Explaining Predictions with Shapley Values in R

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.) Some remarks on causality here are probably also warranted.
- Fill out KernelSHAP 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** vignette here: `https://bgreenwell.github.io/fastshap/articles/fastshap-vs-shap.html`.
- DIscuss advantages and disadvantages described in `https://christophm.github.io/interpretable-ml-book/shapley.html`.
- Mention the technical difference between SHAP (which satisfies the consistency property) and Shapley values, and how we loosen the distinction of the terminology here.

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 *characteristic function* that assigns a value to each subset of players; in particular, $v : 2^p \to \mathbb{R}$, 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

FIXME: Need to make it clear how $v(S)$ is computed in the context of a machine learning model; maybe come up with a numeric example?

FIXME: Clear up notation (e.g., S^c or x_{S^c}?)

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. (2020a), 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. Chen et al. (2020a) make the distinction between two possibilities, each of which differs in their conditioning argument. The Shapley value implementations discussed in this paper (e.g., KernelSHAP and TreeSHAP) rely on what Chen et al. (2020a) call the *interventional conditional expectation*, which can be expressed using Pearl's (2009) *do* $(\cdot)$ operator:

$$\begin{aligned} v(S) &= \mathbb{E}\left[f(x_S, x_{S^c}) \,|do(x_S)\right] \\ &= \int f(x_S, x_{S^c})\, p(x_{S^c})\, dx_{S^c}, \end{aligned} \tag{2}$$

where S^c is the complement S, x_S and x_{S^c} are the set of features in S and S^c, respectively, and $p(x_{S^c})$ is the joint probability density of x_{S^c}. Equation (2) can be interpreted as the expected value of $f(x)$ given some intervention on the features in S, which assumes independence between x_S and x_{S^c}; a similar assumption is also used in the construction of *partial dependence plots* (Friedman, 2001), with the connection to Pearl's (2009) *do* $(\cdot)$ operator established in Zhao and Hastie (2021). Shapley values based on this formulation of v are referred to as *interventional Shapley values* (Chen et al., 2020a). The various Shapley value algorithms discussed over the next several sections fall under this form.

Estimating Shapley values/feature contributions in practice

In this section, we'll detail several algorithms for estimating Shapley values for the purpose 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^[1], that assumes independent features[2]. Their approach is described in Algorithm 1.

Here, A single estimate of the contribution of x_i to $f(x)$ 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.

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 $\phi_i(x)$. 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 (**FIXME:** Is there a good reference for this?). In Section 2.3.2, 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.

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

```
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) $\phi_{ij}(x) = f(b_1) - f(b_2)$.
2. $\phi_i(x) = \sum_{j=1}^{R} \phi_{ij}(x) / R$.
```

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

LinearSHAP: Shapley values from additive linear models

FIXME: Proof of the following is given in Aas et al. (2020).

First, lets discuss how a feature's value contributes to a prediction $f(X)$ in a simple (additive) linear model with independent features. 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:

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

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).

Note that if you're using R, then $\beta_i(X_i - \mathbb{E}(X_i))$ 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 and Section 2.3.3 for an example with a logistic regression model.

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 (Molnar and Schratz, 2020) 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** (Chen et al., 2020b)), CatBoost (**?**), and LightGBM (Ke et al., 2017). Both **fastshap** (Greenwell, 2020) 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

The **iml** package includes a consistent interface to several machine learning interpretability, including: variable importance plots, as described in Fisher et al. (2019), accumulated local effects (ALE) plots (Apley and Zhu, 2019); partial dependence plots (Friedman, 2001), individual conditional expectation (ICE) curves (Goldstein et al., 2015), the SampleSHAP algorithm described in Štrumbelj and Kononenko (2014) and discussed in Section 2.2.1, the *H*-statistic for quantifying the strength of interaction effects (Friedman and Popescu, 2008), tree-based surrogate models (see, for example, Molnar (2019, Chap. 5)), and others..

The `iml::Shapley()` function implements the SampleSHAP approximation described in Section 2.2.1.

Example: credit card default

To illustrate the use of `iml::Shapley()` we'll use the credit card default data set (Yeh and hui Lien, 2009) available from the UCI Machine Learning Repository at `https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients#`.

To start, we'll download the data set into a file named 'credit.xls'. Since it's an Excel file, we can read the data into R using the **readxl** package (**?**):

```
# Download and read in the credit default data from the UCI ML repo
tf <- tempfile(fileext = ".xls")
url <- paste0("https://archive.ics.uci.edu/ml/machine-learning-databases/",
              "00350/default%20of%20credit%20card%20clients.xls")
download.file(url, destfile = tf)
credit <- as.data.frame(readxl::read_xls(tf, skip = 1))
```

