Explaining Predictions with Shapley Values in R

by Brandon M. Greenwell

Abstract An abstract of less than 150 words.

Introduction

The *Shapley value* (Shapley 2016) is an idea from coalitional/cooperative game theory. In a coalitional game (i.e., a competetive game between groups of players called *coalitions*), assume we have p players that form a grand coalition (S) worth a certain payout (Δ_S). Suppose we also know how much any smaller coalition ($Q \subseteq S$) (i.e., any subset of p players) is worth (Δ_Q). The goal is to distribute the total payout Δ_S to the individual p players in a "fair" way; that is, so that each player receives their "fair" share. The Shapley value is one such solution and the only one that uniquely satisfies a particular set of "fairness properties."

Let v be a *characteristic function* (or mapping) that assigns a value to each subset of players; in particular, $v : 2^p \to \mathbb{R}$, where $v(S) = \Delta_S$, $v(\emptyset) = 0$, and $\emptyset$ is the empty set (i.e., zero players). Let $\phi_i(v)$ be the contribution (or portion of the total payout) attributed to player i in a particular game with total payout $v(S) = \Delta_S$. The Shapley values $\{\phi_i\}_{i=1}^p$ satisfy the following four properties:

1. Efficiency: $\sum_{i=1}^p \phi_i(v) = \Delta_S$.
2. Null player: $\forall W \subseteq S \setminus \{i\} : \Delta_W = \Delta_{W \cup \{i\}} \implies \phi_i(v) = 0$.
3. Symmetry: $\forall W \subseteq S \setminus \{i, j\} : \Delta_{W \cup \{i\}} = \Delta_{W \cup \{j\}} \implies \phi_i(v) = \phi_j(v)$.
4. Linearity: If v and w are functions describing two coalitional games, then $\phi_i(v + w) = \phi_i(v) + \phi_i(w)$.

The above properties can be interpreted as follows: 1) the individual player contributions sum to the total payout, hence, are implicitly normalized; 2) if a player does not contribute to any coalition they receive a payout of zero; 3) if two players have the same impact across all coalitions, they receive equal payout; and 4) the local contributions are additive across different games.

Shapley (2016) showed that the unique solution satisfying the above properties is given by

$$\phi_i(x) = \frac{1}{p!} \sum_{\mathcal{O} \in \pi(p)} \left[v\left(S^{\mathcal{O}} \cup i\right) - v\left(S^{\mathcal{O}}\right) \right], \quad i = 1, 2, \ldots, p, \tag{1}$$

where $\mathcal{O}$ is a specific permutation of the players indices $\{1, 2, \ldots, p\}$, $\pi(p)$ is the set of all such permutations of size p, and $S^{\mathcal{O}}$ is the set of players joining the coalition before player i.

In other words, the Shapley value is the average marginal contribution of a player across all possible coalitions in a game. Another way to interpret Equation (1) is as follows. Imagine the coalitions (i.e., subsets of players) being formed one player at a time (which can happen in different orders), with the i-th player demanding a fair contribution/payout of $v\left(S^{\mathcal{O}} \cup i\right) - v\left(S^{\mathcal{O}}\right)$. The Shapley value for player i is given by the average of this contribution over all possible permutations in which the coalition can be formed.

A simple example may help clarify the main ideas. Suppose three friends (players)—Alex, Brad, and Brandon—decide to go out for drinks after work (the game). They shared a few pitchers of beer, but nobody payed attention to how much each person drank (collaborated). What's a fair way to split the tab (total payout)? Suppose we knew the follow information, perhaps based on historical data:

- If Alex drank alone, he'd only pay \$10.
- If Brad drank alone, he'd only pay \$20.
- If Brandon drank alone, he'd only pay \$10.
- If Alex and Brad drank together, they'd only pay \$25.
- If Alex and Brandon drank together, they'd only pay \$15.
- If Brad and Brandon drank together, they'd only pay \$13.
- If Alex, Brad, and Brandon drank together, they'd only pay \$30.

Note that $S = \{\text{Alex}, \text{Brad}, \text{Brandon}\}$ and that $\Delta_S = \$30$. With only three players, we can enumerate all possible coalitions. In Table 1, we list out all possible permutations of the three players along with the marginal contribution of each. Take the first row, for example. In this particular permutation, we start with Alex. We know that if Alex drinks alone, he'd spend \$10, so his marginal contribution by entering first is \$10. Next, we assume Brad enters the coalition. We know that if Alex and Brad drank together, they'd pay a total of \$25, leaving \$15 left over for Brad's marginal contribution. Similarly, if Brandon joins the party last, his marginal contribution would be only \$5 (the difference between \$30 and \$25). The Shapley value for each player is the average marginal contribution across all six possible permutations (these are the column averages reported in the last row).

Table 1: Marginal contribution for each permutation of the players Alex, Brad, Brandon (i.e., the order in which they arrive). The Shapley contribution is the average marginal contribution cross all permutations. (Notice how each row sums to the total bill of \$30.)

Permutation/order of players	Alex	Brad	Brandon
Alex, Brad, Brandon	\$10.00	\$15.00	\$5.00
Alex, Brandon, Brad	\$10.00	\$15.00	\$5.00
Brad, Alex, Brandon	\$5.00	\$20.00	\$5.00
Brad, Brandon, Alex	\$10.00	\$20.00	\$0.00
Brandon, Alex, Brad	\$5.00	\$15.00	\$10.00
Brandon, Brad, Alex	\$17.00	\$3.00	\$10.00
Shapley contribution:	\$9.50	\$14.67	\$5.83

In this example, Brandon would get away with the smallest payout (i.e., have to pay the smallest portion of the total tab). The next time the bartender asks how you want to split the tab, whip out a pencil and do the math! In the next section, we'll show how the same idea can be used to help quantify the contribution each feature value makes to its corresponding prediction in a machine learning model.

Shapley values for explaining predictions

Štrumbelj and Kononenko (2014) suggested using the Shapley value (1) to help explain predictions from a machine learning model. In the context of machine learning:

- a game is represented by the prediction task for a single observation $\boldsymbol{x} = (x_1, x_2, \ldots, x_p)$ (i.e., there are p features in total);
- the total payout/worth (Δ_S) for $\boldsymbol{x}$ is the prediction for $\boldsymbol{x}$, denoted $\hat{f}(\boldsymbol{x})$, minus the average prediction for all training observations (call this the baseline prediction, which we'll denote by $\bar{f}$);
- the players are the individual feature values of $\boldsymbol{x}$ that collaborate to receive the payout (i.e., $\hat{f}(\boldsymbol{x}) - \bar{f}$).

