Variable Importance Plots—An Introduction to the vip Package

by Brandon M. Greenwell, Bradley C. Boehmke

Abstract In the era of "big data", it is becoming more of a challenge to not only build state-of-the-art predictive models, but also gain an understanding of what's really going on in the data. For example, it is often of interest to know which, if any, of the predictors in a fitted model are relatively influential on the predicted outcome. Some modern algorithms—like random forests (RFs) and gradient boosted decision trees (GBMs)—have a natural way of quantifying the importance or relative influence of each feature. Other algorithms—like naive Bayes classifiers and support vector machines—are not capable of doing so and *model-agnostic approaches* are generally used to measure each predictor's importance. Enter **vip**, an R package for constructing variable importance scores/plots for many types of supervised learning algorithms using model-specific and novel model-agnostic approaches. We'll also discuss a novel way to display both feature importance and feature effects together using *sparklines*, a very small line chart conveying the general shape or variation in some feature that can be directly embedded in text or tables.

Introduction

Too often machine learning (ML) models are summarized using a single metric (e.g., cross-validated accuracy) and then put into production. Although we often care about the predictions from these models, it is becoming routine to also understand the predictions! Understanding how an ML model makes its predictions helps build trust in the model and is the fundamental idea of the emerging field of *interpretable machine learning* (IML). For an in-depth discussion on IML, see Molnar (2019). In this paper, we focus on global methods for quantifying the importance of features in an ML model. Computing variable importance (VI) and communicating them through variable importance plots (VIPs) is a fundamental component of IML and is the main topic of this paper.

VI scores and VIPs can be constructed for general ML models using a number of packages. The **iml** package (Molnar et al., 2018) provides the `FeatureImp()` function which computes feature importance for general prediction models using the permutation approach (discussed later). It is written in **R6** (Chang, 2019) and allows the user to specify a generic loss function or select one from a pre-defined list (e.g., `loss = "mse"` for mean squared error). It also allows the user to specify whether importance is measured as the difference or as the ratio of the original model error and the model error after permutation. The user can also specify the number of repetitions used when permuting each feature to help stabilize the variability in the procedure.

The **ingredients** package (Biecek, 2019) also provides permutation-based VI scores through the `feature_importance()` function. (Note that this function recently replaced the now deprecated **DALEX** function `variable_importance()` (Biecek, 2018).) Similar to `iml::FeatureImp()`, this function allows the user to specify a loss function and how the importance scores are computed (e.g., using the difference or ratio). It also provides an option to sample the training data before shuffling the data to compute importance (the default is to use `n_sample = 1000`). This can help speed up computation.

The **caret** package (Kuhn, 2017b) includes a general `varImp()` function for computing model-specific and filter-based VI scores. Filter-based approaches, which are described in Kuhn and Johnson (2013), do not make use of the fitted model to measure VI. They also do not take into account the other predictors in the model. For regression problems, a popular filter-based approach to measuring the VI of a numeric predictor x is to first fit a flexible nonparametric model between x and the target Y; for example, the locally-weighted polynomial regression (LOWESS) method developed by Cleveland (1979). From this fit, a pseudo-R^2 measure can be obtained from the resulting residuals and used as a measure of VI. For categorical predictors, a different method based on standard statistical tests (e.g., t-tests and ANOVAs) can be employed; see Kuhn and Johnson (2013) for details. For classification problems, an area under the ROC curve (AUC) statistic can be used to quantify predictor importance. The AUC statistic is computed by using the predictor x as input to the ROC curve. If x can reasonably separate the classes of Y, that is a clear indicator that x is an important predictor (in terms of class separation) and this is captured in the corresponding AUC statistic. For problems with more than two classes, extensions of the ROC curve or a one-vs-all approach can be used.

If you use the **mlr** interface for fitting ML models (Bischl et al., 2016), then you can use the `getFeatureImportance()` function to extract model-specific VI scores from various tree-based models (e.g., RFs and GBMs). Unlike **caret**, the model needs to be fit via the **mlr** interface; for instance, you cannot use `getFeatureImportance()` on a **gbm** model (Ridgeway, 2017) unless it was fit using **mlr**.

While the **iml** and **DALEX** packages provide model-agnostic approaches to computing VI, **caret**, and to some extent **mlr**, provide model-specific approaches (e.g., using the absolute value of the *t*-statistic for linear models) as well less accurate filter-based approaches . Furthermore, each package has a completely different interface (e.g., **iml** is written in R6). The **vip** package (Greenwell and Boehmke, 2018) strives to provide a consistent interface to both model-specific and model-agnostic approaches to feature importance that is simple to use. The three most important functions exported by **vip** are described below:

- `vi()` computes VI scores using model-specific or model-agnostic approaches (the results are always returned as a tibble (Müller and Wickham, 2019));
- `vip()` constructs VIPs using model-specific or model-agnostic approaches with **ggplot2**-style graphics (Wickham, 2016);
- `add_sparklines()` adds a novel sparkline representation of feature effects (e.g., *partial dependence plots*) to any VI table produced by `vi()`.

There's also a function called `vint()` (for variable interactions) but is experimental and will not be discussed here; the interested reader is pointed to Greenwell et al. (2018). Note that `vi()` is actually a wrapper around four workhorse functions, `vi_model()`, `vi_ice()`, `vi_pdp()`, and `vi_permute()`, that compute various types of VI scores. The first computes model-specific VI scores, while the latter three produce model-agnostic ones. The workhorse function that actually gets called is controlled by the `method` argument in `vi()`; the default is `method = "model"` which corresponds to model-specific VI (see `?vip::vi` for details and links to further documentation).

Constructing VIPs in R

We'll illustrate major concepts using the Friedman 1 benchmark problem described in Friedman (1991) and Breiman (1996):

$$Y_i = 10\sin\left(\pi X_{1i} X_{2i}\right) + 20\left(X_{3i} - 0.5\right)^2 + 10X_{4i} + 5X_{5i} + \epsilon_i, \quad i = 1, 2, \dots, n, \tag{1}$$

where $\epsilon_i \overset{iid}{\sim} N\left(0, \sigma^2\right)$. Data from this model can be generated using the `vip::gen_friedman()`. By default, the features consist of 10 independent variables uniformly distributed on the interval $[0, 1]$; however, only 5 out of these 10 are actually used in the true model. The code chunk below simulates 500 observations from the model in Equation~(1) with $\sigma = 1$; see `?vip::gen_friedman` for details.

```
trn <- vip::gen_friedman(500, sigma = 1, seed = 101)  # simulate training data
tibble::as_tibble(trn)  # inspect output
```

```
#> # A tibble: 500 x 11
#>        y     x1    x2    x3    x4     x5      x6    x7    x8    x9   x10
#>    <dbl>  <dbl> <dbl> <dbl> <dbl>  <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1 14.9  0.372  0.406 0.102 0.322 0.693  0.758   0.518 0.530 0.878 0.763
#>  2 15.3  0.0438 0.602 0.602 0.999 0.776  0.533   0.509 0.487 0.118 0.176
#>  3 15.1  0.710  0.362 0.254 0.548 0.0180 0.765   0.715 0.844 0.334 0.118
#>  4 10.7  0.658  0.291 0.542 0.327 0.230  0.301   0.177 0.346 0.474 0.283
#>  5 17.6  0.250  0.794 0.383 0.947 0.462  0.00487 0.270 0.114 0.489 0.311
#>  6 18.3  0.300  0.701 0.992 0.386 0.666  0.198   0.924 0.775 0.736 0.974
#>  7 14.6  0.585  0.365 0.283 0.488 0.845  0.466   0.715 0.202 0.905 0.640
#>  8 17.0  0.333  0.552 0.858 0.509 0.697  0.388   0.260 0.355 0.517 0.165
#>  9  8.54 0.622  0.118 0.490 0.390 0.468  0.360   0.572 0.891 0.682 0.717
#> 10 15.0  0.546  0.150 0.476 0.706 0.829  0.373   0.192 0.873 0.456 0.694
#> # ... with 490 more rows
```

From Equation~(1), it should be clear that features X_1–X_5 are the most important! (The others don't influence Y at all.) Also, based on the form of the model, we'd expect X_4 to be the most important feature, probably followed by X_1 and X_2 (both comparably important), with X_5 probably less important. The influence of X_3 is harder to determine due to its quadratic nature, but it seems likely that this nonlinearity will suppress the variable's influence over its observed range (i.e., 0–1).

Model-specific VI

Some machine learning algorithms have their own way of quantifying the importance of each feature. We describe some of these in the subsections that follow. The issue with model-specific VI scores is that they are not necessarily comparable across different types of models. For example, directly comparing the impurity-based VI scores from tree-based models to the t-statistic from linear models.

Decision trees and tree ensembles

Decision trees probably offer the most natural model-specific approach to quantifying the importance of each feature. In a binary decision tree, at each node t, a single predictor is used to partition the data into two homogeneous groups. The chosen predictor is the one that maximizes some measure of improvement i^t. The relative importance of predictor X is the sum of the squared improvements over all internal nodes of the tree for which X was chosen as the partitioning variable; see Breiman et al. (1984) for details. This idea also extends to ensembles of decision trees, such as RFs and GBMs. In ensembles, the improvement score for each predictor is averaged across all the trees in the ensemble. Fortunately, due to the stabilizing effect of averaging, the improvement-based VI metric is often more reliable in large ensembles; see Hastie et al. (2009, p. 368). RFs offer an additional method for computing VI scores. The idea is to use the leftover out-of-bag (OOB) data to construct validation-set errors for each tree. Then, each predictor is randomly shuffled in the OOB data and the error is computed again. The idea is that if variable X is important, then the validation error will go up when X is perturbed in the OOB data. The difference in the two errors is recorded for the OOB data then averaged across all trees in the forest. Later on, we'll discuss a more general permutation method that can be applied to any supervised learning model.