Next, we'll clean up the data set a bit by fixing the column names, re-encoding some of the categorical variables, removing the ID column, and coercing categorical variables to factors. A detailed description of the columns can be found in (Yeh and hui Lien, 2009), or at `https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients#`

```
# Clean up column names a bit
names(credit) <- tolower(names(credit))
names(credit)[names(credit) == "default payment next month"] <- "default"

# Remove ID column
credit$id <- NULL
```

```
# Clean up categorical features
credit$sex <- ifelse(credit$sex == 1, yes = "male", no = "female")
credit$education <- ifelse(
  test = credit$education == 1,
  yes = "graduate school",
  no = ifelse(
    test = credit$education == 2,
    yes = "university",
    no = ifelse(
      test = credit$education == 3,
      yes = "high school",
      no = "other"
    )
  )
)
credit$marriage <- ifelse(
  test = credit$marriage == 1,
  yes = "married",
  no = ifelse (
    test = credit$marriage == 2,
    yes = "single",
    no = "other"
  )
)
credit$default <- ifelse(credit$default == 1, yes = "yes", no = "no")

# Coerce to factors
for (i in seq_len(ncol(credit))) {
  if (is.character(credit[[i]])) {
    credit[[i]] <- as.factor(credit[[i]])
  }
}
```

Finally, we'll split the data into train test/test sets using a 70/30 split:

```
set.seed(1342)  # for reproducibility
ids <- sample(nrow(credit), size = 0.7 * nrow(credit), replace = FALSE)
credit.trn <- credit[ids, ]  # train
credit.tst <- credit[-ids, ]  # test
head(credit.trn)
```

```
#>       limit_bal    sex       education marriage age pay_0 pay_2 pay_3 pay_4
#> 8320      5e+04   male      university  married  42     0     0     0     0
#> 27602     2e+04 female      university   single  27     0     0     0     0
#> 16068     8e+04 female      university   single  31     0     0     2     2
#> 4729      5e+04 female graduate school   single  26     0     0     2     2
#> 4241      2e+05   male graduate school  married  34    -1    -1    -1    -1
#> 10363     6e+04 female     high school   single  40     0     0     0     0
#>       pay_5 pay_6 bill_amt1 bill_amt2 bill_amt3 bill_amt4 bill_amt5 bill_amt6
#> 8320      0    -1     46100     46949     18755     11112     11374      5919
#> 27602     0    -1     18854     19116     15030     13025     12223     15975
#> 16068     0     0     75953     81055     81587     78103     78335     78678
#> 4729      2     2      5800      8189      7909      9767      9466     11300
#> 4241     -1    -2      5879      5884      5171      4598       -95     -2175
#> 10363     0     0     35602     35911     36246     29831     29667     30062
#>       pay_amt1 pay_amt2 pay_amt3 pay_amt4 pay_amt5 pay_amt6 default
#> 8320      2133     1058     2000     2043     5919    16346     yes
#> 27602     1292     1509     1522     3111    20000     1121      no
#> 16068     7000     3000        0     3000     3100     5900      no
#> 4729      2500        0     2000        0     2000      400     yes
#> 4241      5884     5171     4598        0        0        0      no
#> 10363     1654     1762     1044     1065     1266      979      no
```

For modeling, we'll fig a gradient boosted tree ensemble (GBM) using the **R-gbm** package (originally by Greg Ridgeway). While more efficient and scalable GBM implementations certainly exist in R (e.g., **xgboost** and **lightgbm**), they don't often support categorical features without having to re-encode them numerically.

```
library(gbm)

# gbm required the target to be coded as 0/1
credit.trn$default <- ifelse(credit.trn$default == "yes", 1, 0)

# Fit a GBM to the credit default training set
set.seed(1554)  # for reproducibility
bst <- gbm(default ~ ., data = credit.trn, distribution = "bernoulli",
           n.trees = 300, interaction.depth = 5, shrinkage = 0.1, cv.folds = 5)

# Plot results and print optimal number of trees
par(las = 1)  # horizontal y-axis labels
(best <- gbm.perf(bst, plot.it = TRUE, method = "cv"))
```

