Explaining Predictions with Shapley Values—An Introduction to the fastshap Package

by Brandon M. Greenwell

Abstract An abstract of less than 150 words.

Introduction

Introductory section which may include references in parentheses (**?**), or cite a reference such as **?** in the text.

Background

So what's a Shapley value? The Shapley value is the average marginal contribution of a *player* across all possible *coalitions* in a *game*. In the context of statistical/machines learning,

Game: The prediction task for a single observation x.

Gain: The prediction for x minus the average prediction for all training observations.

Players The feature values of x that collaborate to receive the gain (i.e., predict a certain value).

In particular, the Shapley contribution of the i-th feature to an instance x is defined as

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

where ...

A simple example may help clarify the main ideas.

Fairly splitting a bar tab

Alex, Brad, and Brandon decide to go out for drinks after work. They shared a few pitchers of beer, but nobody payed attention to how much each person drank. What's a fair way to split the tab? 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 Ales, Brad, and Brandon drank together, they'd only pay $30.

So the next time the bartender asks how you want to split the tab, whip out a pencil and do the math!

Advantages

Disadvantages

Estimating Shapley values via Monte Carlo simulation: SampleSHAP

A single estimate of the contribution of x_i to $f\left(x\right)$ is nothing the more than the difference between two predictions, where each prediction is based on a sort of Frankenstein instance that' are's constructed by swapping out values between the instance being explained (x) and an instance selected at random

	Marginal contribution		
Permutation	Alex	Brad	Brandon
Alex, Brad, Brandon	\$10	\$15	\$5
Alex, Brandon, Brad	\$10	\$15	\$5
Brad, Alex, Brandon	\$5	\$20	\$5
Brad, Brandon, Alex	\$10	\$20	\$0
Brandon, Alex, Brad	\$5	\$15	\$10
Brandon, Brad, Alex	\$17	\$3	\$10
Shapley contribution:	\$9.50	\$14.67	\$5.83

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

```
1. For $j = 1, 2, \ldots, R$:
    (a) Select a random permutation $\mathcal{O}$ of the sequence $1, 2, \ldots, p$.
    (b) Select a random instance $w$ from the training instances $\boldsymbol{X}$.
    (c) Construct two new instances as follows:
        • $b_1 = x$, but all the features in $\mathcal{O}$ that appear after feature $x_i$ get their values
          swapped with the corresponding values in $w$.
        • $b_2 = x$, 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 $w$.
    (d) $\phi_{ij}\left(x\right) = f\left(b_1\right) - f\left(b_2\right)$.
2. $\phi_i\left(x\right) = \sum_{j=1}^{R} \phi_{ij}\left(x\right)/R$.
```

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

from the training data. To help stabilize the results, the procedure is repeated a large number, say, R, times, and the result averaged together.

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 $\phi_i\left(x\right)$. In practice, this depends on the variance of each feature in the observed training data, but typically $R >= 50 - -100$ will suffice (**Need reference**).

Special cases

The following sections discuss two special cases where exact Shapley explanations can be computed efficiently: additive linear models, and shallow trees and tree ensembles.

Linear models: LinearSHAP

Cite somewhere Štrumbelj and Kononenko (2014).

First, lets discuss how a feature's value contributes to a prediction $f\left(X\right)$ in a simple (additive) linear model. That is, let's assume for a moment that f takes the form

$$f\left(X\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$$

Recall that the contribution of the i-th feature to the prediction $f\left(X\right)$ is the difference between

$f(X)$ and the expected prediction if the i-th feature's value were not known:

$$\begin{aligned}\phi_i(X) &= \beta_0 + \cdots + \beta_i X_i + \cdots + \beta_p X_p \\ &\quad - \left(\beta_0 + \cdots + \beta_i \mathbb{E}(X_i) + \cdots + \beta_p X_p\right), \\ &= \beta_i \left(X_i - \mathbb{E}(X_i)\right)\end{aligned}$$

where we estimate $\mathbb{E}(X_i)$ with the corresponding sample mean $\bar{X}_i$. The quantity $\phi_i(X)$ is also referred to as the *situational importance of* X_i (Achen, 1982).

Tree-based models: TreeSHAP

TBD.

Kernal-based approximate Shapley values: KernelSHAP

KernelSHAP (Lundberg and Lee, 2017) uses a specially-weighted local linear regression to estimate SHAP values for any model. Unlike SampleSHAP...

Shapley values in R (and other lnaguages)

The **iml** package (**?**) provides the `Shapley()` function, which is a direct implementation of Algorithm~1. It is written in **R6** (**?**).

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

shapper provides an R interface to the Python **shap** library (Lundberg and Lee, 2017) using **reticulate** (**?**); however, it currently only supports KernelSHAP (**shap** itself suppors SampleSHAP, TreeSHAP, LinearSHAP, as well as various other model-specific Shapley explanation methods).