To illustrate, we fit a CART-like regression tree, RF, and GBM to the simulated training data. (Note: there are a number of different packages available for fitting these types of models, we just picked popular implementations for illustration.)

```
# Load required packages
library(rpart)         # for fitting CART-like decision trees
library(randomForest)  # for fitting RFs
library(xgboost)       # for fitting GBMs

# Fit a single regression tree
tree <- rpart(y ~ ., data = trn)

# Fit an RF
set.seed(101)
rfo <- randomForest(y ~ ., data = trn, importance = TRUE)

# Fit a GBM
set.seed(102)
bst <- xgboost(
  data = data.matrix(subset(trn, select = -y)),
  label = trn$y,
  objective = "reg:linear",
  nrounds = 100,
  max_depth = 5,
  eta = 0.3,
  verbose = 0  # suppress printing
)
```

Each of the above packages include the ability to compute VI scores for all the features in the model; however, the implementation is rather package specific, as shown in the code chunk below. The results are displayed in Figure 1 (the code to reproduce these plots has been omitted but can be made available upon request).

```
# Extract VI scores from each model
vi_tree <- tree$variable.importance
vi_rfo <- rfo$variable.importance
vi_bst <- xgb.importance(model = bst)
```

As we would expect, all three methods rank the variables `x1`–`x5` as more important than the others. While this is good news, it is unfortunate that we have to remember the different functions and ways of extracting and plotting VI scores from various model fitting functions. This is one place where **vip** can help. . . one function to rule them all! Once **vip** is loaded, we can use `vi()` to extract a tibble of VI scores.

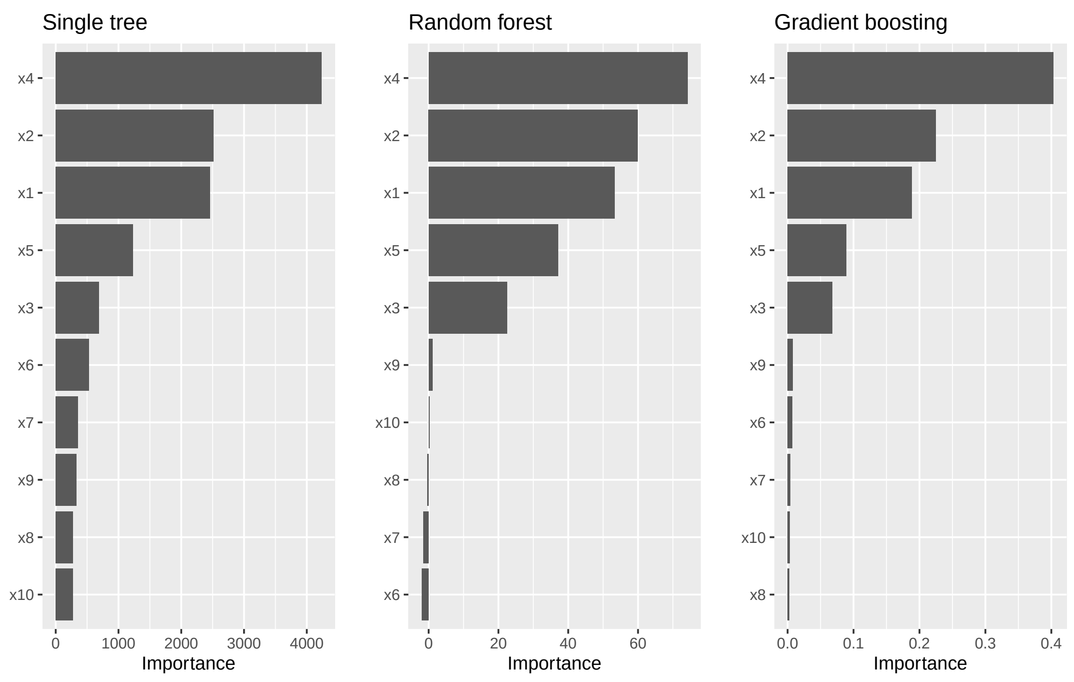


Figure 1: Model-specific VIPs for the three different tree-based models fit to the simulated Friedman data.

```
# Load required packages
library(vip)

# Compute model-specific VI scores
vi(tree)  # CART-like decision tree
```

```
#> # A tibble: 10 x 2
#>    Variable Importance
#>    <chr>         <dbl>
#>  1 x4            4234.
#>  2 x2            2513.
#>  3 x1            2461.
#>  4 x5            1230.
#>  5 x3             688.
#>  6 x6             533.
#>  7 x7             357.
#>  8 x9             331.
#>  9 x8             276.
#> 10 x10            275.
```

```
vi(rfo)   # RF
```

```
#> # A tibble: 10 x 2
#>    Variable Importance
#>    <chr>         <dbl>
#>  1 x4           74.2
#>  2 x2           59.9
#>  3 x1           53.3
#>  4 x5           37.1
#>  5 x3           22.5
#>  6 x9            1.05
#>  7 x10           0.254
#>  8 x8           -0.408
#>  9 x7           -1.56
#> 10 x6           -2.00
```

```
vi(bst)   # GBM
```

```
#> # A tibble: 10 x 2
#>    Variable Importance
#>    <chr>         <dbl>
#>  1 x4          0.403
#>  2 x2          0.225
#>  3 x1          0.189
#>  4 x5          0.0894
#>  5 x3          0.0682
#>  6 x9          0.00802
#>  7 x6          0.00746
#>  8 x7          0.00400
#>  9 x10         0.00377
#> 10 x8          0.00262
```

Notice how the `vi()` function always returns a tibble[1] with two columns: `Variable` and `Importance` (the exceptions are coefficient-based models which also include a `Sign` column giving the sign of the corresponding coefficient and permutation importance involving multiple Monte Carlo simulations, but more on that later). Also, by default, `vi()` always orders the VI scores from highest to lowest; this, among other options, can be controlled by the user (see `?vip::vi` for details). Plotting VI scores with `vip()` is just as straightforward. For example, the following code can be used to reproduce Figure 1.

```
p1 <- vip(tree) + ggtitle("Single tree")
p2 <- vip(rfo) + ggtitle("Random forest")
p3 <- vip(bst) + ggtitle("Gradient boosting")

# Figure 1
grid.arrange(p1, p2, p3, nrow = 1)
```

Notice how the `vip()` function always returns a `"ggplot"` object (by default, this will be a bar plot). For large models with many features, a Cleveland dot plot is more effective (in fact, a number of useful plotting options can be fiddled with). Below we call `vip()` and change a few useful options (the resulting plot is displayed in Figure 2). Note that we can also call `vip()` directly on a `"vi"` objects if it's already been constructed.

```
# Load required packages
library(ggplot2)  # for theme_light() function
theme_set(theme_light())  # set theme for all ggplot2-based plots

vip(bst, num_features = 5, geom = "point", horizontal = FALSE,
    aesthetics = list(color = "red", shape = 17, size = 4))
```

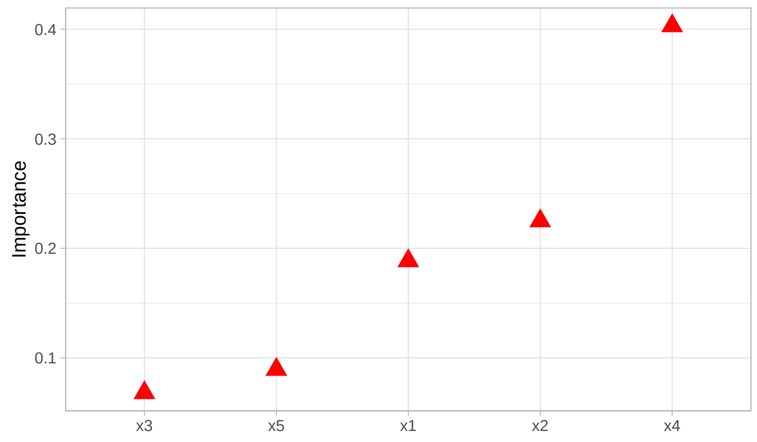


Figure 2: Illustrating various plotting options.