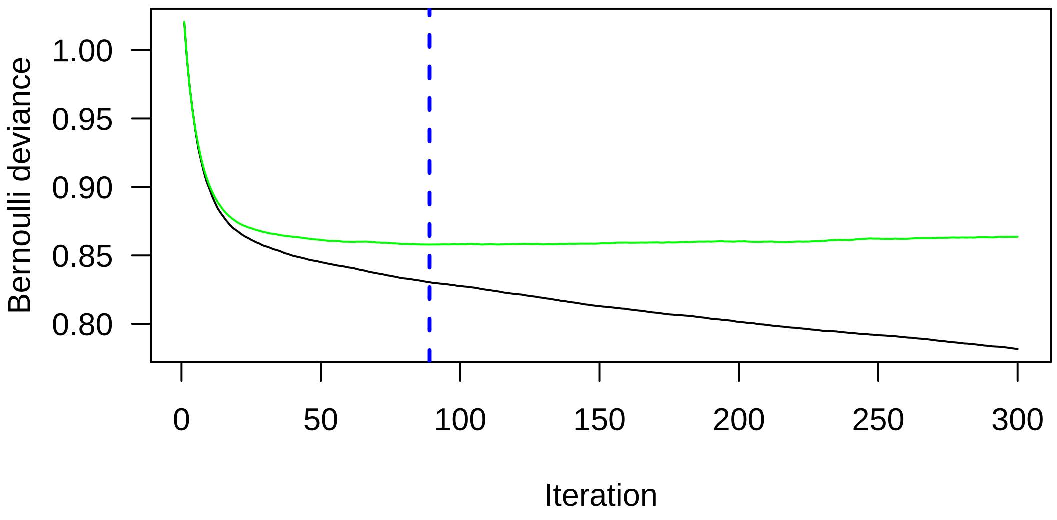

Figure 1: 5-fold cross-validation performance as a function of the number of trees from the GBM model applied to the credit default data set.

```
#> [1] 89
```

To use the **iml** package with out **gbm** model, we need to create a `Predictor` object; that is, a special object used by the functions in the **iml** package that holds the fitted machine learning model as well as any required metadata (e.g., the training data and a prediction wrapper so that **iml** knows how to obtain new predictions from your model). Below we construct a `Predictor` object for the previously fit **gbm** model. (Note that the prediction wrapper, `pfun()`, returns predictions on the logit scale.)

```
# Prediction wrapper
pfun <- function(object, newdata) {
  predict(object, newdata = newdata, n.trees = best)
}

# Construct a `Predictor` object
predictor <- iml::Predictor$new(bst, data = credit.trn, y = "default",
                                predict.fun = pfun)
```

We can now compute Shapley values for any observation (e.g., instances from the training set or a new instance). Here, we'll compute the feature contributions for the most extreme prediction in the test set. In this case, the maximum prediction (on the logit scale) is `2.639` corresponds to an estimated probability of roughly `0.933`.

```
# Compute test set predictions
max(p <- pfun(bst, newdata = credit.tst))

#> [1] 2.638971
```

```
# Compute approximate Shapley values for the most extreme prediction
set.seed(2241)  # for reproducibility
(ex <- iml::Shapley$new(predictor, x.interest = credit.tst[which.max(p), ],
                        sample.size = 100))

#> Interpretation method:  Shapley
#> Predicted value: 2.638971, Average prediction: -1.497362 (diff = 4.136333)
#>
#> Analysed predictor:
#> Prediction task: unknown
#>
#>
#> Analysed data:
#> Sampling from data.frame with 21000 rows and 23 columns.
#>
#> Head of results:
#>     feature         phi     phi.var        feature.value
#> 1 limit_bal -0.16780381 0.073059029     limit_bal=340000
#> 2       sex -0.01610681 0.001496965           sex=female
#> 3 education  0.08189806 0.045681690 education=university
#> 4  marriage -0.04323803 0.007992601       marriage=single
#> 5       age -0.04727680 0.003910510               age=31
#> 6     pay_0  1.25911873 0.500260821              pay_0=5
```

The output provides lots of relevant information. The predicted value for this observation is roughly 2.639 (on the logit scale), which is nearly 4.136 more than the baseline or average training prediction of -1.497. We can also see the explanations for the first few features. The approximate Shapley values, based on $R = 100$ replications of Algorithm 1, are given in the column labeled `phi`. The corresponding sample variance of each is given in column `phi.var`. In this example, we can see that `pay_0=5` contributed roughly 1.259 to the 4.136 difference.

It can often be useful to plot the feature contributions from a single explanation. With **iml**, you can just call the `plot()` method on the Shapley output:

```
# Plot feature contributions
plot(ex)  # Figure XYZ
```

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 repetitions (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., $R = 1$)[3]. 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** (Eddelbuettel et al., 2020)—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.