From the last point, it's important to note that Shapley explanations are not trying to quantify the contribution each feature value in $\boldsymbol{x}$ makes to its prediction $\hat{f}(\boldsymbol{x})$, but rather to the quantity $\hat{f}(\boldsymbol{x}) - \bar{f}$, the difference between its prediction and the baseline. This seems to be a common source of confusion in interpreting a set of Shapley explanations from a given model.

In the following sections, we'll discuss several popular ways to compute Shapley values in practice.

Choice of characteristic function v

The challenge of using Shapley values for the purpose of explaining predictions is in defining the functional form of v. As discussed in H. Chen et al. (2020), there are several ways to do this. However, since we are primarily interested in understanding how much each feature contributed to a particular prediction, v is typically related to a conditional expectation of the model's prediction. H. Chen et al. (2020) make the distinction between two possibilities, each of which differs in their conditioning argument. The Shapley value implementations discussed in this paper (e.g., MC SHAP and Tree SHAP) rely on what H. Chen et al. (2020) call the *interventional conditional expectation*, which can be expressed using Pearl's *do* $(\cdot)$ operator (Pearl 2009):

$$\begin{aligned} v(S) &= \mathbb{E}\left[f\left(\boldsymbol{x}_S, \boldsymbol{x}_{S^c}\right) | do\left(\boldsymbol{x}_S\right)\right] \\ &= \int f\left(\boldsymbol{x}_S, \boldsymbol{x}_{S^c}\right) p\left(\boldsymbol{x}_{S^c}\right) d\boldsymbol{x}_{S^c}, \end{aligned} \tag{2}$$

where S^c is the complement of S, $\boldsymbol{x}_S$ and $\boldsymbol{x}_{S^c}$ are the set of features in S and S^c, respectively, and $p\left(\boldsymbol{x}_{S^c}\right)$ is the joint probability density of $\boldsymbol{x}_{S^c}$. Equation @ref((eq:ice) can be interpreted as the expected value of $f\left(\boldsymbol{x}\right)$ given some intervention on the features in S, which assumes independence between $\boldsymbol{x}_S$ and $\boldsymbol{x}_{S^c}$; a similar assumption is also used in the construction of *partial dependence plots* (Friedman 2001), with the connection to Pearl's $do\left(\cdot\right)$ operator established in Zhao and Hastie (2021). Shapley values based on this formulation of v are referred to as *interventional Shapley values* (H. Chen et al. 2020). The various Shapley value algorithms discussed over the next several sections fall under this form.

The following sections detail several algorithms for estimating Shapley explanations in practice.

Monte Carlo (MC) SHAP: approximate Shapley values via Monte Carlo simulation

Computing the exact Shapley value is computationally infeasible, even for moderately large p. To that end, Štrumbelj and Kononenko (2014) suggest a Monte Carlo approximation, which we'll call SampleSHAP[1], that assumes independent features[2]. Their approach is described in Algorithm 1 below.

Algorithm 1: Approximating the i-th feature's contribution to $f\left(\boldsymbol{x}\right)$.

1. For $j = 1, 2, \ldots, R$:
    a. Select a random permutation $\mathcal{O}$ of the feature values $x_1, x_2, \ldots, x_p$.
    b. Select a random instance $\boldsymbol{w}$ from the set of training observations $\boldsymbol{X}$.
    c. Construct two new instances as follows:
        - $\boldsymbol{b}_1 = \boldsymbol{x}$, but all the features in $\mathcal{O}$ that appear after feature x_i get their values swapped with the corresponding values in $\boldsymbol{w}$.
        - $\boldsymbol{b}_2 = \boldsymbol{x}$, but feature x_j, as well as all the features in $\mathcal{O}$ that appear after x_j, get their values swapped with the corresponding values in $\boldsymbol{w}$.
    d. Compute the difference in predictions: $\phi_{ij}\left(\boldsymbol{x}\right) = f\left(\boldsymbol{b}_1\right) - f\left(\boldsymbol{b}_2\right)$.
2. Aggregate the results: $\phi_i\left(\boldsymbol{x}\right) = \sum_{j=1}^{R} \phi_{ij}\left(\boldsymbol{x}\right) / R$.

Here, A single estimate of the contribution of x_i to $f\left(\boldsymbol{x}\right) - \bar{f}$ is nothing the more than the difference between two predictions, where each prediction is based on a set of "Frankenstein instances"[3] that are constructed by swapping out values between the instance being explained ($\boldsymbol{x}$) and an instance selected at random from the training data. To help stabilize the results, the procedure is repeated a large number, say, R, times, and the results averaged together. Note that Algorithm 1 can be parallelized across features or MC repetitions.

If there are p features and m instanced to be explained, this requires $2 \times R \times p \times m$ predictions (or calls to a scoring function). In practice, this can be quite computationally demanding, especially since R needs to be large enough to produce good approximations to each $\phi_i\left(x\right)$. How large does R need to be to produce accurate explanations? It depends on the variance of each feature in the observed training data, but typically $R \in [30, 100]$ will suffice. In a later section, we'll discuss a particularly optimized implementation of Algorithm 1 that only requires $2mp$ calls to a scoring function.

Even with certain optimizations or parallel processing, MC SHAP can be computationally prohibitive if you need to explain a large number of predictions Fortunately, you often only need to explain a handful of predictions, for example the most extreme predictions. However, generating individual explanations for the entire training set, or a large enough sample thereof, can be useful for generating aggregated (i.e., *global*) model summaries, like Shapley-based variable importance plots (Scott M. Lundberg et al. 2020).

A simple R implementation of Algorithm 1 is given below. Here, `obj` is a fitted model with scoring function `f()` (e.g., `predict()`), `R` is the number of MC repetitions to perform, `feature` gives the name of the corresponding feature in `x` to be explained, and `X` is the training set of features.

[1] **FIXME:** Make a note on the technical use of the term SHAP, and how we're being loose with the terminology here.

[2] While SampleSHAP, along with many other common Shapley value procedures, assumes independent features, several arguments can be made in favor of this assumption; see, for example, H. Chen et al. (2020) and the references therein.