Linear models

In multiple linear regression, or linear models (LMs), the absolute value of the *t*-statistic (or some other scaled variant of the estimated coefficients) is commonly used as a measure of VI. The same idea also

[1]Technically, it's a tibble with an additional `"vi"` class.

extends to generalized linear models (GLMs). In the code chunk below, we fit an LM to the simulated `trn` data set allowing for all main effects and two-way interactions, then use the `step()` function to perform backward elimination. The resulting VIP is displayed in Figure 3.

```
# Fit a LM
linmod <- lm(y ~ .^2, data = trn)
backward <- step(linmod, direction = "backward", trace = 0)

# Extract VI scores
(vi_backward <- vi(backward))
```

```
#> # A tibble: 21 x 3
#>    Variable Importance Sign
#>    <chr>         <dbl> <chr>
#>  1 x4            14.2  POS
#>  2 x2             7.31 POS
#>  3 x1             5.63 POS
#>  4 x5             5.21 POS
#>  5 x3:x5          2.46 POS
#>  6 x1:x10         2.41 NEG
#>  7 x2:x6          2.41 NEG
#>  8 x1:x5          2.37 NEG
#>  9 x10            2.21 POS
#> 10 x3:x4          2.01 NEG
#> # ... with 11 more rows
```

```
# Plot VI scores; by default, `vip()` displays the top ten features
vip(vi_backward, num_features = length(coef(backward)),
    geom = "point", horizontal = FALSE)
```

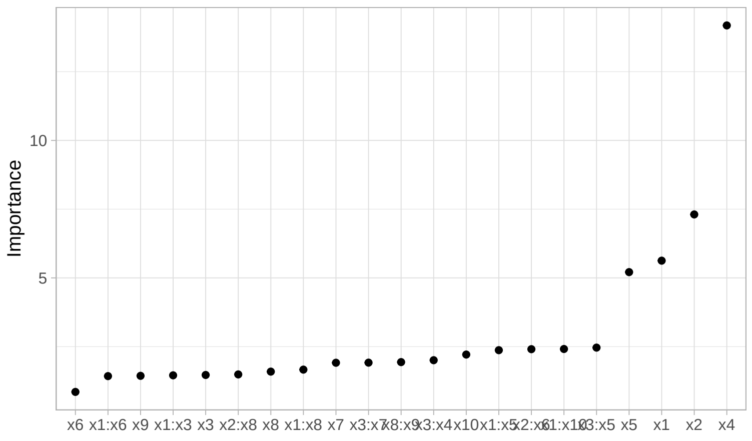


Figure 3: Example VIP from a linear model fit to the simulated Friedman data.

One issue with computing VI scores for LMs using the *t*-statistic approach is that a score is assigned to each term in the model, rather than to each individual feature! We can solve this problem using one of the model-agnostic approaches discussed later.

Multivariate adaptive regression splines (MARS), which were introduced in Friedman (1991), is an automatic regression technique and can be seen as a generalization of LMs and GLMs. In the MARS algorithm, the contribution (or VI score) for each predictor is determined using a generalized cross-validation (GCV) statistic (though, other statistics can also be used; see `?vip::vi_model` for details). An example using the **earth** package (Milborrow, 2017) is given below (the results are plotted in Figure 4):

```
# Load required packages
library(earth)

# Fit a MARS model
mars <- earth(y ~ ., data = trn, degree = 2, pmethod = "exhaustive")

# Extract VI scores
vi(mars, type = "gcv")
```

```
#> # A tibble: 10 x 2
#>    Variable Importance
#>    <chr>         <dbl>
#>  1 x4            100
#>  2 x1             83.2
#>  3 x2             83.2
#>  4 x5             59.3
#>  5 x3             43.5
#>  6 x6              0
#>  7 x7              0
#>  8 x8              0
#>  9 x9              0
#> 10 x10             0

# Plot VI scores
vip(mars)
```

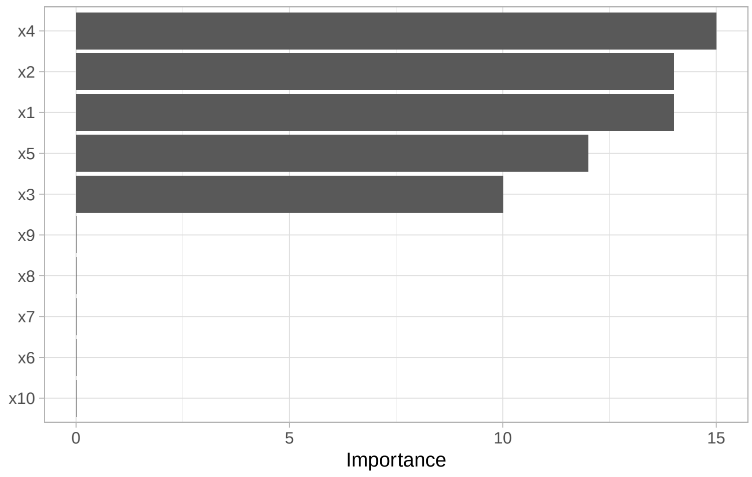


Figure 4: Example VIP from a MARS model fit to the simulated Friedman data.

To access VI scores directly in **earth**, you can use the `earth::evimp()` function.

Neural networks

For neural netwroks (NNs), two popular methods for constructing VI scores are the Garson algorithm (Garson, 1991), later modified by Goh (1995), and the Olden algorithm (Olden et al., 2004). For both algorithms, the basis of these VI scores is the network's connection weights. The Garson algorithm determines VI by identifying all weighted connections between the nodes of interest. Olden's algorithm, on the other hand, uses the products of the raw connection weights between each input and output neuron and sums these products across all hidden neurons. This has been shown to outperform the Garson method in various simulations. For DNNs, a similar method due to Gedeon (1997) considers the weights connecting the input features to the first two hidden layers (for simplicity and speed); but this method can be slow for large networks. We illustrate these two methods below using `vip()` with the **nnet** package (Venables and Ripley, 2002) (see the results in Figure 5).

```
# Load required packages
library(nnet)

# Fit a neural network
set.seed(0803)
nn <- nnet(y ~ ., data = trn, size = 7, decay = 0.1, linout = TRUE)

#> # weights:  85
#> initial  value 126298.176281
#> iter  10 value 5148.185025
#> iter  20 value 3563.828062
#> iter  30 value 3095.876922
#> iter  40 value 2638.672528
#> iter  50 value 1983.009001
#> iter  60 value 1759.428911
#> iter  70 value 1483.284575
#> iter  80 value 1112.219052
#> iter  90 value 835.941067
#> iter 100 value 748.120719
#> final  value 748.120719
#> stopped after 100 iterations

# VIPs
p1 <- vip(nn, type = "garson")
p2 <- vip(nn, type = "olden")

# Figure 5
grid.arrange(p1, p2, nrow = 1)
```

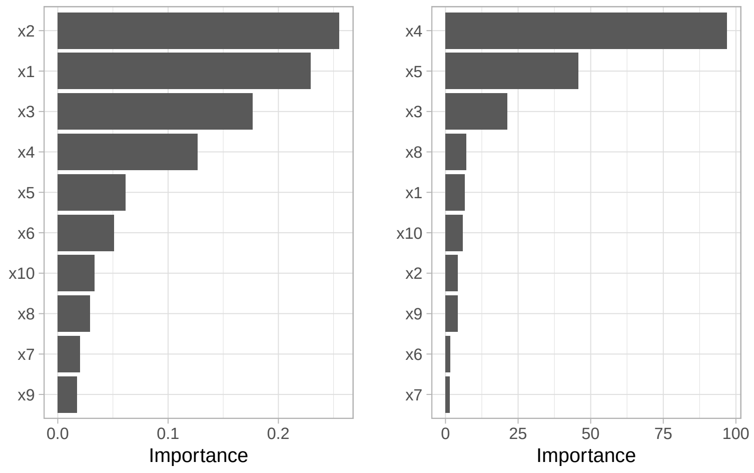


Figure 5: Example VIPs from a single-hidden-layer NN fit to the simulated Friedman data.

Model-agnostic VI

Model-agnostic interpredibility separates interpretation from the model. Compared to model-specific approaches, model-agnostic VI methods are more flexible and can be applied to any supervised learning algorithm. In this section, we discuss model-agnostic methods for quantifying global feature importance using three different approaches: 1) partial dependence plots (PDPs), 2) *individual conditional expectation* (ICE) curves, and 3) permutation-based feature importance. For specific details on approaches 1)–2), see Greenwell et al. (2018).

FIRM method

Our first model-agnostic approach is based on a simple *feature importance ranking measure* (FIRM); for details, see Greenwell et al. (2018) and Scholbeck et al. (2019). The specific approach used here is based on quantifying the "flatness" of the effects of each feature. Feature effects can be assessed using *partial dependence plots* (PDPs) or *individual conditional expectation* (ICE) curves. These approaches are model-agnostic and can be applied to any supervised learning algorithm. PDPs and ICE curves help visualize the effect of low cardinality subsets of the feature space on the estimated prediction surface (e.g., main effects and two/three-way interaction effects.). PDPs and ICE curves provide model-agnostic interpretations and can be constructed in the same way for any supervised learning algorithm. Below, we fit a projection pursuit regression (PPR) model and construct PDPs for each feature using the **pdp** package Greenwell (2017). The results are displayed in Figure 6. Notice how the PDPs for the uninformative features are relatively flat compared to the PDPs for features `x1`–`x2`!

