

pdp: An R Package for Constructing Partial Dependence Plots

by Brandon M. Greenwell

Abstract Complex nonparametric models—like neural networks, random forests, and support vector machines—are more common than ever in predictive analytics, especially when dealing with large observational databases that don't adhere to the strict assumptions imposed by traditional statistical techniques (e.g., multiple linear regression which assumes linearity, homoscedasticity, and normality). Unfortunately, it can be challenging to understand the results of such models and explain them to management. Partial dependence plots offer a simple solution. Partial dependence plots are low-dimensional graphical renderings of the prediction function $\hat{f}(x)$ so that the relationship between the outcome and predictors of interest can be more easily understood. These plots are especially useful in explaining the output from black box models. In this paper, we introduce **pdp**, a general R package for constructing partial dependence plots.

Introduction

Harrison and Rubinfeld (1978) were among the first to analyze the well-known Boston housing data. One of their goals was to find a housing value equation using data on median housing values from $n = 506$ census tracts in the suburbs of Boston from the 1970 census; see Table IV of Harrison and Rubinfeld (1978) for a description of each variable. The data violate many classical assumptions like linearity, normality, and constant variance. Nonetheless, Harrison and Rubinfeld—using a combination of transformations, significance testing, and grid searches—were able to find a reasonable fitting model ($R^2 = 0.81$). Part of the payoff for their efforts was an interpretable prediction equation which is reproduced in Equation (1).

$$\begin{aligned} \log(\widehat{MV}) = & 9.76 + 0.0063RM^2 + 8.98 \times 10^{-5}AGE - 0.19 \log(DIS) + 0.096 \log(RAD) \\ & - 4.20 \times 10^{-4}TAX - 0.031PTRATIO + 0.36(B - 0.63)^2 - 0.37 \log(LSTAT) \\ & - 0.012CRIM + 8.03 \times 10^{-5}ZN + 2.41 \times 10^{-4}INDUS + 0.088CHAS - 0.0064NOX^2 \end{aligned} \quad (1)$$

Nowadays, supervised learning algorithms can fit the data automatically in seconds—typically with higher accuracy. (We will revisit the Boston housing data in Section 2.2.) The downfall, however, is some loss of interpretation; for example, there is usually no simple prediction formula like in Equation (1). Fortunately, these methods can still provide insight into the data, but it is not in the form of simple equations. For example, many modern methods, like boosting and random forests, can assign importance scores to all of the predictors in the training data.

Determining predictor importance is a crucial task in any supervised learning problem. However, ranking variables is only part of the story and once a subset of "important" features is identified it is often necessary to assess the relationship between them (or subset thereof) and the response. This can be done in many ways, but in machine learning it is often accomplished by constructing *partial dependence plots* (PDPs); see Friedman (2001) for details. PDPs help visualize the relationship between a subset of the features (typically 1-3) and the response while accounting for the average effect of the other predictors in the model. They are particularly effective with black box models like random forests and support vector machines.

Let $x = \{x_1, x_2, \dots, x_p\}$ represent the predictors in a model whose prediction function is $\hat{f}(x)$. If we partition x into an interest set, z_s , and its complement, $z_c = x \setminus z_s$, then the "partial dependence" of the response on z_s is defined as

$$f_s(z_s) = E_{z_c} [\hat{f}(z_s, z_c)] = \int \hat{f}(z_s, z_c) p_c(z_c) dz_c, \quad (2)$$

where $p_c(z_c)$ is the marginal probability density of z_c : $p_c(z_c) = \int p(x) dz_s$. Equation (2) can be estimated from a set of training data by

$$\bar{f}_s(z_s) = \frac{1}{n} \sum_{i=1}^n \hat{f}(z_s, z_{i,c}), \quad (3)$$

where $z_{i,c}$ ($i = 1, 2, \dots, n$) are the values of z_c that occur in the training sample; that is, we average out

the effects of all the other predictors in the model.

Constructing a PDP (3) in practice is rather straightforward. To simplify, let $z_s = x_1$ be the predictor variable of interest with unique values $\{x_{11}, x_{12}, \dots, x_{1k}\}$. The partial dependence of the response on x_1 can be constructed as follows:

1. For $i \in \{1, 2, \dots, k\}$:
 - (a) Copy the training data and replace the original values of x_1 with the constant x_{1i} .
 - (b) Compute the vector of predicted values from the modified copy of the training data.
 - (c) Compute the average prediction to obtain $\bar{f}_1(x_{1i})$.
2. Plot the pairs $\{x_{1i}, \bar{f}_1(x_{1i})\}$ for $i = 1, 2, \dots, k$.

Algorithm 1: A simple algorithm for constructing the partial dependence of the response on a single predictor x_1 .

Algorithm 1 can be quite computationally intensive since it involves k passes over the training records. Fortunately, this can be parallelized quite easily (more on this in a later section). This algorithm can also be easily extended to larger subsets of two or more features as well.