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

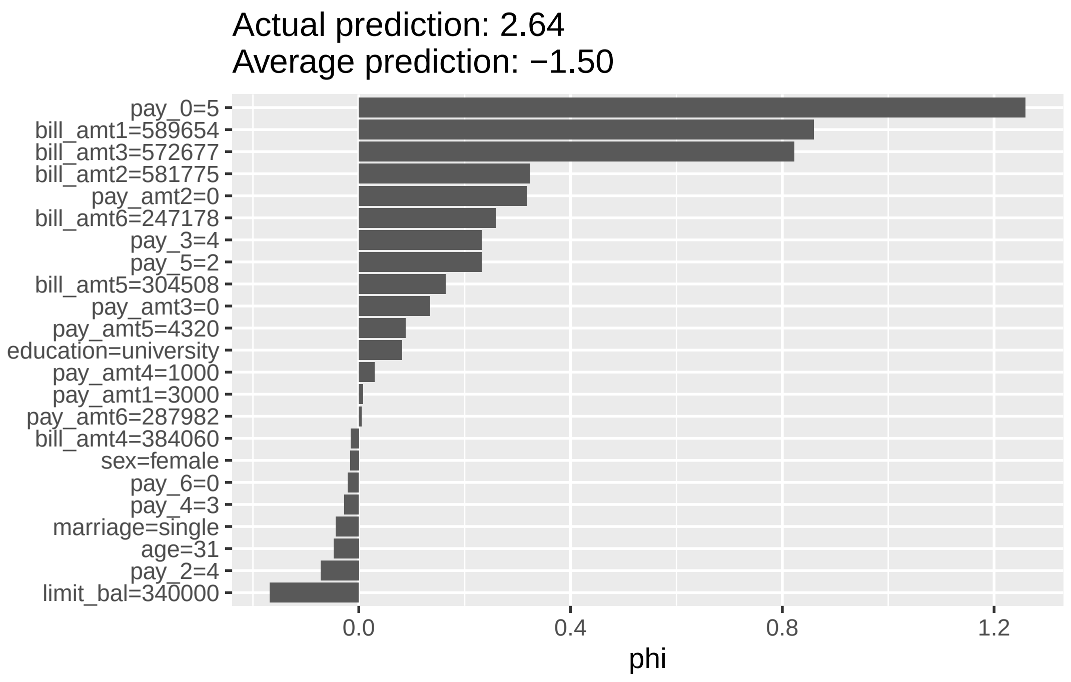

Figure 2: Approximate Shapley contributions for the most extreme prediction in the test set for the credit card default example. The predicted value for this observation is roughly 2.64 (on the logit scale), which is nearly 4.14 more than the baseline or average training prediction of -1.50. The three biggest contributers to this difference are `pay_0`, `bill_amt1`, and `bill_amt3`, each of which had a positive contribution to the difference.

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: Ames housing data

For illustration, we'll use the Ames housing data (Cock, 2011) which are available in the **AmesHousing** package (Kuhn, 2020). These data describe the sale of individual residential properties in Ames, Iowa from 2006–2010. The data set contains 2930 observations, 80 features (23 nominal, 23 ordinal, 14 discrete, and 20 continuous), and a continuous target giving the sale price of the home (`Sale_Price`). The version we'll load is a cleaned up version of the original data set and treats all categorical variables as nominal (see `?AmesHousing::make_ames` for details).

To start, we'll load the Ames housing data from the **AmesHousing** package (Kuhn, 2020) and fit a (default) random forest to the entire data set using the highly efficient **ranger** package (Wright et al., 2020).

```
library(ggplot2)
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 2930×80 training set; to speed up computation, we'll turn on parallel processing using the **doParallel** parallel backend (Corporation and Weston, 2020)^footnote{Note that **fastshap** depends on the **plyr** package (Wickham, 2020), which supports any parallel backend compatible with **foreach** (Revolution Analytics and Weston)}. (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>
```

As discussed in Lundberg et al. (2020), the individual feature contributions can be used to obtain a global understanding of a particular model (e.g., feature importance and feature effect plots).

A valuable way to summarize the feature contributions from a set of data is to aggregate them into an overall Shapley-based variable importance metric (Lundberg et al., 2020). You can compute the Shapley-based variable importance of a particular feature as the sum of the absolute values of the individual contributions across all rows; see Figure 3 (left) for an example on the Ames housing data. Here you can see that `Gr_Liv_Area` and `Overall_Qual` (the overall quality rating of the property) are the most important features in terms of their impact on, or contribution to, predicted sale price.

Shapley-based dependence plots (Lundberg et al., 2020) show how a feature's value (x-axis) impacts the predicted outcome (y-axis). In Figure 3 (right) we show the dependence of `Gr_Liv_Area` (above ground square footage) on the predicted sale price for the Ames housing data. For additional insight, you can color these plots by any other feature (which can help understand potential interaction effects); for illustration, we used `Central_Air` (whether or not central air is available) to color the individual points. Here you can see that `Gr_Liv_Area` has a roughly monotonic increasing relationship with predicted sale price, as you might expect.

```
# 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)
```