Next, we compute PDP-based VI scores for the PPR and NN models. The PDP method constructs VI scores that quantify the relative "flatness" of each PDP (by default, this is defined by computing the standard deviation of the y-axis values for each PDP). To use the PDP method, specify `method = "firm"` in the call to `vi()` or `vip()` (or just use `vi_firm()` directly):

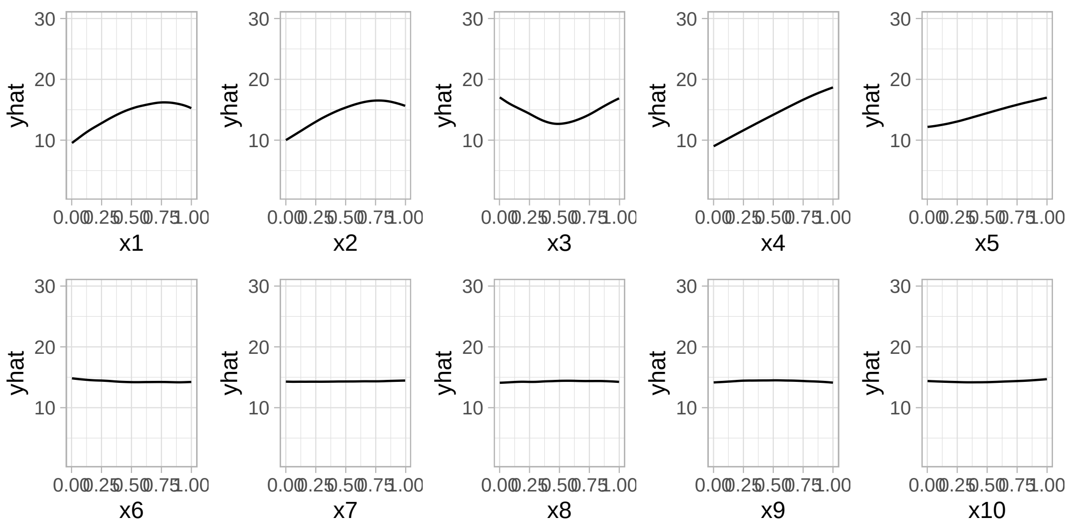


Figure 6: PDPs of main effects in the PPR model fit to the simulated Friedman data.

```
# Fit a PPR model (nterms was chosen using the caret package with 5 repeats of
# 5-fold cross-validation)
pp <- ppr(y ~ ., data = trn, nterms = 11)

# Plot VI scores
p1 <- vip(pp, method = "firm") + ggtitle("PPR")
p2 <- vip(nn, method = "firm") + ggtitle("NN")

# Figure 7
grid.arrange(p1, p2, ncol = 2)
```

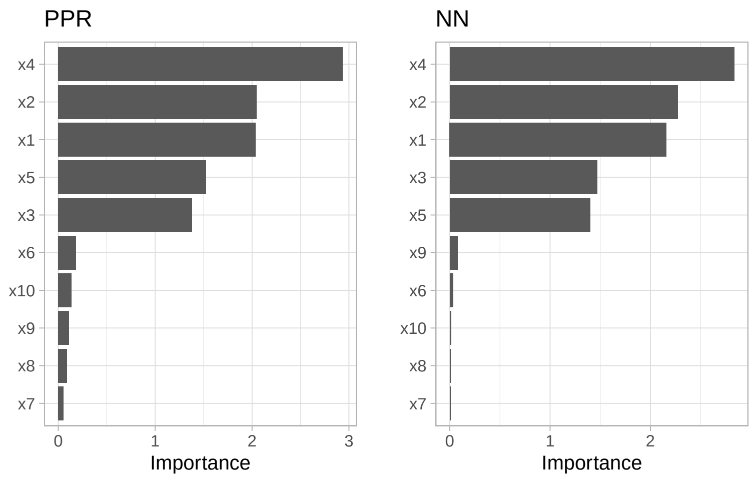


Figure 7: PDP-based feature importance for the PPR and NN models fit to the simulated Friedman data.

In Figure 7 we display the PDP-based feature importance for the previously obtained PPR and NN models. These VI scores essentially capture the variability in the partial dependence values for each main effect.

The ICE curve method is similar to the PDP method. The only difference is that we measure the "flatness" of each ICE curve and then aggregate the results (e.g., by averaging). If there are no (substantial) interaction effects, using `ice = TRUE` will produce results similar to using `ice = FALSE`. However, if strong interaction effects are present, they can obfuscate the main effects and render the PDP-based approach less useful (since the PDPs for important features can be relatively flat when certain interactions are present; see Goldstein et al. (2015) for details). In fact, it is probably safest to always use `ice = TRUE` when using the FIRM method.

Below, we display the ICE curves for each feature in the PPR model using the same y-axis scale;

see Figure 8. Again, there is a clear difference between the ICE curves for features `x1`–`x5` and `x6`–`x10`; the later being relatively flat by comparison. Also, notice how the ICE curves within each feature are relatively parallel (if the ICE curves within each feature were perfectly parallel, the standard deviation for each curve would be the same and the results will be identical to the PDP method). In this example, the interaction term between `x1` and `x2` does not obfuscate the PDPs for the main effects and the results are not much different.

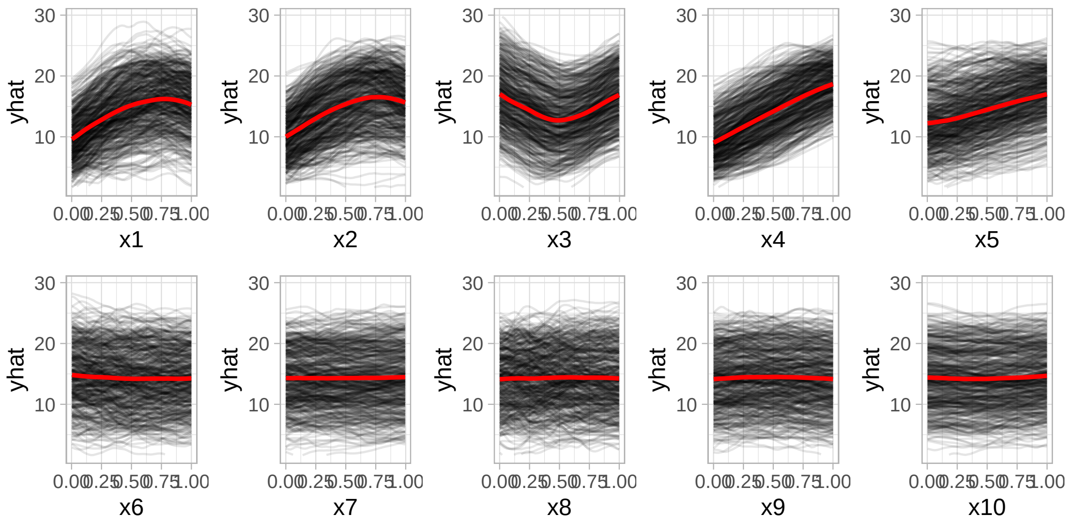


Figure 8: ICE curves for each feature in the PPR model fit to the simulated Friedman data. The red curve represents the PDP (i.e., the averaged ICE curves).

Obtaining the ICE-based feature importance scores is also straightforward, just specify `method = "ice"` in the call to `vip::vi()` or `vip::vip()`. This is illustrated in the code chunk below and the results are Figure 9 are similar to those obtained using the PDP method.

```
# Plot VI scores
p1 <- vip(pp, method = "firm", ice = TRUE) + ggtitle("PPR")
p2 <- vip(nn, method = "firm", ice = TRUE) + ggtitle("NN")

# Figure 9
grid.arrange(p1, p2, ncol = 2)
```

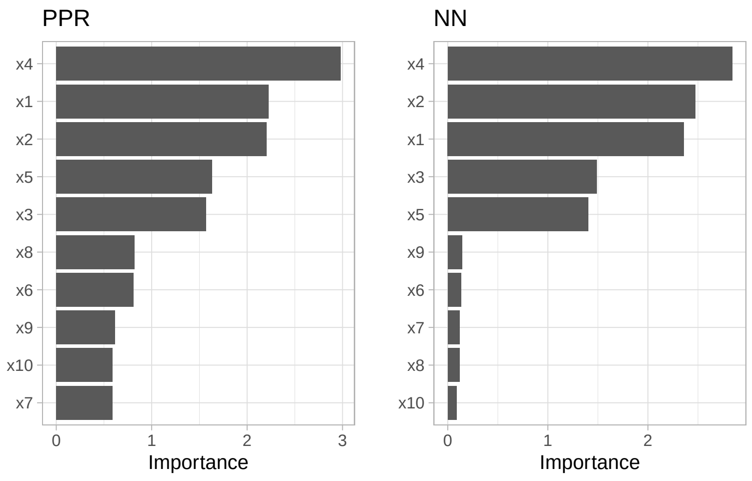


Figure 9: ICE-based feature importance for the PPR and NN models fit to the simulated Friedman data.

