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

by Brandon M. Greenwell and ...

Abstract An abstract of less than 150 words.

WARNING:

This article is very much a work in progress. Read at your own risk... If you notice a major issue, or have suggestions, feel free to contribute!

TODO:

- ~~Flesh out outline/section headers.~~
- ~~Finish bar tab example (or switch to something better).~~
- Discuss SHAP as a unification of Shapley, LIME, etc.
- Find a good place to talk about "true to the model" versus "true to the data": `https://arxiv.org/pdf/2006.16234.pdf` (I think this is important for motivating the SampleSHAP approximation, which relies on randomly permuting instance values.)
- Fill out KernalSHAP section.
- Find motivating example for **iml** package; maybe credit card default risk?
- Find motivating example for **fastshap** package; maybe Ames housing?
- Find motivating example of interfacing with **shap** via **reticulate**. Can probably lift from the **fastshap** vigenette here: `https://bgreenwell.github.io/fastshap/articles/fastshap-vs-shap.html`.

Background

The *Shapley value* (Shapley, 2016) is an idea from coalitional game theory. In a coalitional game, 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 ($Q \subset S$) (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 value function that maps subsets of players to the real numbers: $v : 2^p \rightarrow \mathbb{R}$, where $v(S) = \Delta_S$ and $v(\emptyset) = 0$, where $\emptyset$ 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 value satisfies the following properties:

- Efficiency: $\sum_{i=1}^{p} \phi_i(v) = \Delta_S$.
- Null player: $\forall W \in S \setminus \{i\} : \Delta_W = \Delta_{W \cup \{i\}} \implies \phi_i(v) = 0$.
- Symmetry: $\forall W \in S \setminus \{i, j\} : \Delta_{W \cup \{i\}} = \Delta_{W \cup \{j\}} \implies \phi_i(v) = \phi_j(v)$.
- 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 the 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

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

where $\mathcal{O}$ is a specific permutation of the players indices $\{1, 2, \dots, p\}$, $\pi(p)$ is the set of all suck 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 (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 Ales, Brad, and Brandon drank together, they'd only pay \$30.

With only three players, we can enumerate all possible coalitions. In Table~1, we list out all possible permutations of the three players and list 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 across all six possible permutations (these are the column averages reported in the last row). In this case, 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!

	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.)

Shapley values for explaining predictions

Štrumbelj and Kononenko (2014) suggested using the Shapley value (Equation~(1)) to help explain predictions from a machine learning model. In the context of statistical/machines learning,

- A game is represented by the prediction task for a single observation x.
- The total payout/worth (Δ_S) for x is the prediction for x minus the average prediction for all training observations (call this the baseline prediction).
- The players are the individual feature values of x that collaborate to receive the payout (i.e., predict a certain value).

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 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 the conditional expectation of the model's prediction. 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 rely on what Chen et al. (2020) call the *interventional conditional expectation*, which can be expressed using Pearl (2009)'s $do\left(\cdot\right)$ operator:

$$v\left(S\right) = \mathbb{E}\left[f\left(x\right)|do\left(S\right)\right], \tag{2}$$

Equation~2 can be interpreted as the expected value of $f\left(x\right)$ given some intervention on the features in S, which assumed independence between the features in subset S and the remaining features. Shapley values based on this formulation of v are referred to as *interventional Shapley values* (Chen et al., 2020). The various Shapley value algorithms discussed over the next several sections fall under this form.

Estimating feature contributions in practice

In this section, we'll detail several algorithms for estimating Shapley values for the purporse of explain predictions from a machine learning model.

SampleSHAP: 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, that assumes independent features[1]. Their approach is described in Algorithm~1.