[3] The terminology used here takes inspiration from Molnar (2019) (p. 231).

```
sample.shap <- function(f, obj, R, x, feature, X) {
  phi <- numeric(R)  # to store Shapley values
  N <- nrow(X)   # sample size
  p <- ncol(X)   # number of features
  b1 <- b2 <- x  # initialize new instances
  for (m in seq_len(R)) {
    w <- X[sample(N, size = 1), ]  # sample random obs from X     # (step 2. b.)
    ord <- sample(names(w))  # random permutation of features     #
    swap <- ord[seq_len(which(ord == feature) - 1)]               #
    b1[swap] <- w[swap]                                          # (step 1. c.)
    b2[c(swap, feature)] <- w[c(swap, feature)]                  # (step 1. c.)
    phi[m] <- f(obj, newdata = b1) - f(obj, newdata = b2)        # (step d.)
  }                                                              #
  mean(phi)  # return approximate feature contribution           # (step 2.)
}
```

Linear SHAP: shapley values from additive linear models

First, lets discuss how a feature's value contributes to a prediction $f(\boldsymbol{x})$ in an additive linear model with independent features. That is, let's assume for a moment that f takes the form

$$f(\boldsymbol{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$$

Recall that the contribution of x_i (the i-th feature component of $\boldsymbol{x}$) to the prediction $f(\boldsymbol{x})$ is the difference between $f(\boldsymbol{x})$ and the expected prediction if the i-th feature's value were not known:

$$\begin{aligned}\phi_i(\boldsymbol{x}) &= \beta_0 + \cdots + \beta_i x_i + \cdots + \beta_p x_p \\ &\quad - \left(\beta_0 + \cdots + \beta_i \bar{x}_i + \cdots + \beta_p \bar{x}_p\right), \\ &= \beta_i (x_i - \bar{x}_i)\end{aligned}$$

where, for example, $\bar{x}_i$ corresponds to the sample mean of the i-th features values in the training sample. For a proof, see Aas, Jullum, and Løland (2020). The quantity $\phi_i(\boldsymbol{x})$ is also referred to as the *situational importance* of x_i (Achen 1982).

Note that if you're using R, then $\beta_i(x_i - \bar{x}_i)$ is exactly what's returned by R's `predict()` method when applied to `lm/glm` models, provided you specify `type = "terms"`; see `?predict.lm` for details.

Kernel SHAP: approximate Shapley values using kernel approximations

Kernel SHAP Lundberg and Lee (2017) uses a linear regression-based approximation to estimate Shapley values from a given model. It is model-agnostic in the sense that it can be applied in the same way to any type of supervised learning model.

In the Kernel SHAP formulation, the computation is represented as a linear model, with a specific Shapley Kernel.

Kernel SHAP samples coalitions $z'_k \in \{0,1\}^M$, with $k \in \{1,\ldots,K\}$, where '0' signifies that a feature is absent and '1' that it is present. A function $h_x : Z \mapsto X$ maps coalitions to the feature space, making it possible to get the predictions for the sampled coalitions: $f(h_x(z'_k))$. A dataset is generated by sampling coalitions and computing their model predictions. A weighted linear model is then fitted with the coalition vector as features and the model predictions as target. The weight used is the kernel:

$$\pi_x(z') = \frac{(M-1)}{\binom{M}{|z'|}|z'|(M-|z'|)}$$

where M is the maximum coalition size (for tabular data the number of features) and $|z'|$ is the number of 1's in the coalition vector, or the number of features that are present. The function h_x maps elements of the coalition vector with a 1 to the original feature vector, and the elements with 0's to the respective feature values of a randomly sampled data point. The estimated coefficients in this weighted linear model can be interpreted as Shapley value estimates. The accuracy of the estimate depends on the size of the coalitions that are sampled. Like SampleSHAP and LinearSHAP, Kernel SHAP assumes independent features. The Kernel SHAP supposedly requires less computational power than SampleSHAP to obtain a similar approximation accuracy (i.e., fewer replications).

NOTES:

(Christoph, this was my initial "thought dump", feel free to delete or use.)

- Kernel SHAP, or at least it's implementation in `shap`, distributes the number of replications unevenly across the different features. Apparently, features with higher variance are attributed more replications.
- Extended to handle dependent features in Aas et al. (2020) and is available in the R package **shapr** (Sellereite et al., 2021).
- Kernel SHAP (Scott M. Lundberg and Lee 2017b) uses a specially-weighted local linear regression to estimate SHAP values for any model. Unlike MC SHAP...
- SHAP decomposes a prediction into

$$f(x) = \phi_0 + \sum_{i=1}^{p} \phi_i, \tag{3}$$

  where $f(x)$, $\phi_0 = \mathbb{E}[f(x)]$, p is the number of predictors, and ϕ_i is the contribution of the i-th feature. It should be noted that ϕ_i ($i = 1, 2, \ldots, p$) depend on the observation x, whereas ϕ_0 is constant. From Equation **FIXME:** eq:shap, it should be clear that $\sum_{i=1}^{p} \phi_i = f(x) - \phi_0$. In other words, SHAP values help to explain the difference between a particular prediction and the global average prediction. The quantity ϕ_0 is often referred to as the baseline prediction and is estimated in practice using the average prediction across all N training observations: $\bar{y}_{trn} = \sum_{i=1}^{N} y_i / N$
- Covert and Lee (2021) (and the corresponding GitHub repo: https://github.com/iancovert/shapley-regression) offer a nice discussion on Kernel SHAP and some insight into its properties.
- Discuss Shapley values as the solution to a weighted least squares problem; see Charnes et al. (1988) for details.

Unlike sampling-based approaches (e.g., MC SHAP), Kernel SHAP does not provide estimates of uncertainlty. An improvemed version of Kernel SHAP was proposed in Covert and Lee (2021). This version is unbiased and has better convergence.

Tree SHAP: efficient Shapley values for tree ensembles

FIXME: Need to find the right balance of details and complexity here.

NOTES:

Tree SHAP assumes "less" feature independence in the sense that it accounts for some of the dependence, but not all (Aas, Jullum, and Løland 2020).

Only applicable to tree-based models, and implemented for only a few algorithms (e.g., XGBoost and LightGBM).

Implementations in R

Probably the first, and most widely used implementation of Shapley explanations is the Python **shap** library (Scott M. Lundberg and Lee 2017b), which provides a Python implementation of SampleSHAP, KernelSHAP, TreeSHAP, and a few other model-specific Shapley methods (e.g., DeepSHAP, which is provides approximate Shapley values for deep learning models).