When using `method = "firm"`, the feature effect values are stored in an attribute called `"effects"`. This is a convenience so that the feature effect plots (e.g., PDPs and ICE curves) can easily be re-constructed and compared with the VI scores, as demonstrated in the example below (see Figure 10):

```
# Construct PDP-based variable importance scores
(vis <- vi(pp, method = "firm"))

#> # A tibble: 10 x 2
#>    Variable Importance
#>    <chr>         <dbl>
#>  1 x4           2.93
#>  2 x2           2.05
#>  3 x1           2.04
#>  4 x5           1.53
#>  5 x3           1.38
#>  6 x6           0.183
#>  7 x10          0.139
#>  8 x9           0.113
#>  9 x8           0.0899
#> 10 x7           0.0558

# Reconstruct PDPs for all 10 features
par(mfrow = c(2, 5))
for (name in paste0("x", 1:10)) {
  plot(attr(vis, which = "effects")[[name]], type = "l", ylim = c(9, 19), las = 1)
}
```

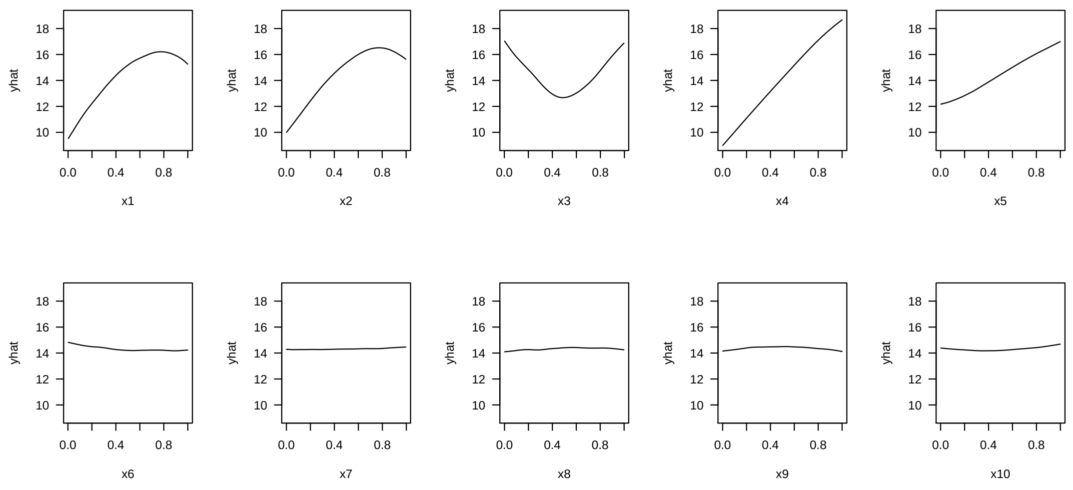


Figure 10: PDPs for all ten features reconstructed from the pdp attribute of the `vis` object.

Permutation method

The permutation method exists in various forms and was made popular in Breiman (2001) for RFs, before being generalized and extended in Fisher et al. (2018). The permutation approach used in **vip** is quite simple and is outlined in Algorithm 1 below. The idea is that if we randomly permute the values of an important feature in the training data, the training performance would degrade (since permuting the values of a feature effectively destroys any relationship between that feature and the target variable). This of course assumes that the model has been properly tuned (e.g., using cross-validation) and is not over fitting. The permutation approach uses the difference between some baseline performance measure (e.g., training R^2, AUC, or RMSE) and the same performance measure obtained after permuting the values of a particular feature in the training data (**Note:** the model is NOT refit to the training data after randomly permuting the values of a feature). It is also important to note that this method may not be appropriate when you have, for example, highly correlated features (since permuting one feature at a time may lead to unlikely data instances).

Let $X_1, X_2, \dots, X_j$ be the features of interest and let $\mathcal{M}_{orig}$ be the baseline performance metric for the trained model; for brevity, we'll assume smaller is better (e.g., classification error or RMSE). The permutation-based importance scores can be computed as follows:

Algorithm 1 can be improved or modified in a number of ways. For instance, the process can be repeated several times and the results averaged together. This helps to provide more stable VI scores, and also the opportunity to measure their variability. Rather than taking the difference in

```
1. For $i = 1, 2, \ldots, j$:
    (a) Permute the values of feature $X_i$ in the training data.
    (b) Recompute the performance metric on the permuted data $\mathcal{M}_{perm}$.
    (c) Record the difference from baseline using $imp(X_i) = \mathcal{M}_{perm} - \mathcal{M}_{orig}$.
2. Return the VI scores $imp(X_1), imp(X_2), \ldots, imp(X_j)$.
```

Algorithm 1: A simple algorithm for constructing permutation-based VI scores.

step (c), Molnar (2019, Chapter 5, Section 5.5.4) argues that using the ratio $\mathcal{M}_{perm}/\mathcal{M}_{orig}$ makes the importance scores more comparable across different problems. It's also possible to assign importance scores to groups of features (e.g., by permuting more than one feature at a time); this would be useful if features can be categorized into mutually exclusive groups, for instance, categorical features that have been *one-hot-encoded*.

To use the permutation approach in **vip**, specify `method = "permute"` in the call to `vi()` or `vip()`. Note that using `method = "permute"` requires specifying a few additional arguments (e.g., the training data, target name, a prediction function, etc.); see `?vi_permute` for details.

An example is given below for the previously fitted PPR and NN models. Here we use R^2 (`metric = "rsquared"`) as the evaluation metric. The results, which are displayed in Figure 11, agree with those obtained using the PDP- and ICE-based methods.

```
# Plot VI scores
set.seed(2021)  # for reproducibility
p1 <- vip(pp, method = "permute", target = "y", metric = "rsquared",
          pred_wrapper = predict) + ggtitle("PPR")
p2 <- vip(nn, method = "permute", target = "y", metric = "rsquared",
          pred_wrapper = predict) + ggtitle("NN")

# Figure 11
grid.arrange(p1, p2, ncol = 2)
```

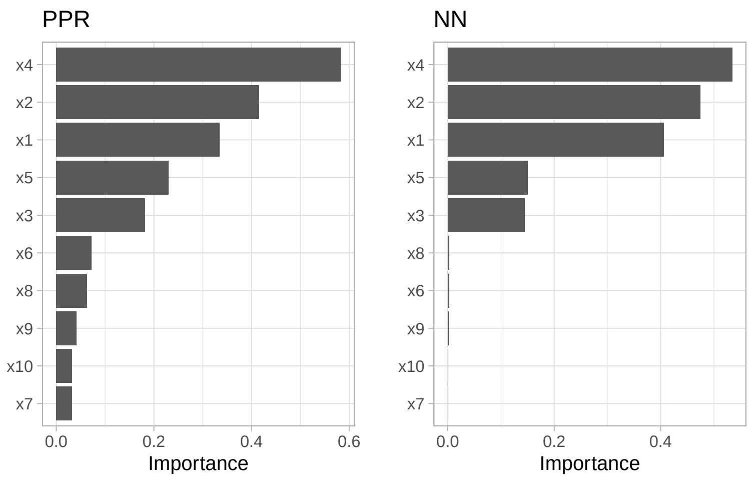
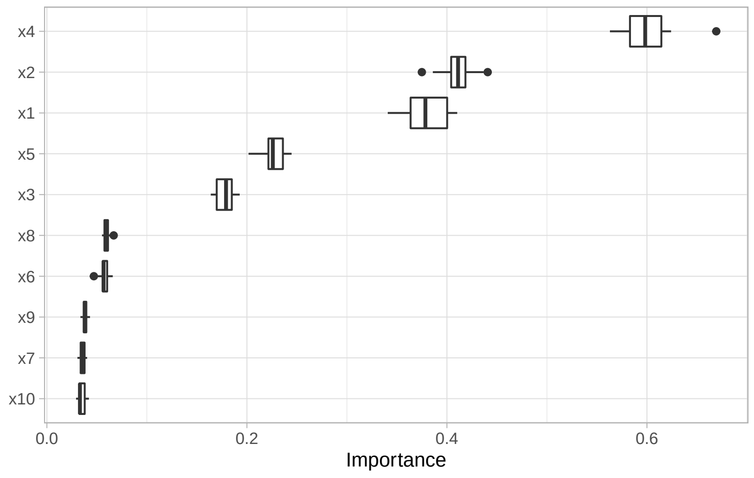


Figure 11: Permutation-based feature importance for the PPR and NN models fit to the simulated Friedman data.

The permutation approach introduces randomness into the procedure and therefore should be run more than once if computationally feasible. Fortunately, this also allows us to compute standard errors for the estimated VI scores, as illustrated in the example below where we specify `nsim = 10` to request that each feature be permuted 10 times and the results averaged together. (Additionally, if `nsim >1`, you can set `geom = "boxplot"` in the call to `vip()` to construct boxplots of the raw permutation-based VI scores. This is useful if you want to visualize the variability in each of the VI estimates; see Figure **??** for an example.)

```
# Use 10 Monte Carlo reps
set.seed(403)  # for reproducibility
```

```
vis <- vi(pp, method = "permute", target = "y", metric = "rsquared",
          pred_wrapper = predict, nsim = 15)
vip(vis, geom = "boxplot")
```



All available performance metrics for regression and classification can be listed using the `list_metrics()` function, for example:

```
list_metrics()

#>      Metric                      Description                          Task
#> 1  accuracy          Classification accuracy Binary/multiclass classification
#> 2     error          Misclassification error Binary/multiclass classification
#> 3       auc           Area under (ROC) curve            Binary classification
#> 4   logloss                         Log loss            Binary classification
#> 5      mauc Multiclass area under (ROC) curve       Multiclass classification
#> 6       mae              Mean absolute error                       Regression
#> 7       mse               Mean squared error                       Regression
#> 8        r2                        R squared                       Regression
#> 9  rsquared                        R squared                       Regression
#> 10     rmse          Root mean squared error                       Regression
#> 11      sse            Sum of squared errors                       Regression
```

We can also use a custom metric (i.e., loss function). Suppose for example you want to measure importance using the *mean absolute error* (MAE):