Limited implementations of Friedman's PDPs are available in packages **randomForest** (Liaw and Wiener, 2002) and **gbm**, among others; these are all limited in the sense that they only apply to the models fit using the respective package. For example, the `partialPlot` function in **randomForest** only applies to objects of class "randomForest" and the `plot` function in **gbm** only applies to "gbm" objects. While the **randomForest** implementation will only allow for a single predictor, the **gbm** implementation can deal with any subset of the predictor space. Partial dependence functions are not restricted to tree-based models; they can be applied to any supervised learning algorithm (e.g., multiple linear regression and neural networks). However, to our knowledge, there is no general package for constructing PDPs in R. For example, PDPs for a *conditional random forest* as implemented by the `cforest` function in the **party** and **partykit** packages; see Torsten Hothorn and Zeileis (2015) and Hothorn and Zeileis (2016), respectively. The **pdp** (Greenwell, 2016) package tries to close this gap by offering a general framework for constructing PDPs that can be applied to several classes of fitted models.

The **plotmo** package (Milborrow, 2015) is one alternative to **pdp**. According to its author, **plotmo** constructs "a poor man's partial dependence plot." In particular, it plots a model's response when varying one or two predictors while holding the other predictors in the model constant (continuous features are fixed at their median value, while factors are held at their first level). These plots allow for up to two variables at a time. They are also less accurate than PDPs, but are faster to construct. For additive models (i.e., models with no interactions), these plots are identical in shape to PDPs.

PDPs can be misleading in the presence of substantial interactions (Goldstein et al., 2015). To overcome this issue Goldstein, Kapelner, Bleich, and Pitkin developed the concept of *individual conditional expectation* (ICE) plots—available in the **ICEbox** package. **ICEbox** only allows for one variable at a time (i.e., no multivariate displays), though color can be used effectively to display information about an additional predictor. The ability to construct derivative plots is also available in **ICEbox**.

Many other techniques exist for visualizing relationships between the predictors and the response based on a fitted model. For example, the **car** package (Fox and Weisberg, 2011) contains many functions for constructing partial-residual and marginal-model plots. The **effects** package (Fox, 2003) is also of interest. It provides tabular and graphical displays for the terms in parametric models. However, these methods were designed for simpler parametric models (e.g., linear and generalized linear models), whereas **plotmo**, **ICEbox**, and **pdp** are more useful for black box models (although, they can be used for simple parametric models as well).

Constructing PDPs in R

The **pdp** package is useful for constructing PDPs for many classes of fitted models in R. PDPs are especially useful for visualizing the relationships discovered by complex machine learning algorithms such as a random forest. The latest stable release is available from CRAN. The development version is located on GitHub: <https://github.com/bgreenwell/pdp>. Bug reports and suggestions are appreciated and should be submitted to <https://github.com/bgreenwell/pdp/issues>. Currently, only two

functions are exported by **pdp**:

- `partial`
- `plotPartial`

The `partial` function evaluates the partial dependence (3) from a fitted model over a grid of predictor values; the fitted model and predictors are specified using the `object` and `pred.var` arguments, respectively—these are the only required arguments. If `plot = FALSE` (the default), `partial` returns an object of class "partial" which inherits from the class "data.frame"; put another way, by default, `partial` returns a data frame with an additional class that is recognized by the `plotPartial` function. The columns of the data frame are labeled in the same order as the features supplied to `pred.var`, and the last column is always labeled `yhat` and contains the values of the partial dependence function $\hat{f}_s(z_s)$. If `plot = TRUE`, then `partial` makes an internal call to `plotPartial` and returns the PDP in the form of a **lattice** plot (Sarkar, 2008).

The `plotPartial` function can be used for displaying more advanced PDPs; it operates on objects of class "partial" and has many useful plotting options. For example, `plotPartial` makes it straight forward to add a LOESS smooth, or produce a 3-D surface instead of a false color level plot (the default). Of course, since the default output produced by `partial` is still a data frame, the user can easily use any plotting package he/she desires to visualize the results—**ggplot2** (Wickham, 2009), for instance.

Note: as mentioned above, **pdp** relies on **lattice** for its graphics. **lattice** itself is built on top of **grid** (R Core Team, 2016). **grid** graphics behave a little differently than traditional R graphics, and two points are worth making (see `?lattice` for more details):

1. **lattice** functions return a "trellis" object, but do not display it; the `print` method produces the actual display. However, due to R's automatic printing rule, the result is automatically printed when using these functions in the command line. If `plotPartial` is called inside of source or inside a loop (e.g., `for` or `while`), an explicit `print` statement is required to display the resulting graph; hence, the same is true when using `partial` with `plot = TRUE`.
2. Setting graphical parameters via `par` typically has no effect on **lattice** plots. Instead, **lattice** provides its own `trellis.par.set` function for modifying graphical parameters.

A consequence of the second point is that the `par` function cannot be used to control the layout of multiple **lattice** (and hence **pdp**) plots. Simple solutions are available in packages **latticeExtra** (Sarkar and Andrews, 2016) and **gridExtra** (Auguie, 2016). For convenience, **pdp** imports the `grid.arrange` function from **gridExtra** which makes it easy to display multiple **pdp** plots on a single page.