The **shapper** package (**R-shapper?**) provides an R interface to the Python **shap** library using **reticulate** (**?**); however, it currently only supports Kernel SHAP (**shap** itself additionally supports MC SHAP, Tree SHAP, Linear SHAP, as well as various other model-specific Shapley explanation methods).

There are several R packages available for computing Shapley-based feature contributions. You can perform a quick search for CRAN packages related to Shapley value using the **pkgsearch** (Csárdi and Salmon 2020):

```
pkgsearch::ps("Shapley")  # set `format = "long"` for more detailed results
```

```
#> - "Shapley" ----------------------------------- 16 packages in 0.008 seconds -
#>   #     package             version by                          @ title
#>   1 100 fastshap            0.0.7   Brandon Greenwell        10M Fast Appr...
```

```
#>   2  51 shapr                 0.2.0   Martin Jullum              2y Predictio...
#>   3  51 ShapleyValue          0.2.0   Jingyi Liang               1y Shapley V...
#>   4  22 vip                   0.3.2   Brandon Greenwell          2y Variable ...
#>   5  13 kappalab              0.4.7   Ivan Kojadinovic           7y Non-Addit...
#>   6  13 iml                   0.11.1  Christoph Molnar          29d Interpret...
#>   7  11 GameTheoryAllocation 1.0      Alejandro Saavedra-Nieves  6y Tools for...
#>   8  10 shapper               0.1.3   Szymon Maksymiuk           2y Wrapper o...
#>   9  10 SHAPforxgboost        0.1.1   Yang Liu                   2y SHAP Plot...
#>  10   9 matchingR            1.3.3   Jan Tilly                  1y Matching ...
```

While we won't demonstarte use of the package, it's worth point readers to the **shapr** package (Sellereite and Jullum, 2019). As previously discussed, one drawback of traditional Shapley values (like the ones computed by the MC SHAP procedure) is the assumption of independent features (an assumption made by many IML procedures, in fact). To that end, the **shapr** package implements Shapley explanations that can account for the dependence between features (Aas et al., 2019), resulting in significantly more accurate approximations to the Shapley values. The package also includes an implementation of KernelSHAP that's consistent with the **shap** package for Python.

Tree SHAP has been directly incorporated into most implementations of XGBoost (Chen and Guestrin, 2016) (including **xgboost** (Chen et al., 2020)), CatBoost (**?**), and LightGBM (Ke et al., 2017). Both **fastshap** (Greenwell, 2020) and **SHAPforxgboost** (**?**) provide an interface to **xgboost**'s TreeSHAP implementation.

The remainder of this article will focus applying Shapley values to machine learning using a handful of packages: **fastshap** (Greenwell 2020), **iml** (Molnar and Schratz 2020), **iBreakDown** (Biecek et al. 2020), and **lightgbm** (Shi et al. 2022). The first three packages all prvide an implementation of MC SHAP (i.e., Algorithm 1), while the latter includes an implementation of Tree SHAP; note that **xgboost** (T. Chen et al. 2020), an efficient boosting library similar to `lightgbm`, provides similar Tree SHAP functionality.

fastshap provides an efficient implementation of SampleSHAP and makes it a viable option for explaining the predictions from model's where efficient model-specific Shapley methods do not exist or are not yet implemented.

The **iml** package provides the `Shapley()` function, which is a direct implementation of Algorithm 1. It is written in **R6** (Chang 2021). Moreover, the **iml** package provides a standard interface to several other interpretable machine learning (IML) algorithms, whence the package name.

Package **iBreakDown** implements a general approach to explaining the predictions from supervised models, called *Break Down* (Gosiewska and Biecek 2019). MC SHAP values can be computed as a special case from random Break Down profiles; see `?iBreakDown::shap` for details.

While several of these packages provide their own plotting function for visualizing the output, the **shapviz** package (Mayer 2022) provides a generic set of function for plotting Shapley explanations with direct support for a number of packages (including **fastshap**, **lightgbm**, **xgboost**, and **shapr**, to name a few). The package is general enough and can be applied to any set of Shapley explanations stored in an ordinary R matrix.

Example: explaining survival on the Titanic

In this section, we'll look at a simple example related to predicting survival on the ill-fated Titanic. We'll use this as an opportunity to introduce all four packages mentioned above. To start, we'll load…

```
library(fastshap)

# Use one of fastshap's imputed versions of the Titanic data
head(titanic <- titanic_mice[[1L]])
```

```
#>   survived pclass   age    sex sibsp parch
#> 1      yes      1 29.00 female     0     0
#> 2      yes      1  0.92   male     1     2
#> 3       no      1  2.00 female     1     2
#> 4       no      1 30.00   male     1     2
#> 5       no      1 25.00 female     1     2
#> 6      yes      1 48.00   male     0     0
```

While **lightgbm** now supports categorical features, it's easier just to re-encode binary variables as 0/1, which we do below. We then construct a matrix (`X`) containing only the feature columns before

calling `lightgbm()` too fit a model using log loss (the number of trees, or `nrounds`, was found using 5-fold cross-validation vi the `lgb.cv()` function):

```
library(lightgbm)

# Re-encode binary variables as 0/1
titanic$survived <- ifelse(titanic$survived == "yes", 1, 0)
titanic$sex <- ifelse(titanic$sex == "male", 1, 0)

# Matrix of only predictor values
X <- data.matrix(subset(titanic, select = -survived))

params <- list(
  num_leaves = 10L,
  learning_rate = 0.1,
  objective = "binary"
)

set.seed(1420)  # for reproducibility
bst <- lightgbm(X, label = titanic$survived, params = params, nrounds = 45,
                verbose = 0)

#> [LightGBM] [Warning] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000936 seconds.
#> You can set `force_col_wise=true` to remove the overhead.
```