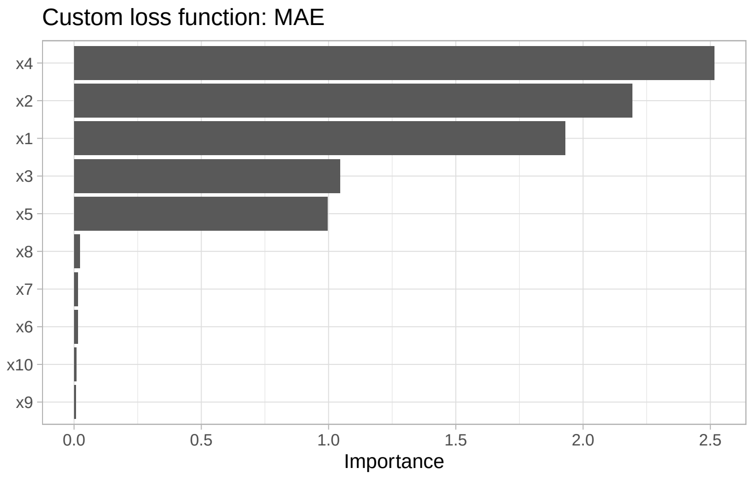
$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|Y_i - \widehat{f}\left(\boldsymbol{X}_i\right)\right|, \tag{2}$$

where $\widehat{f}\left(\boldsymbol{X}_i\right)$ is the predicted value of Y_i. A simple function implementing this metric is given below (note that, according to the documentation in `?vi_permute`, this function requires two arguments: `actual` and `predicted`).

```
mae <- function(actual, predicted) {
  mean(abs(actual - predicted))
}
```

To use this for computing permutation-based VI scores just pass it via the `metric` argument (be warned, however, that the metric used for computing permutation importance should be the same as the metric used to train and tune the model). Also, since this is a custom metric, we need to specify whether a smaller value indicates better performance by setting `smaller_is_better = TRUE`. The results, which are displayed in Figure **??**, are similar to those in Figure 11, albeit a different scale.

```
# Figure 12
set.seed(2321)  # for reproducibility
pfun <- function(object, newdata)  predict(object, newdata = newdata)
vip(nn, method = "permute", target = "y", metric = mae,
    smaller_is_better = TRUE, pred_wrapper = pfun) +
  ggtitle("Custom loss function: MAE")
```



Although permutation importance is most naturally computed on the training data, it may also be useful to do the shuffling and measure performance on new data! This is discussed in depth in Molnar (2019)[Chapter 5, Section 5.2]. For users interested in computing permutation importance using new data, just supply it to the `train` argument in the call to `vi()`, `vip()`, or `vi_permute()`. For instance, suppose we wanted to only use a fraction of the original training data to carry out the computations. In this case, we could simply pass the sampled data to the `train` argument as follows:

```
# Figure 13
set.seed(2327)  # for reproducibility
vip(nn, method = "permute", pred_wrapper = pfun, target = "y", metric = "rmse",
    train = trn[sample(nrow(trn), size = 400), ])  # sample 400 observations
```

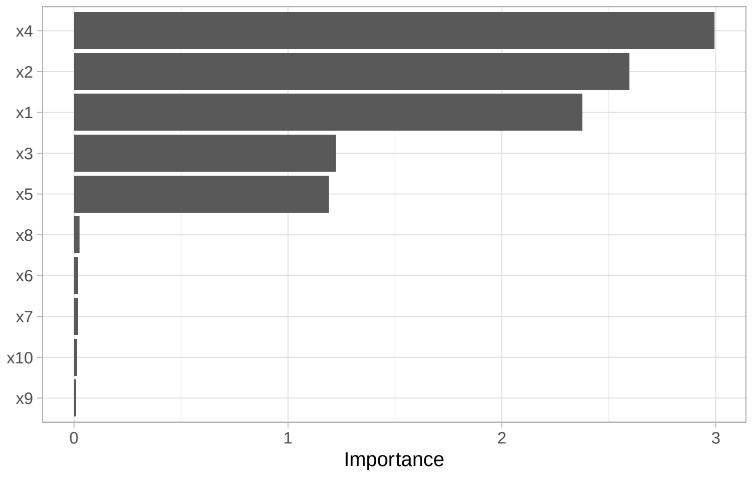


Figure 12: Permutation-based feature importance for the NN model fit to the simulated Friedman data. In this example, permutation importance is based on a random sample of 400 training observations.

```
  ggtitle("Using a random subset of training data")

#> $title
#> [1] "Using a random subset of training data"
#>
#> attr(,"class")
#> [1] "labels"
```

When using the permutation method with `nsim >1`, the default is to keep all the permutation scores as an attribute called `"raw_scores"`; you can turn this behavior off by setting `keep = FALSE` in the call to `vi_permute()`, `vi()`, or `vip()`. If `keep = TRUE` and `nsim >1`, you can request all permutation scores to be plotted by setting `all_permutation = TRUE` in the call to `vip()` as demonstrated in the code chunk below (see Figure 13). This let's you visually inspect the variability in the permutation scores within each feature.

```
# Figure 14
set.seed(8264)  # for reproducibility
vip(nn, method = "permute", pred_wrapper = pfun, target = "y", metric = "mae",
    geom = "point", all_permutations = TRUE, jitter = TRUE) +
  ggtitle("Plotting all permutation scores")
```

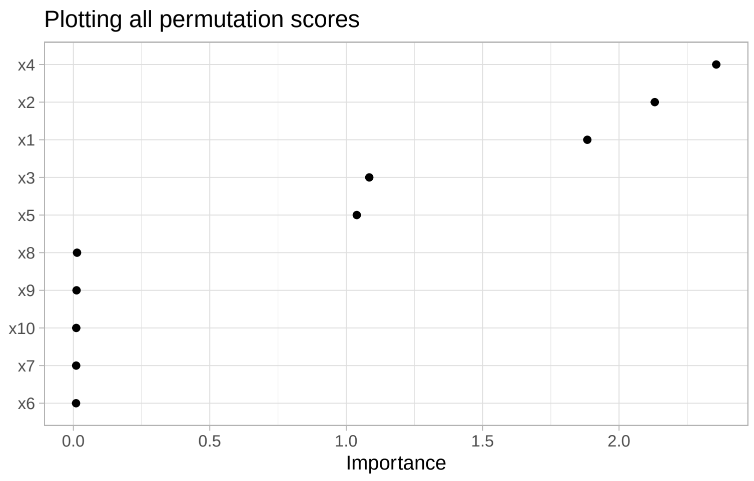


Figure 13: Permutation-based feature importance for the NN model fit to the simulated Friedman data. In this example, all the permutation importance scores (points) are displayed for each feature along with their average (bars).

Benchmarks

In this section, we compare the performance of three implementations of permutation-based VI scores: `iml::FeatureImp()` (version 0.9.0), `ingredients::feature_importance()` (version 0.5.0), and `vip::vi()` (version 0.1.3.9000).

We simulated 10,000 training observations from the Friedman 1 benchmark problem and trained a random forest using the **ranger** package. For each implementation, we computed permutation-based VI scores 100 times using the **microbenchmark** package (Mersmann, 2018). For this benchmark we did not use any of the parallel processing capability available in the **iml** and **vip** implementations. The results from **microbenchmark** are displayed in Figure 14 and summarized in the output below. In this case, the **vip** package (version 0.1.4) was the most efficient, followed closely by **ingredients**. It should be noted, however, that the implementations in **vip** and **iml** can be parallelized. To the best of our knowledge, this is not the case for **ingredients**.

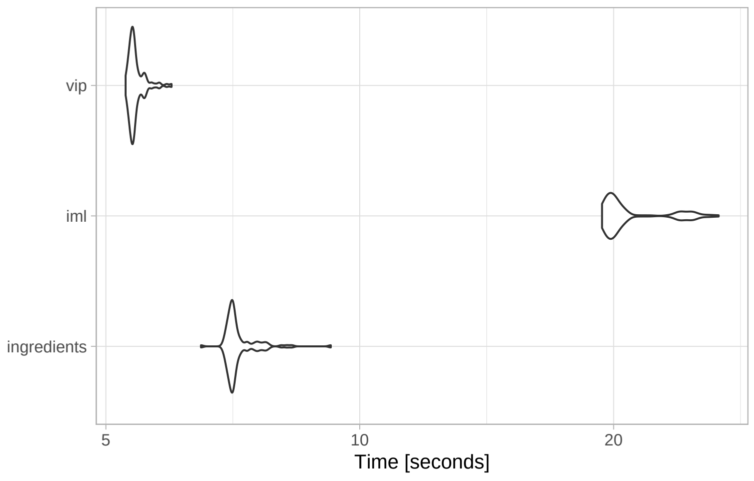


Figure 14: Violin plots comparing the computation time from three different implementations of permutation-based VI scores across 100 simulations.