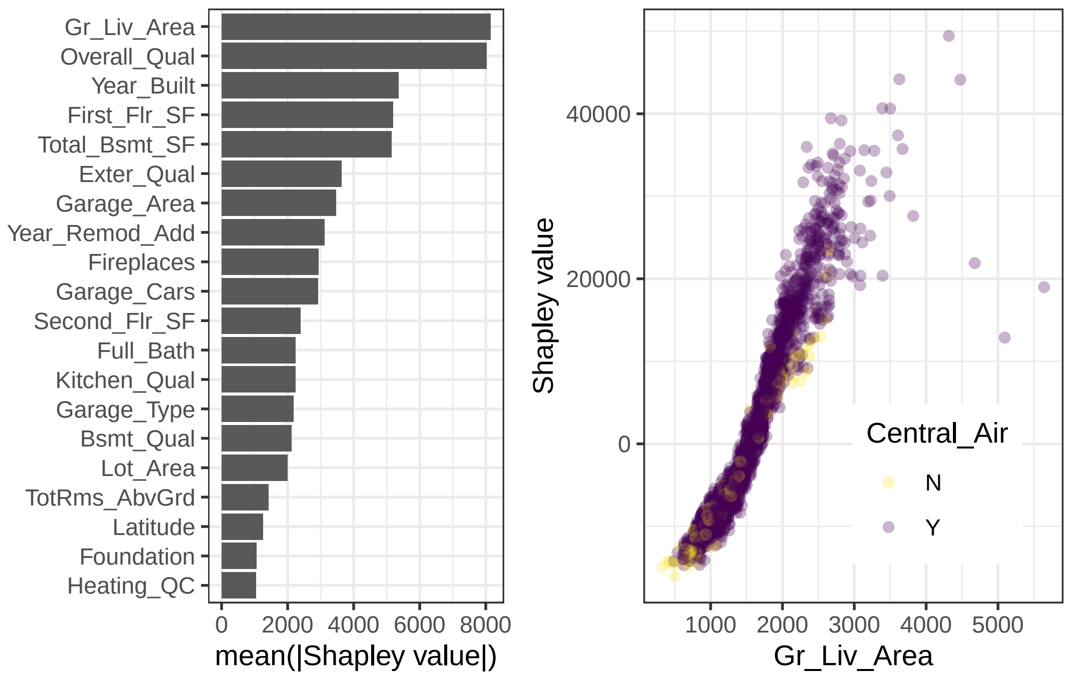

Figure 3: Global summary plots of the Ames housing random forest model. Left: Shapley-based variable importance plot. Right: Shapley-based dependence of predicted sale price on above ground square footage.

Example: Interfacing with shap via reticulate

While the **shapper** package provides a convenient (albeit currently limited) interface to Python's \pkg{shap package, we're going to use this example to illustrate how to interface directly with **shap** through **reticulate**.

For illustration, we'll compute feature contributions from a *multivariate adaptive regression splines* (MARS) model (Friedman, 1991) applied to the well-known Boston housing data (Harrison and Rubinfeld, 1978). MARS models can be fit using the **earth** (from mda:mars by Trevor Hastie and utilities with Thomas Lumley's leaps wrapper., 2020) and we'll use the corrected version of the Boston housing data available in the **pdp** package **?**; see ?pdp::boston for column descriptions and the original source.

To start, we'll load the data and fit a third degree MARS model (i.e., allowing for up to three-way interaction effects) using 5-fold cross-validation for pruning:

```
library(earth)

# Load the Boston housing data
boston <- pdp::boston
boston$chas <- as.integer(boston$chas) - 1  # coerce to 0/1
X <- subset(boston, select = -cmedv)  # feature columns only

# Fit a third degree MARS model using 5-fold cross-validation
(boston.mars <- earth(cmedv ~ ., data = boston, degree = 3, pmethod = "cv",
                      nfold = 5, ncross = 10))

#> Selected 17 of 28 terms, and 11 of 15 predictors (pmethod="cv")
#> Termination condition: Reached nk 31
#> Importance: rm, lstat, ptratio, tax, dis, nox, b, crim, age, lon, chas, ...
#> Number of terms at each degree of interaction: 1 7 8 1
#> GRSq 0.9076423  RSq 0.9216937  mean.oof.RSq 0.8038903 (sd 0.11)
#>
#> pmethod="backward" would have selected:
#>     23 terms 12 preds,  GRSq 0.9121471  RSq 0.9302413  mean.oof.RSq 0.7993457
```

The fitted model obtained a generalized R^2 (GRSq) of 0.908. Next, we'll compute feature contributions for each row in the training set using the KernelSHAP algorithm with $R = 100$ repetitions. Before proceeding, we need to define a special prediction wrapper that will work with the **shap** package. In this case, it's about as simple as it gets. In particular, when interfacing with **shap**, the prediction wrapper needs to be a function of newdata only, which means the fitted model object (in this case, boston.mars) needs to be supplied directly to the call to predict() inside the function. Further, the output from the prediction wrapper need to return a data frame or matrix (i.e., it needs to have a row and column dimension). Fortunately, **earth**'s predict method already returns the predictions in a matrix, so we don't need to do anything further here.

```
# Prediction wrapper for use with `shap$KernelExplainer()`
pfun.shap <- function(newdata) {  # Note: only a function of newdata!
  predict(boston.mars, newdata = newdata)
}  # returns an `nrow(newdata)` x 1 matrix of predictions
```

Next, we'll use **reticulate** to interface directly with the Python **shap** module^[4] and use it to run KernelSHAP (Section **??**) on our fitted MARS model. We'll request SHAP values for the entire training set. For comparison, we'll time the expression using system.time(). For nicer printing, we convert the results to a tibble using the **tibble** package (Müller and Wickham, 2020). (Note that, as indicated, the following expression will take a few minutes to run.)

```
library(reticulate)

# Import {shap} and run KernelSHAP algorithm with 100 repetitions
shap <- import("shap")
explainer <- shap$KernelExplainer(pfun.shap, data = X)  # initialize explainer
system.time({  # time KernelSHAP
  ex.ks <- explainer$shap_values(X, nsamples = 100L)
})
ex.ks <- ex.ks[[1L]]  # results returned as a list
colnames(ex.ks) <- colnames(X)  # add column names
(ex.ks <- tibble::as_tibble(ex.ks))  # for nicer printing
```