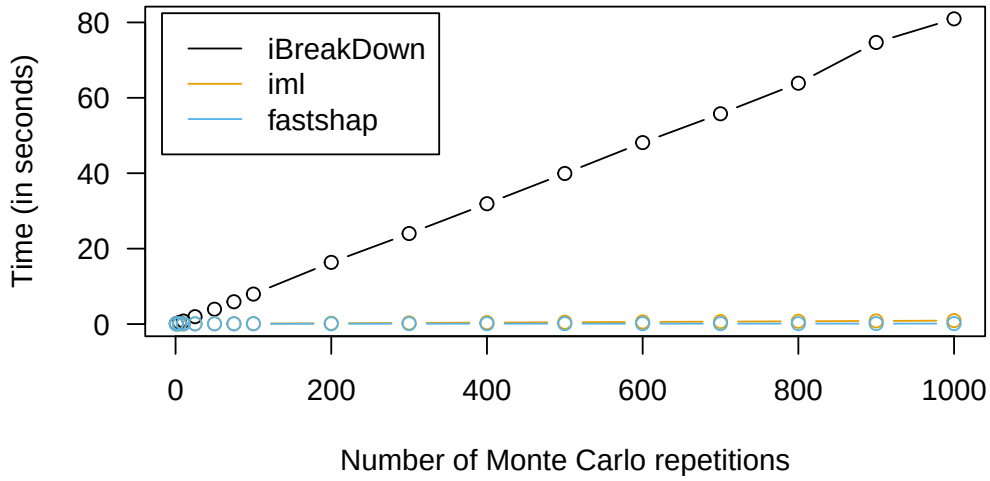
To illustrate the simplest use of Shapley values for quantifying feature contributions, we need an observation to predict. While we can use any observation from the training set, we'll construct an observation for a new passenger. Everyone, meet Jack:

```
jack.dawson <- data.matrix(data.frame(
  #survived = 0L,  # in case you haven't seen the movie
  pclass = 3L,     # third-class passenger
  age = 20.0,      # twenty years old
  sex = 1L,        # male
  sibsp = 0L,      # no siblings/spouses aboard
  parch = 0L       # no parents/children aboard
))  # lightgbm doesn't like data frames
```

Note that **fastshap**, **iml**, and **iBreakDown** typically require a predefined prediction wrapper; that is, a simple function that tells each package how to extract the appropriate predictions from the fitted model. In this case, for comparison with Tree SHAP, our prediction wrapper will return the predictions on the raw (i.e., logit) scale:

```
pfun <- function(object, newdata) {  # prediction wrapper
  predict(object, data = data.matrix(newdata), rawscore = TRUE)
}

# Compute Jack's predicted likelihood of survival
(jack.logit <- pfun(bst, newdata = jack.dawson))  # logit scale

#> [1] -1.732893

(jack.prob <- plogis(jack.logit))  # probability scale

#> [1] 0.1502179

ex.lightgbm <- predict(bst, data = jack.dawson, predcontrib = TRUE)
colnames(ex.lightgbm) <- c(colnames(X), "baseline")
ex.lightgbm

#>          pclass        age        sex      sibsp       parch   baseline
#> [1,] -0.4635056 0.06744438 -0.8439261 0.08515607 -0.01917919 -0.5588822
```

```
sum(ex.lightgbm)  # since baseline is included, this should some to prediction

#> [1] -1.732893
```



In this example, for $R = 1000$ MC repetitions, **fastshap** is roughly `7.3902439` times faster than **iml**, and nearly `658.1300813` times faster than **iBreakDown**.

Example: visualizing global explanations with fastshap and shapviz

Example: predicting/explaining ALS progression

In this example, we'll do something a bit more interesting. Rather than demonstrating the use of `fastshap` on an ordinary prediction task, let's use it to help explain the output from a probabilistic regression framework that's (currently) only available in Python. To illustrate, we'll look at a brief example using the ALS data from Efron and Hastie (2016, p. 349). A description of the data, along with the original source and download instructions, can be found at `https://web.stanford.edu/~hastie/CASI/`}.

The data concern $N = 1,822$ observations on *amyotrophic lateral sclerosis* (ALS or Lou Gehrig's disease) patients. The goal is to predict ALS progression over time, as measured by the slope (or derivative) of a functional rating score (`dFRS`), using 369 available predictors obtained from patient visits. The data were originally part of the DREAM-Phil Bowen ALS Predictions Prize4Life challenge. The winning solution (Küffner et al. 2015) used a tree-based ensemble quite similar to a *random forest* (Breiman 2001), while Efron and Hastie (2016) (Chap. 17) analyzed the data using a *gradient boosted tree ensemble* (Friedman 2001, 2002).

Many classification tasks are inherently probabilistic. For example, probability forests (Malley et al. 2012) can be used to obtain consistent probability estimates for the different class outcomes (i.e., $Pr(y = j|\boldsymbol{x})$). Regression tasks, on the other hand, are typically not probabilistic and the predictions correspond to some location estimate of $y|\boldsymbol{x}$; that is, the distribution of y conditional on a set of predictor values $\boldsymbol{x}$. For instance, the terminal nodes in a regression tree—which are used to compute fitted values and predictions—provide an estimate of the conditional mean $E(y|\boldsymbol{x})$. Often, it is of scientific interest to know about the probability of specific events conditional on a set of features, rather than a single point estimate like $E(y|\boldsymbol{x})$. In the ALS example, rather than using an estimate of the conditional mean $\hat{f}(\boldsymbol{x}) = \hat{E}(\texttt{dFRS}|\boldsymbol{x})$ to predict ALS progression for a new patient, it might be more useful to estimate $Pr(\texttt{dFRS} < c|\boldsymbol{x})$, for some constant c. This is where probabilistic regression/forecasting comes in.

Probabilistic regression models provide estimates of the entire probability distribution of the response conditional on a set of predictors, denoted $\mathcal{D}_{\boldsymbol{\theta}}(y|\boldsymbol{x})$, where $\boldsymbol{\theta}$ represents the parameters of the conditional distribution. For example, the normal distribution has $\boldsymbol{\theta} = (\mu, \sigma)$; examples include *generalized additive models for shape, scale, and location* (GAMLSS) (Rigby and Stasinopoulos 2005), *Bayesian additive regression trees* (BART) (Chipman, George, and McCulloch 2010), and Bayesian deep

learning. While several approaches to probabilistic regression exist, many of them are inflexible (e.g., GAMSLSS), computationally expensive (e.g., BART), or inaccessible to non-experts (e.g., Bayesian deep learning) (Duan et al. 2020). *Natural gradient boosting* (NGBoost) extends the simple ideas of gradient boosting to probabilistic regression by treating the parameters $\boldsymbol{\theta}$ as targets for a multiparameter boosting algorithm similar to gradient boosting. We say "multiparameter" because NGBoost fits a separate model for each parameter at every iteration.

