13] https://github.com/slundberg/shap Accessed: 2020-06-15

[14] Katsushige Fujimoto, Ivan Kojadinovic, Jean-Luc Marichal: Axiomatic characterizations of probabilistic and cardinal-probabilistic interaction indices. Games Econ. Behav. 55(1): 72-99 (2006).

[15] Ronny Kohavi and Barry Becker: Adult Dataset, UCI Machine Learning Repository http://archive.ics.uci.edu/ml

[16] https://github.com/slundberg/shap/ Accessed: 2020-07-01