[4]You need to make sure that you have Python installed, as well as the **shap** package. If you need help setting up **reticulate**, see the vignette vignette("calling_python", package = "reticulate").

```
#>    user  system elapsed
#> 558.527  28.887 343.433
```

```
#> # A tibble: 506 x 15
#>       lon    lat   crim    zn indus  chas   nox    rm    age    dis   rad    tax
#>     <dbl>  <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl> <dbl>  <dbl>
#>  1 -0.637  0      0          0 0         0 0      0     0      0     0     -1.18
#>  2 -0.711  0     -0.191     0 0         0 1.20  -1.43 -0.350 -0.874 0      0.616
#>  3  0      0      0          0 0         0 0      4.71  0      0     0      0
#>  4  0      0      0          0 0         0 0.517  2.17  0.722  0     0      0
#>  5  0.930  0      0          0 0.581     0 0.274  3.92  0.277  0     0.463  0.196
#>  6  1.51  -0.350  0          0 0         0 1.09  -1.09  0     -1.43  0      0
#>  7  0     -0.155 -0.184     0 0         0 0.756 -2.15  0.309 -1.36  0      0
#>  8  0     -0.440 -0.255     0 0         0 1.04  -1.82 -0.925 -1.57  0      0
#>  9  0.635 -0.502  0          0 0         0 1.13  -3.03 -1.09  -2.03  0      0
#> 10  0.860 -0.811  0          0 0         0 0.671 -2.18  0     -2.36  0      0
#> # ... with 496 more rows, and 3 more variables: ptratio <dbl>, b <dbl>,
#> #   lstat <dbl>
```

To get an appreciation for the speed of **fastshap**'s implementation of SampleSHAP, let's use `fastshap::explain()` to compute SampleSHAP feature contributions for the entire training set using 100 Monte Carlo repetitions. The results are displayed in Figure 4, which shows the Shapley-based dependence of median home value (`cmedv`) on percentage of lower status of the population (`lstat`) using 100 Monte-Carlo repetitions from KernelSHAP (left) and SampleSHAP (right). The results are nearly the same. We drew lines at the knot locations MARS used to build piecewise linear terms with the `lstat` predictor (you can see the know locations for each predictor using `coef(boston.mars)`). Notice how the relationship between the Shapley contribution to $\widehat{\texttt{cmedv}}$ and `lstat` is relatively linear in between each knot location. In this case, the contribution of `lstat` to $\widehat{\texttt{cmedv}}$ is relatively monotonically decreasing, and flattens out after around `lstat = 22.11`.

```
pfun.fastshap <- function(object, newdata) {
  predict(object, newdata = newdata)[, 1L, drop = TRUE]
}  # `fastshap::explain()` requires an atomic vector of predictions
set.seed(1503)  # for reproducibility
system.time({
  ex.ss <- fastshap::explain(boston.mars, X = X, pred_wrapper = pfun.fastshap,
                             nsim = 100)
})
```

```
#>    user  system elapsed
#>  14.039   0.559  14.796
```

```
# Plot results
par(mfrow = c(1, 2), las = 1)
plot(X$lstat, ex.ks$lstat, col = adjustcolor(2, alpha.f = 0.6),
     xlab = "lstat", ylab = "KernelSHAP")
abline(v = c(0.713, 6.12, 22.11), lty = 2)
plot(X$lstat, ex.ss$lstat, col = adjustcolor(4, alpha.f = 0.6),
     xlab = "lstat", ylab = "SampleSHAP")
abline(v = c(0.713, 6.12, 22.11), lty = 2)
```

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 (Hendricks, 2015), 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
```

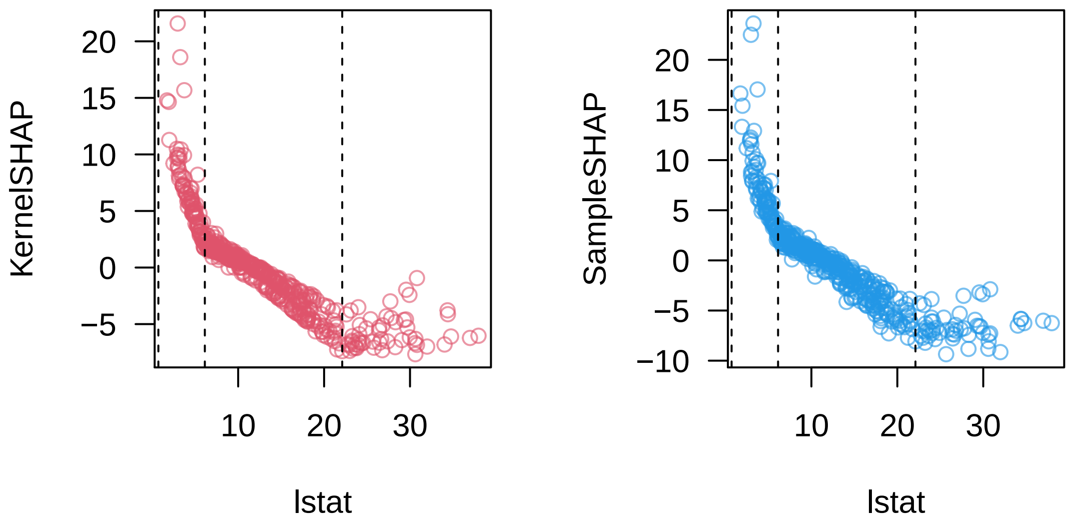

Figure 4: Shapley-based dependence of percentage of lower status of the population (lstat) on median home value (cmedv) using 100 Monte-Carlo repetitions from KernelSHAP (left) and SampleSHAP (right).