Here, 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 set of "Frankenstein instances"^footnote{The terminology used here takes inspiration from `https://christophm.github.io/interpretable-ml-book/shapley.html#the-shapley-value-in-detail`.} that are constructed by swapping out values between the instance being explained (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.

```
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 set of training observations $\boldsymbol{X}$.
    (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 $f\left(x\right)$.

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 $R >= 30 - -100$ will suffice (**FIXME: Is there a good reference**

[1]While SampleSHAP assumes independence features, several arguments can be made in favor of this assumption; see, for example, Chen et al. (2020) and the references therein.

for this?). In Section~**??**, we'll discuss a particularly optimized implementation of Algorithm~1 that only requires $2mp$ calls to a scoring function.

SampleSHAP can be computationally prohibitive if you need to explain large data sets. Fortunately, you often only need to explain a handful of predictions, for example the most extreme predictions. However, generating explanations for the entire training set, or a large enough sample thereof, can be useful for generating aggregated model summaries, like Shapley-based variable importance plots **FIXME: Add reference**.

LinearSHAP: Shapley values from additive linear models

FIXME: I believe these results assume independence between features. Need to check corresponding reference in `https://arxiv.org/pdf/2006.16234.pdf`.

FIXME: Cite somewhere Štrumbelj and Kononenko (2014).

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

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

Recall that the contribution of the i-th feature to the prediction $f(X)$ 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).

TreeSHAP: Efficient Shapley values for tree ensembles

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

KernelSHAP: Approximate Shapley values using kernel approximations

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

Shapley values is R (and other open source software)

Probably the first, and most widely used implementation of Shapley explanations is the Python **shap** library (Lundberg and Lee, 2017), 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 **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 using **reticulate** (**?**); however, it currently only supports KernelSHAP (**shap** itself supports SampleSHAP, TreeSHAP, LinearSHAP, as well as various other model-specific Shapley explanation methods).

I'm also aware of two experimental packages supporting Shapley explanations that are not currently on CRAN: **shapr** (Sellereite and Jullum, 2019) and **shapFlex** (Redell, 2019). As previously discussed, one drawback of traditional Shapley values 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. The **shapFlex** package, short for Shapley flexibility, provides approximate Shapley values that incorporate causal constraints into the model's feature space, as described in Frye et al. (2019).

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.

In Julia, there's **SampleSHAP.jl**, which is a lightweight port of **fastshap**; **ShapML.jl**, which is another Julia implementation of SampleSHAP; and **ShapleyValues.jl**, which hasn't been updated since 2016.

The next two sections illustrate more in-depth use of the **iml** and **fastshap** packages, respectively.

The iml package

TBD.

The fastshap package

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 using a large number of Monte Carlo repitions (R), or explaining a large number of instances with with $R = 1$, the basic idea is to generate all the required Frankenstein instances (Section~2.2.1) 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., $R = 1$)[2]. 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 uses C++—via **Rcpp** (**?**)—to efficiently generate an $N \times p$ logical matrix $\boldsymbol{\mathcal{O}}$, where $\boldsymbol{\mathcal{O}}_{kl} = 1$ 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...

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**.

Example: 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
```

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

```
  "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 Dawson[3]:

```
jack.dawson <- 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; technically, Jack won his ticket
  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.dawson)

#>         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.

[3]Insipiration for this example was taken from https://modeloriented.github.io/iBreakDown/articles/vignette_iBreakDown_titanic.html.