Shapley method

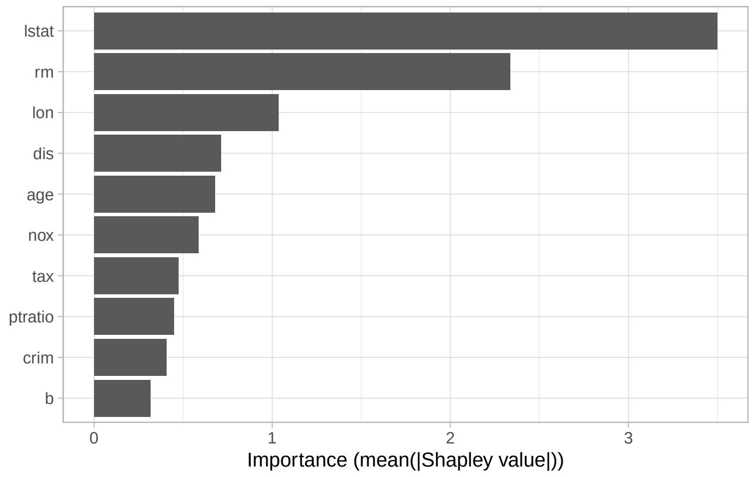
Starting with **vip** 0.1.4 you can now construct VI scores based on the popular *Shapley values* Molnar (2019, Chapter 5, Section 5.9)—a method from coalitional game theory. Using Shapley values, the prediction for a single instance $x^\star$ can be explained by assuming that each feature value in $x^\star$ is a "player" in a game with a payout equal to the corresponding prediction $\widehat{f}\left(x^\star\right)$. Shapley values tell us how to fairly distribute the "payout" among the features. The Shapley-based variable importance for a particular feature is computes as the mean of the abosulte value of the iindividual Shapley values across the entire training set. Although Shapley values can be computationally expensive, the **fastshap** package (Greenwell) exploits a few computational tricks and also includes the option to perform computations in parallel. Also, fast and exact algorithms (Lundberg et al., 2019) can be exploited for certain classes of models, which is illustrated below using the well-known Boston housing data (Figure **??**); see `?fastshap::explain` for details. (**Note:** specifying `include_type = TRUE` in the call to `vip::vip()` causes the type of variable importance computed to be displayed as part of the axis label.)

```
# Load required packages
library(xgboost)

# Load the Boston housing data
# install.packages("pdp)
data(boston, package = "pdp")
X <- data.matrix(subset(boston, select = -cmedv))  # matrix of feature values

# Fit a gradient boosted regression tree ensemble; hyperparameters were tuned
# using `autoxgb::autoxgb()`
set.seed(859)  # for reproducibility
bst <- xgboost(X, label = boston$cmedv, nrounds = 338, max_depth = 3, eta = 0.1,
               verbose = 0)

# Construct Shapley-based variable importance plot
vip(bst, method = "shap", train = X, exact = TRUE, include_type = TRUE)
```

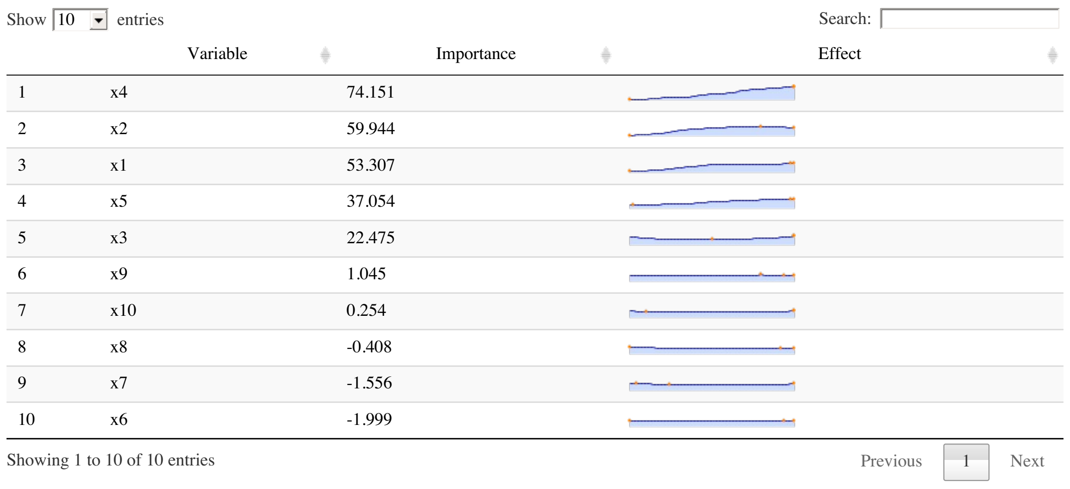


Use sparklines to characterize feature effects

Starting with **vip** 0.1.3, we have included a new function `add_sparklines()` for constructing HTML-based VI tables; however, this feature requires the **DT** package (Xie et al., 2018). The primary difference between `vi()` and `add_sparklines()` is that the latter includes an `Effect` column that displays a sparkline representation of the partial dependence function for each feature. This is a concise way to display both feature importance and feature effect information in a single (interactive) table. See `?vip::add_sparklines` for details. We illustrate the basic use of `add_sparklines()` in the code chunks below.

```
# First, compute a tibble of variable importance scores using any method
var_imp <- vi(rfo)

# Next, convert to an html-based data table with sparklines
add_sparklines(var_imp, fit = rfo)
```


Show 10 entries Search:

	Variable	Importance	Effect
1	x4	74.151	
2	x2	59.944	
3	x1	53.307	
4	x5	37.054	
5	x3	22.475	
6	x9	1.045	
7	x10	0.254	
8	x8	-0.408	
9	x7	-1.556	
10	x6	-1.999	

Showing 1 to 10 of 10 entries Previous 1 Next

Figure 15: Variable importance scores along with a sparkline representation of feature effects.

Ames housing example

For illustration, we'll use the Ames housing data which are available in the **AmesHousing** package (Kuhn, 2017a). 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. 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).

Using the R package **SuperLearner** (Polley et al., 2018), we trained five models using 5-fold cross-validation: a GBM using the **xgboost** package (Chen et al., 2017), an RF using the **ranger** package (Wright, 2017), a MARS model using the **earth** package, a GLMNET model using the **glmnet** package (Friedman et al., 2010), and a support vector regression model using the **kernlab** package (Karatzoglou et al., 2004). The coefficients from the meta learner indicate that the GBM contributes the most to new predictions, followed by the MARS model, RF, and the GLMNET. The support vector regression model did not contribute to the ensemble.

```
# Load the Ames housing data
ames <- AmesHousing::make_ames()
X <- subset(ames, select = -Sale_Price)
y <- ames$Sale_Price

# Load required packages
library(SuperLearner)

# List of base learners
learners <- c("SL.xgboost", "SL.ranger", "SL.earth", "SL.glmnet", "SL.ksvm")

# Stack models
set.seed(840)
ctrl <- SuperLearner.CV.control(V = 5L, shuffle = TRUE)
sl <- SuperLearner(Y = y, X = X, SL.library = learners, verbose = TRUE,
                   cvControl = ctrl)
sl

#>
#> Call:
#> SuperLearner(Y = y, X = X, SL.library = learners, verbose = TRUE, cvControl = ctrl)
#>
#>
#>
#>                      Risk       Coef
#> SL.xgboost_All  569646381 0.43455425
#> SL.ranger_All   666208088 0.06970309
#> SL.earth_All    553872844 0.49574265
#> SL.glmnet_All   908881559 0.00000000
#> SL.ksvm_All    6784289108 0.00000000
```

In the code chunks below we request permutation-based VI scores and add sparkline representations of the PDPs for the top ten features. For this we need to define a couple of wrapper functions. One for computing predictions (for the permutation VI scores) and one for computing averaged predictions (for the PDPs):

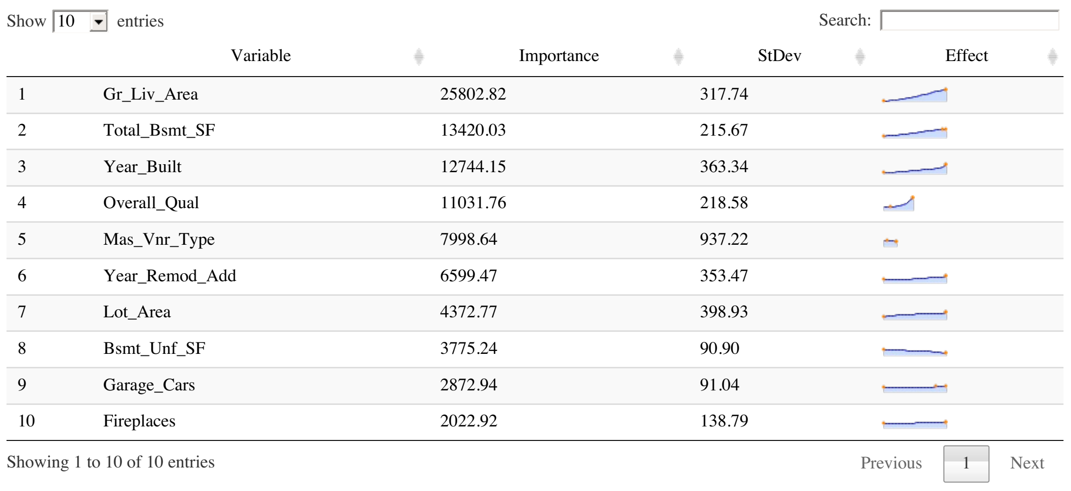
```
# Prediction wrapper functions
imp_fun <- function(object, newdata) {  # for permutation-based VI scores
  predict(object, newdata = newdata)$pred
}
par_fun <- function(object, newdata) {  # for PDPs
  mean(predict(object, newdata = newdata)$pred)
}
```