The "natural" in "natural gradient boosting" refers to the fact that NGBoost uses something called the *natural gradient*, as opposed to the ordinary gradient. The natural gradient provides the direction of steepest descent in *Riemannian space*; this is necessary since gradient descent in the parameter space is not gradient descent in the distribution space because distances don't correspond. The important thing to remember is that NGBoost approximates the gradient of a proper scoring rule—similar to a loss function, but for predicted probabilities and probability distributions of the observed data—as a function of $\boldsymbol{\theta}$. Compared to alternative probabilistic regression methods, NGBoost is fast, flexible, scalable, and easy to use. NGBoost is available in the `ngboost` package for Python. For more info, check out the NGBoost GitHub repository at https://github.com/stanfordmlgroup/ngboost.

To start, we'll read in the data from the companion website to Efron and Hastie (2016). Note that the data already include an indicator for training and validation, so we'll go ahead and split the data into train/validation sets:

```
als <- read.table("https://web.stanford.edu/~hastie/CASI_files/DATA/ALS.txt",
                  header = TRUE)

# Split into train/test sets
als.trn <- als[!als$testset, -1]  # train
als.val <- als[als$testset, -1]  # validation

# Print dimensions
dim(als.trn)

#> [1] 1197  370

dim(als.val)

#> [1] 625 370
```

Next, we'll use **reticulate** (**FIXME:** need citation) to load the Python `ngboost` module:

```
library(reticulate)

ngboost <- import("ngboost")  # requires installation of ngboost

# Construct an NGBoost regressor object
ngb <- ngboost$NGBRegressor(
  Dist = ngboost$distns$Normal,
  n_estimators = 2000L,
  learning_rate = 0.01,
  verbose_eval = 0,
  random_state = 1601L
)
```

In the next chunk, we call the `ngb` object's fit()' method to actually train the model (the validation set is used with early stopping to determine the optimal number of trees in the ensemble):

```
X.trn <- subset(als.trn, select = -dFRS)  # features only
X.val <- subset(als.val, select = -dFRS)  # features only

# Train the model
ngb$fit(
  X = X.trn,
  Y = als.trn$dFRS,
  X_val = X.val,
  Y_val = als.val$dFRS,
  early_stopping_rounds = 5L
)
```

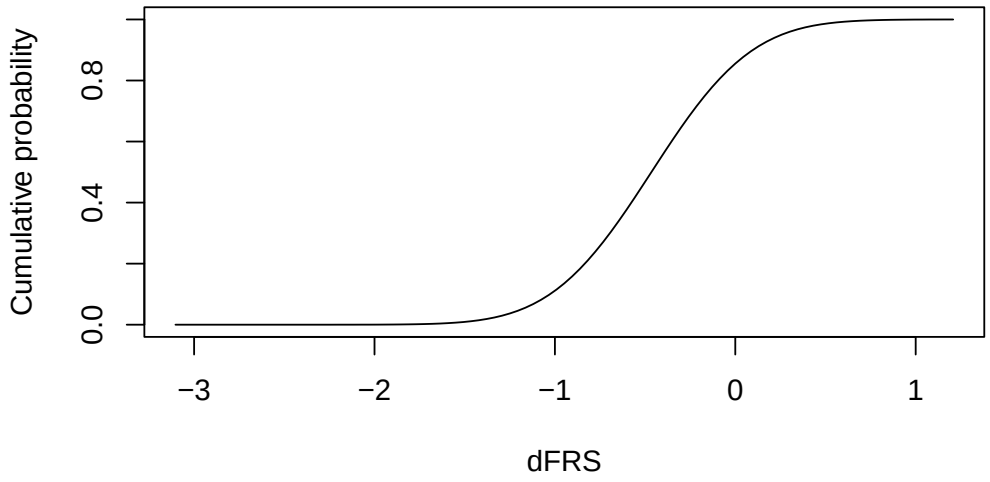


Figure 1: ABC.

```
#> NGBRegressor(n_estimators=2000,
#>              random_state=RandomState(MT19937) at 0x7F5222D48040,
#>              verbose_eval=0.0)

ngb$predict(X.val[1, ])

#> [1] -0.4649206

(params <- ngb$pred_dist(X.val[1, ])$params)

#> $loc
#> [1] -0.4649206
#>
#> $scale
#> [1] 0.4386185
```

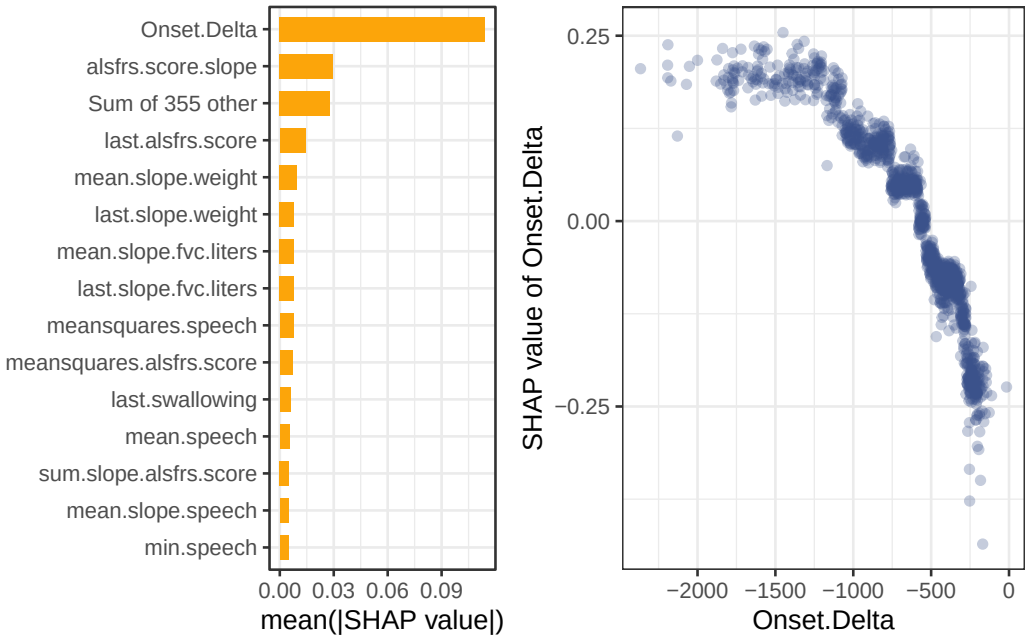
The code chunk below generates a plot of the estimated cumulative probability density function (i.e., $Pr\left(\texttt{dFRS} <= t\right)$) for the first observation in the validation set; see Figure 1.

```
plot(function(x) pnorm(x, mean = params$loc, sd = params$scale),
     from = min(als$dFRS), to = max(als$dFRS),
     xlab = "dFRS", ylab = "Cumulative probability")
```