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

Finally, we plot the resulting explanations. Note that both **fastshap** and **iBreakDown** plot the feature contributions in the original order, whereas **iml** plots them in descending order.

```
library(ggplot2)

# Set ggplot2 theme
theme_set(theme_bw())

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

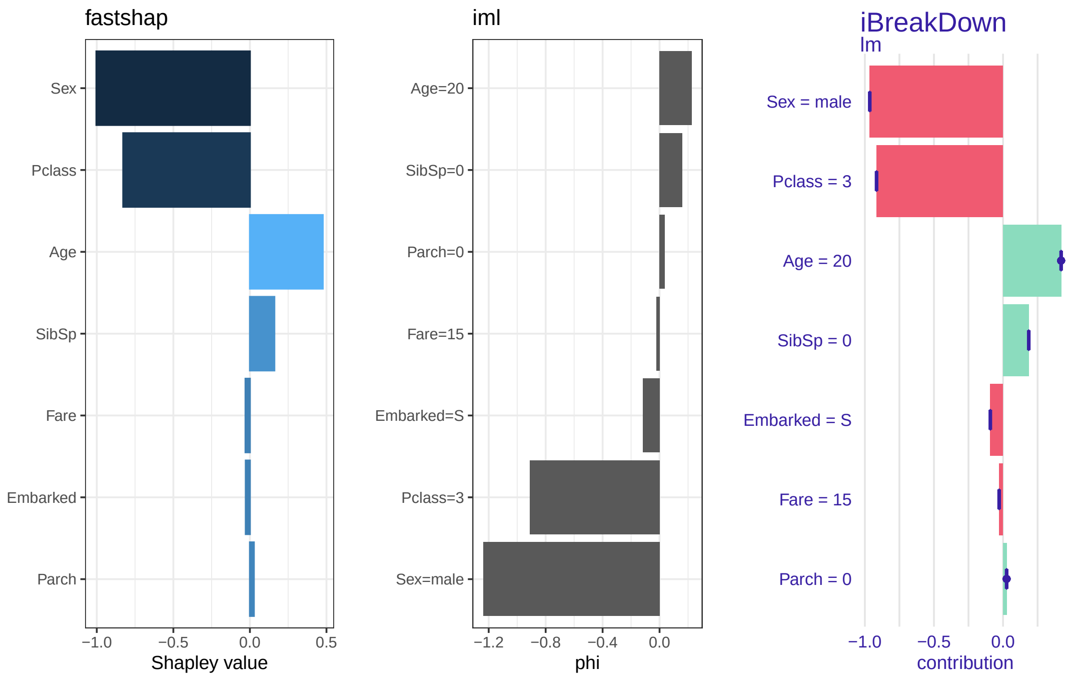

Figure 1: TBD.

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.)

```
# Number of Monte Carlo reps for each simulation
nsims <- c(1, 5, 10, 25, 50, 75, seq(from = 100, to = 1000, by = 100))

# Initialize vectors to store timings
times1 <- times2 <- times3 <- numeric(length(nsims))

# Run simulation
set.seed(904)  # for reproducibility
for (i in seq_along(nsims)) {  # iBreakDown
  message("nsim = ", nsims[i], "...")
  times1[i] <- system.time({
    iBreakDown::shap(explainer, B = nsims[i], new_observation = jack.dawson)
  })["elapsed"]
  times2[i] <- system.time({  # iml
    iml::Shapley$new(predictor, x.interest = jack.dawson,
                     sample.size = nsims[i])
  })["elapsed"]
  times3[i] <- system.time({  # fastshap
    fastshap::explain(fit, X = X, newdata = jack.dawson, pred_wrapper = pfun,
                      nsim = nsims[i])
  })["elapsed"]
}