```
  "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 Dawson[5]:

```
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
```

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

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.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.

```
# 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)
```

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
```

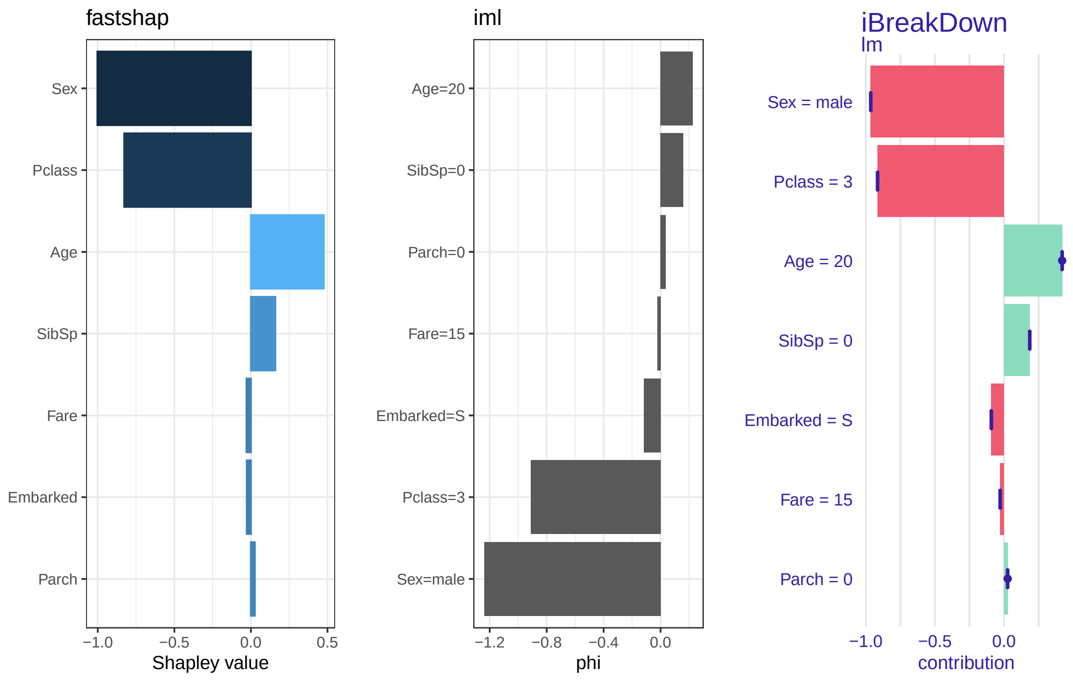

Figure 5: Feature contributions for the tianic example using three different implementations of SampleSHAP.

```
    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)


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

The message to be taken from Figure 6 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 using LinearSHAP (Section 2.2.2):

```
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
```

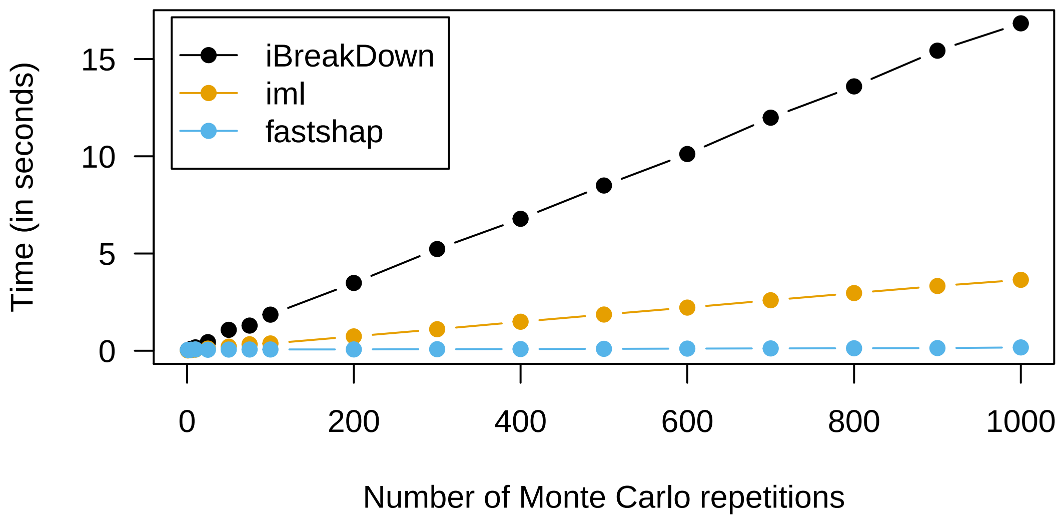

Figure 6: Quick benchmark between three different implementations of SampleSHAP for explaining Jack's unfortunate prediction.

```
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
```