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

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.

On GitHub

- shapr
- shapFlex

In python, there's **shap**. In julia, there's **SampleSHAP.jl** (a lightweight port of **fastshap**), **ShapML.jl**, and **ShapleyValues.jl**.

So why fastshap?

Efficiency Like many post-hoc interpretation techniques (e.g., PDPs and ICE curves), SampleSHAP can be made more efficient by generating all the data up front, and scoring it only once (or twice, in the case of SampleSHAP). For example, PDPs and ICE curves can be efficiently constructed with only a single call to a scoring function by generating all of the required data up front using a single cross-join operation (which can be done rather efficiently in SQL or Spark). The scored data can then be post-processed/aggregated and displayed as either a PDP or set of ICE curves. An example using Spark with **sparklyr** **?** can be found here: `https://github.com/bgreenwell/pdp/issues/97`.

Fortunately, a similar trick can be exploited for SampleSHAP. Whether explaining a single instance with a large value of Monte Carlo reps (R), or explaining a large number of instances, the basic idea is to generate all the required Frankenstein instances b_1 and b_2 upfront, and stored in matrices $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$, respectively.

For example, suppose we wanted to estimate the contribution of x_i for each of the N rows of the available training data $\boldsymbol{X}$ using a single Monte-Carlo repetition in Algorithm~1 (i.e., $R = 1$)[1]. To start, we can generate the N random instances at once and store them in an $N \times p$ matrix $\boldsymbol{W}$. Rather generating N random permutations $\mathcal{O}$, and constructing b_1 and b_2 one at a time, the **fastshap** package

[1]The same idea also extends to explaining new instances.

uses C++—via **Rcpp** (**?**)—to efficiently generate an $N \times p$ logical matrix $\mathcal{O}$, where $\mathcal{O}_{kl} = 1$ if feature x_l appears before feature x_i in the k-th permutation, and 0 otherwise. This logical matrix can then be used to logically subset $\boldsymbol{X}$ and $\boldsymbol{W}$ to more efficiently construct $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ in a single swoop. The matrices (or data frames) can then be each scored once, and the difference taken, to generate a single replication of $\phi_i(x)$ for each row of $\boldsymbol{X}$.

Suppose instead we want to estimate the contribution of x_i for a single instance x, but using a large value of R for accuracy. We could employ the same trick, but in this case $\boldsymbol{X}$ would refer to the $R \times p$ matrix, where each row is a copy of the instance x.

fastshap also uses efficient exact methods for the special cases described in Sections…

Parallelization **fastshap** is faster at computing Shapley values for a single feature for a large number of instances (or a large value of R for a single instance). But what about a large number of features? Fortunately, Algorithm~1 can be trivially parallelized across features, and this is built into **fastshap**.

A simple benchmark comparison This section provides a brief example comparing various implementations of Shapley values using Kaggle's Titanic: Machine Learning from Disaster competition. While the true focus of the competition is to use machine learning to create a model that predicts which passengers survived the Titanic shipwreck, we'll focus on explaining predictions from a simple logistic regression model.

To start, we'll load the data, which are conveniently available in the **titanic** package (**?**), and do a little bit of cleaning.

```
# Read in the data and clean it up a bit
titanic <- titanic::titanic_train
features <- c(
  "Survived",  # passenger survival indicator
  "Pclass",    # passenger class
  "Sex",       # gender
  "Age",       # age
  "SibSp",     # number of siblings/spouses aboard
  "Parch",     # number of parents/children aboard
  "Fare",      # passenger fare
  "Embarked"   # port of embarkation
)
titanic <- titanic[, features]
titanic$Survived <- as.factor(titanic$Survived)
titanic <- na.omit(titanic)

# Data frame containing just the features
X <- subset(titanic, select = -Survived)
```

Next, we'll use the stats::glm() to fit a logistic regression model with only main effects (i.e., no tw-way interactions, etc.).

```
fit <- glm(Survived ~ ., data = titanic, family = binomial)
```

Suppose we wanted to explain the predicted survival probability for a new passenger named Jack:

```
jack <- data.frame(
  Pclass = 3,
  Sex = factor("male", levels = c("female", "male")),
  Age = 20,
  SibSp = 0,
  Parch = 0,
  Fare = 15,  # lower end of third-class ticket prices
  Embarked = factor("S", levels = c("", "C", "Q", "S"))
)
```

Our logistic regression model predicts that Jack's log-odds of survival is

```
predict(fit, newdata = jack)

#>         1
#> -1.845561
```

Yikes, that's equivalent to estimated 13.64% predicted probability of survival! With a baseline (i.e., average) survival rate of 40.62%, can we explain why the model predicts Jack to be much lower? Enter. . . Shapley values.

There is a growing number of R packages that provide Shapley explanations, the two most popular arguably being **iml** and **iBreakDown**. In this example, we'll compare those with **fastshap**.