# Plot results
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,
     xlim = c(0, max(nsims)), ylim = c(0, max(times1, times2, times3)))
lines(nsims, times2, type = "b", pch = 19, col = 2)
lines(nsims, times3, type = "b", pch = 19, col = 3)
legend("topleft",
       legend = c("iBreakDown", "iml", "fastshap"),
       lty = 1, pch = 19, col = 1:3, inset = 0.02)
```

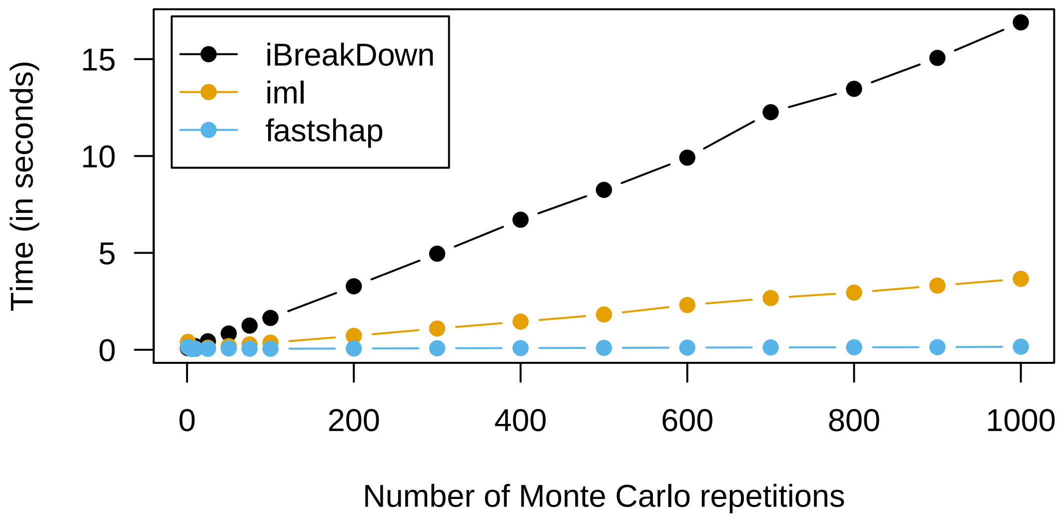

Figure 2: Quick benchmark between three different implementations of SampleSHAP for explaining Jack's unfortunate prediction.

```
palette("default")  # switch back to R's default color palette
```

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.dawson, 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.dawson)  # 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.dawson, 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
```

FIXME: Need a plot of fastshap explaining 1, 10, and 100 rows for various values of `nsim` to show that the increase in computation is not additive (i.e., it won't take twice as long to epxlain two rows compared to just one, etc.)

Example: Ames housing data

Use `fastshap::explain()` to explain predictions from a LightGBM model on the Ames housing data using both exact and approximate explanations (**Note:** there's a current PR that will give exact functionality for LightGBM models, which is now on CRAN).

To start, we'll load the Ames housing data from the **AmesHousing** package (**?**) and fit a random forest using the highly efficient **ranger** (**?**) package.

```
library(ranger)

# Set ggplot2 theme
theme_set(theme_bw())

# Load Ames housing data
ames <- as.data.frame(AmesHousing::make_ames())

# Fit a (default) random forest
set.seed(1644)  # for reproducibility
(rfo <- ranger(Sale_Price ~ ., data = ames))
```

```
#> Ranger result
#>
#> Call:
#>  ranger(Sale_Price ~ ., data = ames)
#>
#> Type:                             Regression
#> Number of trees:                  500
#> Sample size:                      2930
#> Number of independent variables:  80
#> Mtry:                             8
#> Target node size:                 5
#> Variable importance mode:         none
#> Splitrule:                        variance
#> OOB prediction error (MSE):       623733174
#> R squared (OOB):                  0.902265
```