Since `ngboost` is built on top of `sklearn` (Pedregosa et al. 2011), we can actually use the Python `shap` package (Scott M. Lundberg and Lee 2017a) to create efficient explanations for the model. Below, we call the `TreeExplainer()` method on the training set and generate global some global interpretations: a variable importance plot, and a *SHAP dependence plot*:

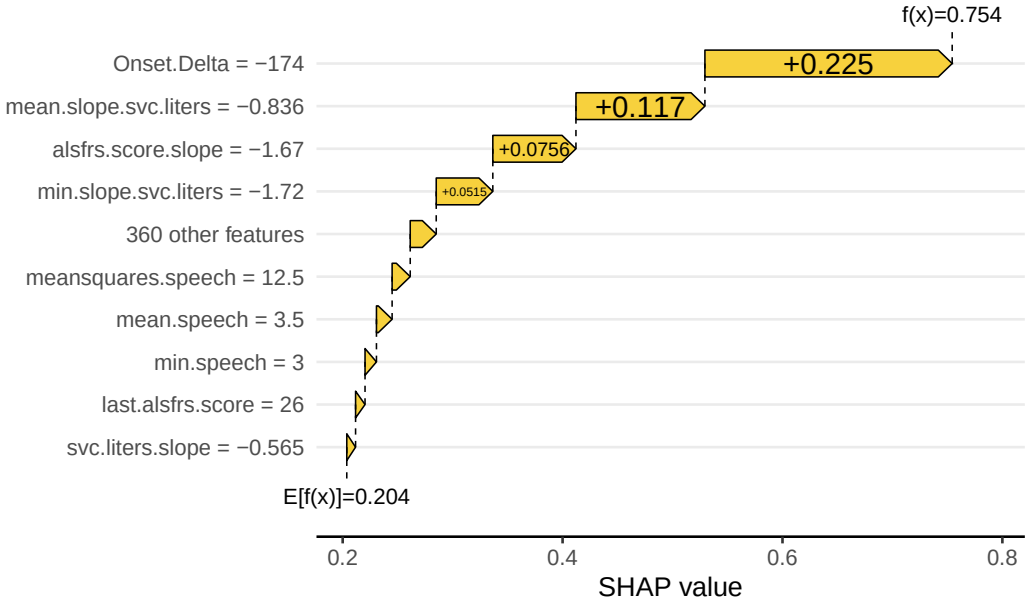
```
library(ggplot2)
library(shapviz)

# Use 'shap' package to compute SHAP values for entire training set
shap <- import("shap")
explainer <- shap$TreeExplainer(ngb, model_output = 0L)
ex.trn <- explainer$shap_values(X.trn)  # training explanations
colnames(ex.trn) <- colnames(X.trn)
```

```
# Construct variable importance and SHAP dependence plots
viz <- shapviz(ex.trn, X = X.trn)
p1 <- sv_importance(viz) + theme_bw()
p2 <- sv_dependence(viz, v = "Onset.Delta", alpha = 0.3) + theme_bw()
gridExtra::grid.arrange(p1, p2, nrow = 1)
```



A dFRS of less than -1.1 is considered to be fast progression (Küffner et al. 2015). Hence, it could be useful to estimate the probability of `dFRS < -1.1` and provide an explanation for any individual who's corresponding probability is considered high. The prediction wrapper (`pfun()`) defined below computes the cumulative probability $Pr\,(\texttt{dFRS} < -1.1|\boldsymbol{x})$. We use it to determine the observation in the validation set with the highest cumulative probability:

```
pfun <- function(object, newdata) {
  dist <- object$pred_dist(newdata)
  pnorm(-1.1, mean = dist$params$loc, sd = dist$params$scale)
}

max(prob.val <- pfun(ngb, newdata = X.val))

#> [1] 0.754336

xval.max <- X.val[which.max(prob.val), ]  # obs with highest predicted prob

library(fastshap)

system.time({
  set.seed(1110)
  ex <- explain(ngb, X = X.trn, nsim = 100, pred_wrapper = pfun,
                newdata = xval.max, adjust = TRUE)
})

#>    user  system elapsed
#> 115.697   0.080 115.779

# Visualize with a force plot
viz <- shapviz(ex, X = xval.max, baseline = attr(ex, which = "baseline"))
sv_waterfall(viz, max_display = 10)
```



References

Aas, Kjersti, Martin Jullum, and Anders Løland. 2020. "Explaining Individual Predictions When Features Are Dependent: More Accurate Approximations to Shapley Values." https://arxiv.org/abs/1903.10464.

Achen, Christopher H. 1982. *Interpreting and Using Regression*. Interpreting and Using Regression. Sage Publications.

Biecek, Przemyslaw, Alicja Gosiewska, Hubert Baniecki, and Adam Izdebski. 2020. *iBreakDown: Model Agnostic Instance Level Variable Attributions*. https://CRAN.R-project.org/package=iBreakDown.

Breiman, Leo. 2001. "Random Forests." *Machine Learning* 45 (1): 5–32. https://doi.org/10.1023/A:1010933404324.

Chang, Winston. 2021. *R6: Encapsulated Classes with Reference Semantics*. https://CRAN.R-project.org/package=R6.

Charnes, A., B. Golany, M. Keane, and J. Rousseau. 1988. "Extremal Principle Solutions of Games in Characteristic Function Form: Core, Chebychev and Shapley Value Generalizations." In *Econometrics of Planning and Efficiency*, edited by Jati K. Sengupta and Gopal K. Kadekodi, 123–33. Dordrecht: Springer Netherlands. https://doi.org/10.1007/978-94-009-3677-5_7.

Chen, Hugh, Joseph D. Janizek, Scott Lundberg, and Su-In Lee. 2020. "True to the Model or True to the Data?" https://arxiv.org/abs/2006.16234.

Chen, Tianqi, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, et al. 2020. *Xgboost: Extreme Gradient Boosting*. https://github.com/dmlc/xgboost.

Chipman, Hugh A., Edward I. George, and Robert E. McCulloch. 2010. "BART: Bayesian additive regression trees." *The Annals of Applied Statistics* 4 (1): 266–98. https://doi.org/10.1214/09-AOAS285.

Covert, Ian, and Su-In Lee. 2021. "Improving KernelSHAP: Practical Shapley Value Estimation via Linear Regression." https://arxiv.org/abs/2012.01536.