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. [p5]

K. Aas, M. Jullum, and A. Løland. Explaining individual predictions when features are dependent: More accurate approximations to shapley values, 2020. [p4]

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

D. W. Apley and J. Zhu. Visualizing the effects of predictor variables in black box supervised learning models, 2019. [p5]

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

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*, 2020b. URL `https://github.com/dmlc/xgboost`. R package version 1.2.0.1. [p5]

D. D. Cock. Ames, iowa: Alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3):1–15, 2011. URL `https://doi.org/10.1080/10691898.2011.11889627`. [p9]

M. Corporation and S. Weston. *doParallel: Foreach Parallel Adaptor for the parallel Package*, 2020. URL `https://CRAN.R-project.org/package=doParallel`. R package version 1.0.16. [p10]

D. Eddelbuettel, R. Francois, J. Allaire, K. Ushey, Q. Kou, N. Russell, D. Bates, and J. Chambers. *Rcpp: Seamless R and C++ Integration*, 2020. URL https://CRAN.R-project.org/package=Rcpp. R package version 1.0.5. [p8]

A. Fisher, C. Rudin, and F. Dominici. All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously, 2019. [p5]

J. H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–67, 1991. URL https://doi.org/10.1214/aos/1176347963. [p12]

J. H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001. URL https://doi.org/10.1214/aos/1013203451. [p3, 5]

J. H. Friedman and B. E. Popescu. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3):916–954, 2008. doi: 10.1214/07-AOAS148. URL https://doi.org/10.1214/07-AOAS148. [p5]

S. M. D. from mda:mars by Trevor Hastie and R. T. U. A. M. F. utilities with Thomas Lumley's leaps wrapper. *earth: Multivariate Adaptive Regression Splines*, 2020. URL http://www.milbo.users.sonic.net/earth/. R package version 5.3.0. [p12]

C. Frye, I. Feige, and C. Rowat. Asymmetric shapley values: incorporating causal knowledge into model-agnostic explainability, 2019. [p5]

A. Goldstein, A. Kapelner, J. Bleich, and E. Pitkin. Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *Journal of Computational and Graphical Statistics*, 24(1):44–65, 2015. URL https://doi.org/10.1080/10618600.2014.907095. [p5]

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]

B. Greenwell. *fastshap: Fast Approximate Shapley Values*, 2020. URL https://github.com/bgreenwell/fastshap. R package version 0.0.5.9000. [p5]

D. Harrison and D. L. Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of Environmental Economics and Management*, 5(1):81–102, 1978. URL https://doi.org/10.1016/0095-0696(78)90006-2. [p12]

P. Hendricks. *titanic: Titanic Passenger Survival Data Set*, 2015. URL https://github.com/paulhendricks/titanic. R package version 0.1.0. [p13]

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]

M. Kuhn. *AmesHousing: The Ames Iowa Housing Data*, 2020. URL https://github.com/topepo/AmesHousing. R package version 0.0.4. [p9]

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]

S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee. From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2(1):2522–5839, 2020. [p11]

C. Molnar. *Interpretable Machine Learning*. 2019. https://christophm.github.io/interpretable-ml-book/. [p5]

C. Molnar and P. Schratz. *iml: Interpretable Machine Learning*, 2020. URL https://CRAN.R-project.org/package=iml. R package version 0.10.1. [p4]

K. Müller and H. Wickham. *tibble: Simple Data Frames*, 2020. URL https://CRAN.R-project.org/package=tibble. R package version 3.0.4. [p12]

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. [p5]

Revolution Analytics and S. Weston. *foreach: Provides Foreach Looping Construct*. [p10]

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. [p5]

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]

H. Wickham. *plyr: Tools for Splitting, Applying and Combining Data*, 2020. URL https://CRAN.R-project.org/package=plyr. R package version 1.8.6. [p10]

M. N. Wright, S. Wager, and P. Probst. *ranger: A Fast Implementation of Random Forests*, 2020. URL https://github.com/imbs-hl/ranger. R package version 0.12.1. [p9]

I.-C. Yeh and C. hui Lien. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert Systems with Applications*, 36(2, Part 1):2473–2480, 2009. URL https://doi.org/10.1016/j.eswa.2007.12.020. [p5]

Q. Zhao and T. Hastie. Causal interpretations of black-box models. *Journal of Business & Economic Statistics*, 39(1):272–281, 2021. URL https://doi.org/10.1080/07350015.2019.1624293. [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, 3, 5]

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