Next we'll compute approximate Shapley values for the entire 2930 $\times$ 80 training set; to speed up computation, we'll turn on parallel processing. (Note that this took about one hour on a 3.1 GHz Dual-Core Intel Core i5 machine with 8 GB of RAM.)

```
library(doParallel)
library(fastshap)

# Set up parallel backend
cl <- if (.Platform$OS.type == "unix") 8 else makeCluster(8)
registerDoParallel(cl)

# Create data frame of only features
X <- subset(ames, select = -Sale_Price)

# Prediction wrapper
pfun <- function(object, newdata) {
  predict(object, data = newdata)$predictions
}
```

```
# Explain entire data set (useful for aggregated model summaries)
ex.all <- explain(rfo, X = X, nsim = 100, pred_wrapper = pfun, adjust = TRUE,
                  .parallel = TRUE)
head(ex)  # peak at results
```

```
#> # A tibble: 6 x 80
#>   MS_SubClass MS_Zoning Lot_Frontage Lot_Area Street Alley Lot_Shape
#>         <dbl>     <dbl>        <dbl>    <dbl>  <dbl> <dbl>     <dbl>
#> 1      284.       562.        2139.     6602.  0     -1.57      642.
#> 2     -308.        73.9        331.      346.  0      9.29     -282.
#> 3       -2.06     668.         977.     3331.  0     11.8       402.
#> 4      271.       729.        1603.     1313. -0.893 -4.70     -349.
#> 5      827.       437.         -67.2    2216.  0     -2.13      420.
#> 6      745.       812.          98.5     106.  0      4.41      369.
#> # ... with 73 more variables: Land_Contour <dbl>, Utilities <dbl>,
#> #   Lot_Config <dbl>, Land_Slope <dbl>, Neighborhood <dbl>, Condition_1 <dbl>,
#> #   Condition_2 <dbl>, Bldg_Type <dbl>, House_Style <dbl>, Overall_Qual <dbl>,
#> #   Overall_Cond <dbl>, Year_Built <dbl>, Year_Remod_Add <dbl>,
#> #   Roof_Style <dbl>, Roof_Matl <dbl>, Exterior_1st <dbl>, Exterior_2nd <dbl>,
#> #   Mas_Vnr_Type <dbl>, Mas_Vnr_Area <dbl>, Exter_Qual <dbl>, Exter_Cond <dbl>,
#> #   Foundation <dbl>, Bsmt_Qual <dbl>, Bsmt_Cond <dbl>, Bsmt_Exposure <dbl>,
#> #   BsmtFin_Type_1 <dbl>, BsmtFin_SF_1 <dbl>, BsmtFin_Type_2 <dbl>,
#> #   BsmtFin_SF_2 <dbl>, Bsmt_Unf_SF <dbl>, Total_Bsmt_SF <dbl>, Heating <dbl>,
#> #   Heating_QC <dbl>, Central_Air <dbl>, Electrical <dbl>, First_Flr_SF <dbl>,
#> #   Second_Flr_SF <dbl>, Low_Qual_Fin_SF <dbl>, Gr_Liv_Area <dbl>,
#> #   Bsmt_Full_Bath <dbl>, Bsmt_Half_Bath <dbl>, Full_Bath <dbl>,
#> #   Half_Bath <dbl>, Bedroom_AbvGr <dbl>, Kitchen_AbvGr <dbl>,
#> #   Kitchen_Qual <dbl>, TotRms_AbvGrd <dbl>, Functional <dbl>,
#> #   Fireplaces <dbl>, Fireplace_Qu <dbl>, Garage_Type <dbl>,
#> #   Garage_Finish <dbl>, Garage_Cars <dbl>, Garage_Area <dbl>,
#> #   Garage_Qual <dbl>, Garage_Cond <dbl>, Paved_Drive <dbl>,
#> #   Wood_Deck_SF <dbl>, Open_Porch_SF <dbl>, Enclosed_Porch <dbl>,
#> #   Three_season_porch <dbl>, Screen_Porch <dbl>, Pool_Area <dbl>,
#> #   Pool_QC <dbl>, Fence <dbl>, Misc_Feature <dbl>, Misc_Val <dbl>,
#> #   Mo_Sold <dbl>, Year_Sold <dbl>, Sale_Type <dbl>, Sale_Condition <dbl>,
#> #   Longitude <dbl>, Latitude <dbl>
```

```
library(ggplot2)

# Set ggplot2 theme
theme_set(theme_bw())

# Shapley summary plots (see Figure XYZ)
p1 <- autoplot(ex.all, num_features = 20)
p2 <- autoplot(ex.all, type = "dependence", feature = "Gr_Liv_Area", X = X,
               color_by = "Central_Air", alpha = 0.3) +
  scale_color_viridis_d(direction = -1) +  # cool colors
  theme(legend.position = c(0.7, 0.2),
        legend.key = element_rect(colour = "transparent", fill = "white"))
gridExtra::grid.arrange(p1, p2, nrow = 1)
```

Example: default of credit card clients

Use `iml::Shapley()` to explain most extreme predictions.