To start, we need to define a few things (prediction wrapper, as well as both **iml**- and **iBreakDown**-related helpers).

```
# Prediction wrapper to compute predicted probability of survive
pfun <- function(object, newdata) {
  predict(object, newdata = newdata)
}

# DALEX-based helper for iBreakDown
explainer <- DALEX::explain(fit, data = X, y = titanic$Survived,

# Helper for iml
predictor <- iml::Predictor$new(fit, data = titanic, y = "Survived",
                                predict.fun = pfun)
```

Next, we call each implementation's Shapley-related function to compute explanations for Jack's prediction using 100 Monte Carlo repetitions.

```
# Compute explanations
set.seed(1039)  # for reproducibility
ex1 <- iBreakDown::shap(explainer, B = 100, new_observation = jack)
ex2 <- iml::Shapley$new(predictor, x.interest = jack, sample.size = 100)
ex3 <- fastshap::explain(fit, X = X, pred_wrapper = pfun, nsim = 100,
                         newdata = jack)

# Plot results
library(ggplot2)  # for `autoplot()` function
p3 <- plot(ex1) + ggtitle("iBreakDown")
p2 <- plot(ex2) + ggtitle("iml")
p1 <- autoplot(ex3, type = "contribution") + ggtitle("fastshap")
fastshap::grid.arrange(p1, p2, p3, nrow = 1)
```

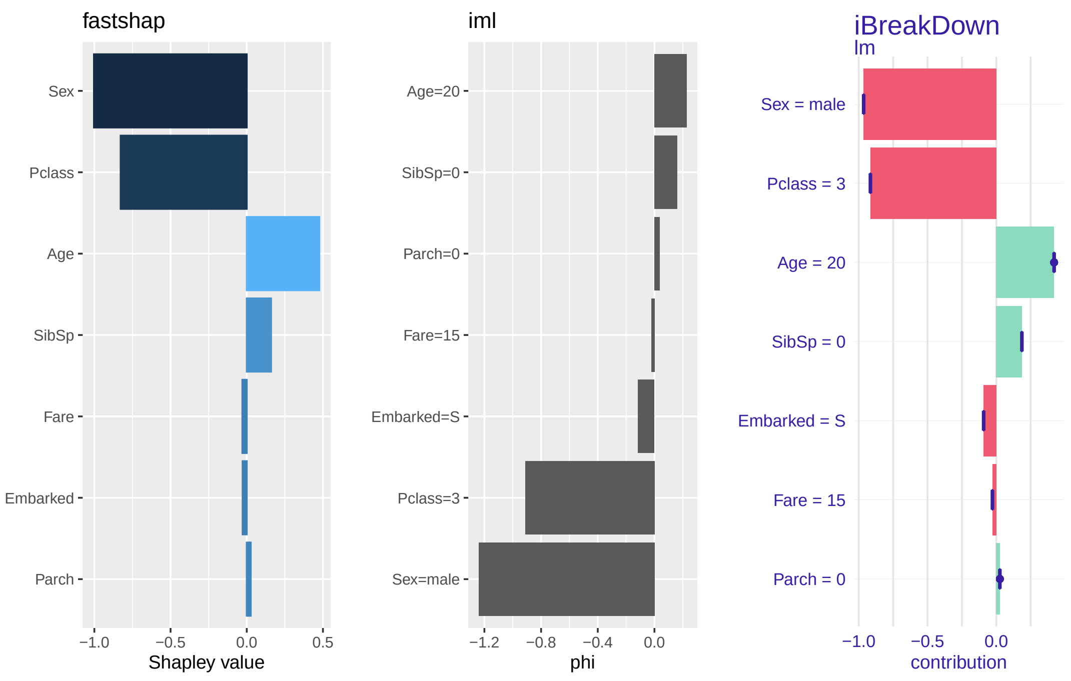

Figure 1: TBD.