Csárdi, Gábor, and Maëlle Salmon. 2020. *Pkgsearch: Search and Query CRAN r Packages*. https://CRAN.R-project.org/package=pkgsearch.

Duan, Tony, Anand Avati, Daisy Yi Ding, Khanh K. Thai, Sanjay Basu, Andrew Y. Ng, and Alejandro Schuler. 2020. "NGBoost: Natural Gradient Boosting for Probabilistic Prediction." arXiv. https://doi.org/10.48550/ARXIV.1910.03225.

Efron, Bradley, and Trevor Hastie. 2016. *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. Institute of Mathematical Statistics Monographs. Cambridge University Press. https://doi.org/10.1017/CBO9781316576533.

Friedman, Jerome H. 2001. "Greedy Function Approximation: A Gradient Boosting Machine." *The Annals of Statistics* 29 (5): 1189–1232. https://doi.org/10.1214/aos/1013203451.

———. 2002. "Stochastic Gradient Boosting." *Computational Statistics & Data Analysis* 38 (4): 367–78. https://doi.org/https://doi.org/10.1016/S0167-9473(01)00065-2.

Gosiewska, Alicja, and Przemyslaw Biecek. 2019. "iBreakDown: Uncertainty of Model Explanations for Non-Additive Predictive Models." *CoRR* abs/1903.11420. http://arxiv.org/abs/1903.11420.

Greenwell, Brandon. 2020. *Fastshap: Fast Approximate Shapley Values*. https://github.com/bgreenwell/fastshap.

Küffner, Robert, Neta Zach, Raquel Norel, Johann Hawe, David Schoenfeld, Liuxia Wang, Guang Li, et al. 2015. "Crowdsourced Analysis of Clinical Trial Data to Predict Amyotrophic Lateral Sclerosis Progression." *Nature Biotechnology* 33 (1): 51–57. https://doi.org/10.1038/nbt.3051.

Lundberg, Scott M., Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. "From Local Explanations to Global Understanding with Explainable AI for Trees." *Nature Machine Intelligence* 2 (1): 2522–5839.

Lundberg, Scott M., and Su-In Lee. 2017a. "A Unified Approach to Interpreting Model Predictions." In *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, 4765–74. Curran Associates, Inc. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

Lundberg, Scott M, and Su-In Lee. 2017b. "A Unified Approach to Interpreting Model Predictions." In *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, 4765–74. Curran Associates, Inc. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

Malley, James D., Jochen Kruppa, Abhijit Dasgupta, Karen Godlove Malley, and Andreas Ziegler. 2012. "Probability Machines: Consistent Probability Estimation Using Nonparametric Learning Machines." *Methods of Information in Medicine* 51 (1): 74–81. https://doi.org/10.3414/ME00-01-0052.

Mayer, Michael. 2022. *Shapviz: SHAP Visualizations*. https://CRAN.R-project.org/package=shapviz.

Molnar, Christoph. 2019. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*.

Molnar, Christoph, and Patrick Schratz. 2020. *Iml: Interpretable Machine Learning*. https://CRAN.R-project.org/package=iml.

Pearl, Judea. 2009. *Causality: Models, Reasoning and Inference*. 2nd ed. USA: Cambridge University Press.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. "Scikit-Learn: Machine Learning in Python." *Journal of Machine Learning Research* 12: 2825–30.

Rigby, Robert A., and Mikis D. Stasinopoulos. 2005. "Generalized Additive Models for Location, Scale and Shape." *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 54 (3): 507–54. https://doi.org/https://doi.org/10.1111/j.1467-9876.2005.00510.x.

Shapley, Lloyd S. 2016. "17. A Value for n-Person Games." In *Contributions to the Theory of Games (AM-28), Volume II*, edited by Harold William Kuhn and Albert William Tucker, 307–18. Princeton University Press. https://doi.org/10.1515/9781400881970-018.

Shi, Yu, Guolin Ke, Damien Soukhavong, James Lamb, Qi Meng, Thomas Finley, Taifeng Wang, et al. 2022. *Lightgbm: Light Gradient Boosting Machine*. https://CRAN.R-project.org/package=lightgbm.

Štrumbelj, Erik, and Igor Kononenko. 2014. "Explaining Prediction Models and Individual Predictions with Feature Contributions." *Knowledge and Information Systems* 31 (3): 647–65. https://doi.org/10.1007/s10115-013-0679-x.

Zhao, Qingyuan, and Trevor Hastie. 2021. "Causal Interpretations of Black-Box Models." *Journal of Business & Economic Statistics* 39 (1): 272–81. https://doi.org/10.1080/07350015.2019.1624293.

Bibliography

K. Aas, M. Jullum, and A. Løland. Explaining individual predictions when features are dependent: More accurate approximations to shapley values, 2019. [p6]

K. Aas, M. Jullum, and A. Løland. Explaining individual predictions when features are dependent: More accurate approximations to shapley values, 2020. [p5]

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016. URL http://arxiv.org/abs/1603.02754. [p6]

T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen, R. Mitchell, I. Cano, T. Zhou, M. Li, J. Xie, M. Lin, Y. Geng, and Y. Li. *xgboost: Extreme Gradient Boosting*, 2020. URL https://github.com/dmlc/xgboost. R package version 1.2.0.1. [p6]

B. Efron and T. Hastie. *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. Institute of Mathematical Statistics Monographs. Cambridge University Press, 2016. doi: 10.1017/CBO9781316576533. [p8]

B. Greenwell. *fastshap: Fast Approximate Shapley Values*, 2020. URL https://github.com/bgreenwell/fastshap. R package version 0.0.5.9000. [p6]

G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf. [p6]

S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf. [p4]

N. Sellereite and M. Jullum. shapr: An r-package for explaining machine learning models with dependence-aware shapley values. *Journal of Open Source Software*, 5(46):2027, 2019. doi: 10.21105/joss.02027. URL https://doi.org/10.21105/joss.02027. [p6]

N. Sellereite, M. Jullum, and A. Redelmeier. *shapr: Prediction Explanation with Dependence-Aware Shapley Values*, 2021. URL https://CRAN.R-project.org/package=shapr. R package version 0.2.0. [p5]

Brandon M. Greenwell
University of Cincinnati
2925 Campus Green Dr
Cincinnati, OH 45221
United States of America
https://github.com/bgreenwell
ORCiD: 0000-0002-8120-0084
greenwell.brandon@gmail.com