Example: An example of interfacing directly with shap via reticulate would be cool!

TBD.

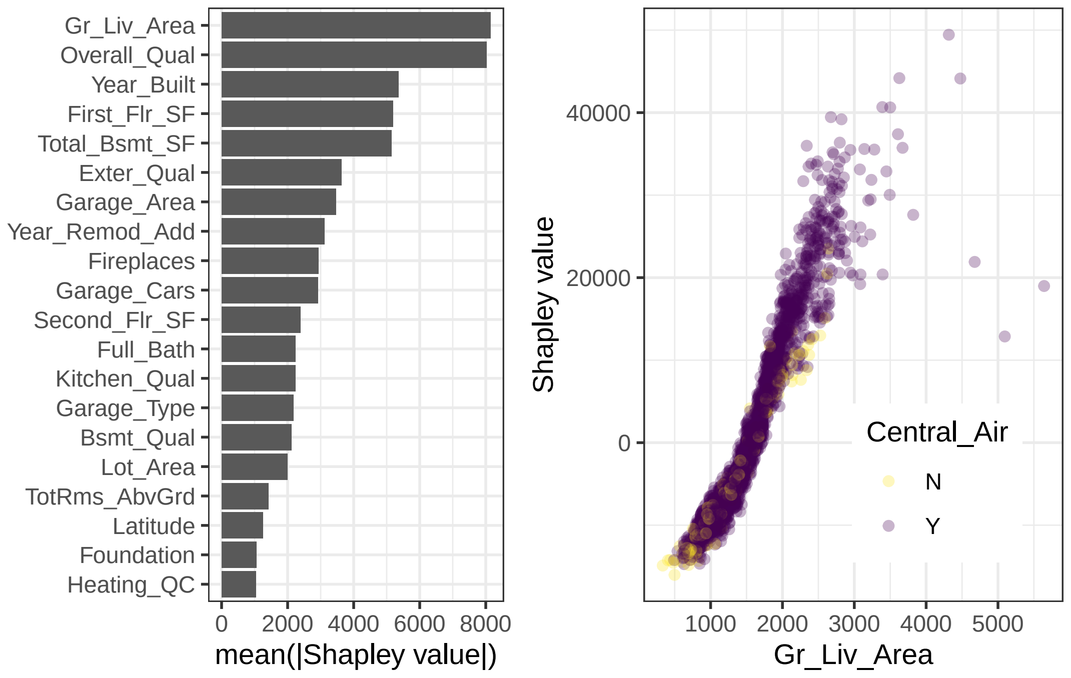

Figure 3: TBD.

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

K. Aas, M. Jullum, and A. Løland. Explaining individual predictions when features are dependent: More accurate approximations to shapley values, 2019. [p4]

C. H. Achen. *Interpreting and Using Regression*. Interpreting and Using Regression. Sage Publications, 1982. ISBN 9780803900004. [p4]

H. Chen, J. D. Janizek, S. Lundberg, and S.-I. Lee. True to the model or true to the data?, 2020. [p3]

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016. URL `http://arxiv.org/abs/1603.02754`. [p5]

C. Frye, I. Feige, and C. Rowat. Asymmetric shapley values: incorporating causal knowledge into model-agnostic explainability, 2019. [p4]

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`. [p4]

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`. [p5]

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]

J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, USA, 2nd edition, 2009. ISBN 052189560X. [p3]

N. Redell. Shapley decomposition of r-squared in machine learning models, 2019. [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`. [p4]

L. S. Shapley. *17. A Value for n-Person Games*, pages 307–318. Princeton University Press, 2016. URL `https://doi.org/10.1515/9781400881970-018`. [p1]

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, 3, 4]

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`

. . .
University of Cincinnati
2925 Campus Green Dr
Cincinnati, OH 45221
United States of America
ORCiD—0000-0002-8120-0084

`greenwell.brandon@gmail.com`