Each package comes loaded with it's own bells and whistles (e.g., **iml** and **iBreakDown** have particularly fantastic visualizations). The main selling point of **fastshap** is speed! For example, all three packages (in fact, all general and practical implementations of Shapley values) use Algorithm~1 which requires a large number of Monte Carlo repetitions to achieve accurate results. Below is a simple benchmark looking at the estimated time (in seconds) to explain Jack's prediction as a function of the number of Monte Carlo repetitions for each implementation. (Note that this comparison does not make use of **fastshap**'s feature-wise parallelization.)

```
nsims <- c(1, 5, 10, 25, 50, 75, seq(from = 100, to = 1000, by = 100))
times1 <- times2 <- times3 <- numeric(length(nsims))
set.seed(904)
for (i in seq_along(nsims)) {
  message("nsim = ", nsims[i], "...")
  times1[i] <- system.time({
    iBreakDown::shap(explainer, B = nsims[i], new_observation = jack)
  })["elapsed"]
  times2[i] <- system.time({
    iml::Shapley$new(predictor, x.interest = jack, sample.size = nsims[i])
  })["elapsed"]
  times3[i] <- system.time({
    fastshap::explain(fit, X = X, newdata = jack, pred_wrapper = pfun,
                      nsim = nsims[i])
  })["elapsed"]
}
pal <- palette.colors(3, palette = "Okabe-Ito")  # colorblind friendly palette
plot(nsims, times1, type = "b", xlab = "Number of Monte Carlo repetitions",
     ylab = "Time (in seconds)", las = 1, pch = 19, col = pal[1L],
     xlim = c(0, max(nsims)), ylim = c(0, max(times1, times2, times3)))
lines(nsims, times2, type = "b", pch = 19, col = pal[2L],)
lines(nsims, times3, type = "b", pch = 19, col = pal[3L],)
legend("topleft",
       legend = c("iBreakDown", "iml", "fastshap"),
       lty = 1, pch = 19, col = pal, inset = 0.02)
```

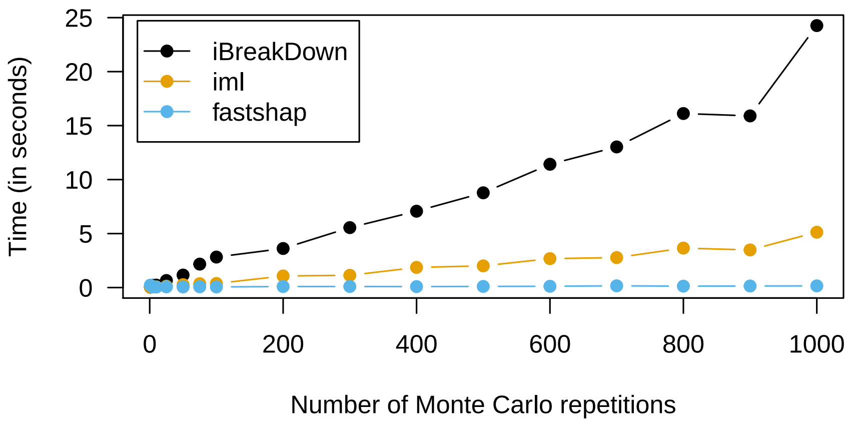

Figure 2: Quick benchmark between three different implementations of SampleSHAP for explaining Jack's unfortunate prediction.

The message to be taken from Figure~2 is that **fastshap** scales incredibly well with N or R, as long as the corresponding `predict()` method does.

Oh, and **fastshap** can produce instant (and exact) Shapley contributions for this example.

```
fastshap::explain(fit, newdata = jack, exact = TRUE)  # ExactSHAP

#> # A tibble: 1 x 7
#>   Pclass    Sex   Age SibSp  Parch    Fare Embarked
#>    <dbl>  <dbl> <dbl> <dbl>  <dbl>   <dbl>    <dbl>
#> 1 -0.915 -0.964 0.420 0.186 0.0260 -0.0282  -0.0919

fastshap::explain(fit, X = X, pred_wrapper = pfun, nsim = 10000,
                  newdata = jack)  # SampleSHAP
```

```
#> # A tibble: 1 x 7
#>   Pclass    Sex   Age SibSp  Parch    Fare Embarked
#>    <dbl>  <dbl> <dbl> <dbl>  <dbl>   <dbl>    <dbl>
#> 1 -0.929 -0.977 0.422 0.185 0.0257 -0.0290  -0.0865
```

```
predict(fit, newdata = jack, type = "terms")  # ExactSHAP (base R)
```

```
#>       Pclass        Sex       Age     SibSp      Parch       Fare    Embarked
#> 1 -0.9153946 -0.9644851 0.4204564 0.1861824 0.02599872 -0.0281944 -0.09194646
#> attr(,"constant")
#> [1] -0.4781785
```

Example: predicing sales prices

TBD.

Example: default of credit card clients

TBD.

Summary

This file is only a basic article template. For full details of *The R Journal* style and information on how to prepare your article for submission, see the Instructions for Authors.

Bibliography

C. H. Achen. *Interpreting and Using Regression*. Interpreting and Using Regression. Sage Publications, 1982. ISBN 9780803900004. [p3]

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016. URL `http://arxiv.org/abs/1603.02754`. [p3]

A. Gosiewska and P. Biecek. ibreakdown: Uncertainty of model explanations for non-additive predictive models. *CoRR*, abs/1903.11420, 2019. URL `http://arxiv.org/abs/1903.11420`. [p3]

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`. [p3]

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`. [p3]

E. Štrumbelj and I. Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and Information Systems*, 31(3):647–665, 2014. URL `https://doi.org/10.1007/s10115-013-0679-x`. [p2]

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