To speed up the process, we perform the computations in parallel by setting `parallel = TRUE` in the call to `vi()` and `add_sparklines()`. Note that we first need to set up a parallel backend for this to work. Both **vip** and **pdp** use **plyr** (Wickham, 2011)—which relies on **foreach** (Revolution Analytics and Weston, 2015b)—so any parallel backend supported by the **foreach** package should work. Below we use a *socket* approach with the **doParallel** backend (Revolution Analytics and Weston, 2015a) using a cluster of size five.

```
# Setup parallel backend
library(doParallel) # load the parallel backend
cl <- makeCluster(5) # use 5 workers
registerDoParallel(cl) # register the parallel backend

# Permutation-based feature importance
set.seed(278)
var_imp <- vi(sl, method = "permute", train = X, target = y, metric = "rmse",
              pred_wrapper = imp_fun, nsim = 5, parallel = TRUE)

# Add sparkline representation of feature effects
add_sparklines(var_imp[1L:10L, ], fit = sl, pred.fun = par_fun, train = X,
               digits = 2, verbose = TRUE, trim.outliers = TRUE,
               grid.resolution = 20, parallel = TRUE)
```


Show 10 entries Search:

	Variable	Importance	StDev	Effect
1	Gr_Liv_Area	25802.82	317.74	
2	Total_Bsmt_SF	13420.03	215.67	
3	Year_Built	12744.15	363.34	
4	Overall_Qual	11031.76	218.58	
5	Mas_Vnr_Type	7998.64	937.22	
6	Year_Remod_Add	6599.47	353.47	
7	Lot_Area	4372.77	398.93	
8	Bsmt_Unf_SF	3775.24	90.90	
9	Garage_Cars	2872.94	91.04	
10	Fireplaces	2022.92	138.79	

Showing 1 to 10 of 10 entries Previous 1 Next

Figure 16: VIP with sparkline representation of feature effects for the top ten features from a Super Learner fit to the Ames housing data.

```
# Shut down cluster
stopCluster(cl)
```

Summary

VIPs help to visualize the strength of the relationship between each feature and the response, while accounting for all the other features in the model. We've discussed two types of VI: model-specific and model-agnostic, as well as some of their strengths and weaknesses. In this paper, we showed how to construct VIPs for various types of "black box" models in R using the **vip** package. We also briefly discussed related approaches available in other R packages. Suggestions to avoid high execution times were discussed and demonstrated via examples. This paper is based on **vip** version 0.1.3. For updates that have occurred since then, see the package's NEWS file. In terms of future development, **vip** can be expanded in a number of ways. For example, we plan to incorporate the option to compute group-based permutation scores. We're also working on including a new option for efficiently computing Shapley-based VI scores via the new fastshap package. Although not discussed in this paper, the package also includes a promising statistic (similar to the PDP- and ICE-based VI scores previously discussed) for measuring the relative strength of interaction between features.

Acknowledgments

TBD.

Bibliography

P. Biecek. Dalex: Explainers for complex predictive models in r. *Journal of Machine Learning Research*, 19 (84):1–5, 2018. URL http://jmlr.org/papers/v19/18-416.html. [p1]

P. Biecek. *ingredients: Effects and Importances of Model Ingredients*, 2019. URL https://CRAN.R-project.org/package=ingredients. R package version 0.3.3. [p1]

B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, and Z. M. Jones. mlr: Machine learning in r. *Journal of Machine Learning Research*, 17(170):1–5, 2016. URL http://jmlr.org/papers/v17/15-066.html. [p1]

L. Breiman. Bagging predictors. *Machine Learning*, 8(2):209–218, 1996. URL https://doi.org/10.1023/A:1018054314350. [p2]

L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. URL https://doi.org/10.1023/A:1010933404324. [p11]

L. Breiman, J. Friedman, and R. A. O. Charles J. Stone. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984. ISBN 9780412048418. [p3]

W. Chang. *R6: Encapsulated Classes with Reference Semantics*, 2019. URL https://CRAN.R-project.org/package=R6. R package version 2.4.0. [p1]

T. Chen, T. He, and M. Benesty. *xgboost: Extreme Gradient Boosting*, 2017. URL https://CRAN.R-project.org/package=xgboost. R package version 0.6-4. [p17]

W. S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–836, 1979. doi: 10.1080/01621459.1979.10481038. [p1]

A. Fisher, C. Rudin, and F. Dominici. Model class reliance: Variable importance measures for any machine learning model class, from the "rashomon" perspective. *arXiv preprint arXiv:1801.01489*, 2018. [p11]

J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010. URL http://www.jstatsoft.org/v33/i01/. [p17]

J. H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–67, 1991. URL https://doi.org/10.1214/aos/1176347963. [p2, 6]

D. G. Garson. Interpreting neural-network connection weights. *Artificial Intelligence Expert*, 6(4):46–51, 1991. [p7]

T. Gedeon. Data mining of inputs: Analysing magnitude and functional measures. *International Journal of Neural Systems*, 24(2):123–140, 1997. URL https://doi.org/10.1007/s10994-006-6226-1. [p7]

A. Goh. Back-propagation neural networks for modeling complex systems. *Artificial Intelligence in Engineering*, 9(3):143–151, 1995. doi: http://dx.doi.org/10.1016/0954-1810(94)00011-S. [p7]

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. [p9]

B. Greenwell. *fastshap: Fast Approximate Shapley Values*. URL https://github.com/bgreenwell/fastshap. R package version 0.0.3.9000. [p16]

B. Greenwell and B. Boehmke. *vip: Variable Importance Plots*, 2018. URL https://CRAN.R-project.org/package=vip. R package version 0.1.0. [p2]

B. M. Greenwell. pdp: An r package for constructing partial dependence plots. *The R Journal*, 9(1):421–436, 2017. URL https://journal.r-project.org/archive/2017/RJ-2017-016/index.html. [p8]

B. M. Greenwell, B. C. Boehmke, and A. J. McCarthy. A simple and effective model-based variable importance measure. *arXiv preprint arXiv:1805.04755*, 2018. [p2, 8]

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer-Verlag, 2009. [p3]

A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis. Kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9):1–20, 2004. URL https://doi.org/10.18637/jss.v011.i09. [p17]

M. Kuhn. *AmesHousing: The Ames Iowa Housing Data*, 2017a. URL https://CRAN.R-project.org/package=AmesHousing. R package version 0.0.3. [p17]

M. Kuhn. *caret: Classification and Regression Training*, 2017b. URL https://CRAN.R-project.org/package=caret. R package version 6.0-76. [p1]

M. Kuhn and K. Johnson. *Applied Predictive Modeling*. SpringerLink : Bücher. Springer New York, 2013. ISBN 9781461468493. [p1]

S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee. Explainable ai for trees: From local explanations to global understanding. *arXiv preprint arXiv:1905.04610*, 2019. [p16]

O. Mersmann. *microbenchmark: Accurate Timing Functions*, 2018. URL https://CRAN.R-project.org/package=microbenchmark. R package version 1.4-6. [p15]

S. Milborrow. *earth: Multivariate Adaptive Regression Splines*, 2017. URL https://CRAN.R-project.org/package=earth. R package version 4.5.0. [p6]

C. Molnar. *Interpretable Machine Learning*. 2019. https://christophm.github.io/interpretable-ml-book/. [p1, 12, 14, 16]

C. Molnar, B. Bischl, and G. Casalicchio. iml: An r package for interpretable machine learning. *JOSS*, 3(26):786, 2018. doi: 10.21105/joss.00786. URL http://joss.theoj.org/papers/10.21105/joss.00786. [p1]

K. Müller and H. Wickham. *tibble: Simple Data Frames*, 2019. URL https://CRAN.R-project.org/package=tibble. R package version 2.1.1. [p2]

J. D. Olden, M. K. Joy, and R. G. Death. An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data. *Ecological Modelling*, 178(3):389–397, 2004. doi: http://dx.doi.org/10.1016/j.ecolmodel.2004.03.013. [p7]

E. Polley, E. LeDell, C. Kennedy, and M. van der Laan. *SuperLearner: Super Learner Prediction*, 2018. URL https://CRAN.R-project.org/package=SuperLearner. R package version 2.0-24. [p17]

Revolution Analytics and S. Weston. *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*, 2015a. URL https://CRAN.R-project.org/package=doParallel. R package version 1.0.10. [p18]

Revolution Analytics and S. Weston. *foreach: Provides Foreach Looping Construct for R*, 2015b. URL https://CRAN.R-project.org/package=foreach. R package version 1.4.3. [p18]

G. Ridgeway. *gbm: Generalized Boosted Regression Models*, 2017. URL https://CRAN.R-project.org/package=gbm. R package version 2.1.3. [p1]

C. A. Scholbeck, C. Molnar, C. Heumann, B. Bischl, and G. Casalicchio. Sampling, intervention, prediction, aggregation: A generalized framework for model agnostic interpretations. *CoRR*, abs/1904.03959, 2019. URL `http://arxiv.org/abs/1904.03959`. [p8]

W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. URL `http://www.stats.ox.ac.uk/pub/MASS4`. ISBN 0-387-95457-0. [p7]

H. Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1): 1–29, 2011. URL `https://doi.org/10.18637/jss.v040.i01`. [p18]

H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. URL `https://ggplot2.tidyverse.org`. [p2]

M. N. Wright. *ranger: A Fast Implementation of Random Forests*, 2017. URL `https://CRAN.R-project.org/package=ranger`. R package version 0.7.0. [p17]

Y. Xie, J. Cheng, and X. Tan. *DT: A Wrapper of the JavaScript Library 'DataTables'*, 2018. URL `https://CRAN.R-project.org/package=DT`. R package version 0.5. [p16]

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`

Bradley C. Boehmke
University of Cincinnati
2925 Campus Green Dr
Cincinnati, OH 45221
United States of America
ORCiD—0000-0002-3611-8516
`bradleyboehmke@gmail.com`