Currently supported models are described in Table 1. In a lot of these cases, `partial` should be able to automatically determine appropriate values for the options `type` (i.e., "regression" or "classification") and `train` (the original training data). In other situations, the user may need to specify one or both of these additional arguments. These two arguments allow `partial` to be flexible enough to handle many of the model types not listed in Table 1; for example, neural networks from the **nnet** package (Venables and Ripley, 2002) and projection pursuit regression (Friedman and Stuetzle, 1981) using the `ppr` function in the **stats** package.

When additional arguments are necessary, the user will be prompted with an informative error message. For instance, the user may occasionally see the following:

```
Error: The training data could not be extracted from object. Please supply
the raw training data using the `train` argument in the call to `partial`.
```

This will occur, for example, when using `partial` with the "xgb.Booster" class (see Section 2.2.7). In fact, using `partial` with any object that does not store the original training data will result in the above error message.

The `partial` function also supports objects of class "train" produced using the `train` function from the well-known **caret** package (Kuhn, 2016). This means that `partial` can be used with any classification or regression model that has been fit using **caret**'s `train` function; see <http://topepo.github.io/caret/available-models.html> for a current list of models supported by **caret**. An example is given in Section 2.2.7.

For illustration, we will use the (corrected) Boston housing data which are available from the **mlbench** package (Dimitriadou, 2010). These data contain the median value of owner-occupied homes in 506 U.S. census tracts in the Boston area, along with 13 independent variables such as the per capita crime rate by town. These are the same data analyzed in Harrison and Rubinfeld (1978), but with a few corrections and additional spatial variables. We begin by loading the data and omitting unimportant columns.

```
data(BostonHousing2, package = "mlbench") # mlbench must be installed!
boston <- BostonHousing2[, -c(1, 2, 5)]
```

Type of model	R package	Object class
Decision tree	C50 (Kuhn et al., 2015)	"C5.0"
	party	"BinaryTree"
	partykit	"constparty"/"party"
	rpart (Therneau et al., 2015)	"rpart"
Bagged decision trees	adabag (Alfaro et al., 2013)	"bagging"
	ipred (Peters and Hothorn, 2015)	"classbagg", "regbagg"
Boosted decision trees	adabag (Alfaro et al., 2013)	"boosting"
	gbm	"gbm"
	xgboost	"xgb.Booster"
Cubist	Cubist (Kuhn et al., 2014)	"cubist"
Random forest	randomForest	"randomForest"
Conditional random forest	party	"RandomForest"
	partykit	"cforest"
Linear model	stats	"lm"
Generalized linear model	stats	"glm", "lme4"
Nonlinear least squares	stats	"nls"
Multivariate adaptive regression splines (MARS)	earth (Milborrow, 2016)	"earth"
Support vector machine	e1071 (Meyer et al., 2015)	"svm"
	kernlab (Karatzoglou et al., 2004)	"ksvm"

Table 1: Models specifically supported by the **pdp** package. **Note:** for some of these cases, the user may still need to supply additional arguments in the call to `partial`.

Next, we fit a random forest to the entire data set with default tuning parameters and 500 trees:

```
library(randomForest)
set.seed(101) # for reproducibility
boston.rf <- randomForest(cmedv ~ ., data = boston, importance = TRUE)
varImpPlot(boston.rf) # Figure 1
```

The model fit is reasonable, with an *out-of-bag* (pseudo) R^2 of 0.89. The variable importance scores are displayed in Figure 1. Both plots indicate that the percentage of lower status of the population (`lstat`) and the average number of rooms per dwelling (`rm`) are highly associated with the median value of owner-occupied homes (`cmedv`). The question then arises, "What is the nature of these associations?" To help answer this, we can look at the partial dependence of `cmedv` on `lstat` and `rm`, both individually and together.

Single predictor PDPs

As previously mentioned, the `randomForest` package has its own `partialPlot` function for visualizing the partial dependence of the response on a single predictor—the keywords here are "single predictor". For example, the following snippet of code plots the partial dependence of `cmedv` on `lstat`:

```
partialPlot(boston.rf, pred.data = boston, x.var = "lstat")
```

The same plot can be achieved using the `partial` function and setting `plot = TRUE` (see the left side of Figure 2):

```
library(pdp)
partial(boston.rf, pred.var = "lstat", plot = TRUE) # Figure 2 (left)
```

The only difference is that **pdp** uses the **lattice** graphics package to produce all of its displays.

For a more customizable plot, we can set `plot = FALSE` in the call to `partial` and then use the `plotPartial` function on the resulting data frame. This is illustrated in the example below which increases the line width, adds a LOESS smooth, and customizes the y -axis label. The result is displayed in the right side of Figure 2. **Note:** to encourage writing more readable code, the *pipe* operator `%>%` provided by the **magrittr** package (Bache and Wickham, 2014) is exported whenever **pdp** is loaded.

```
# Figure 2 (right)
boston.rf %>% # the %>% operator is read as "and then"
```

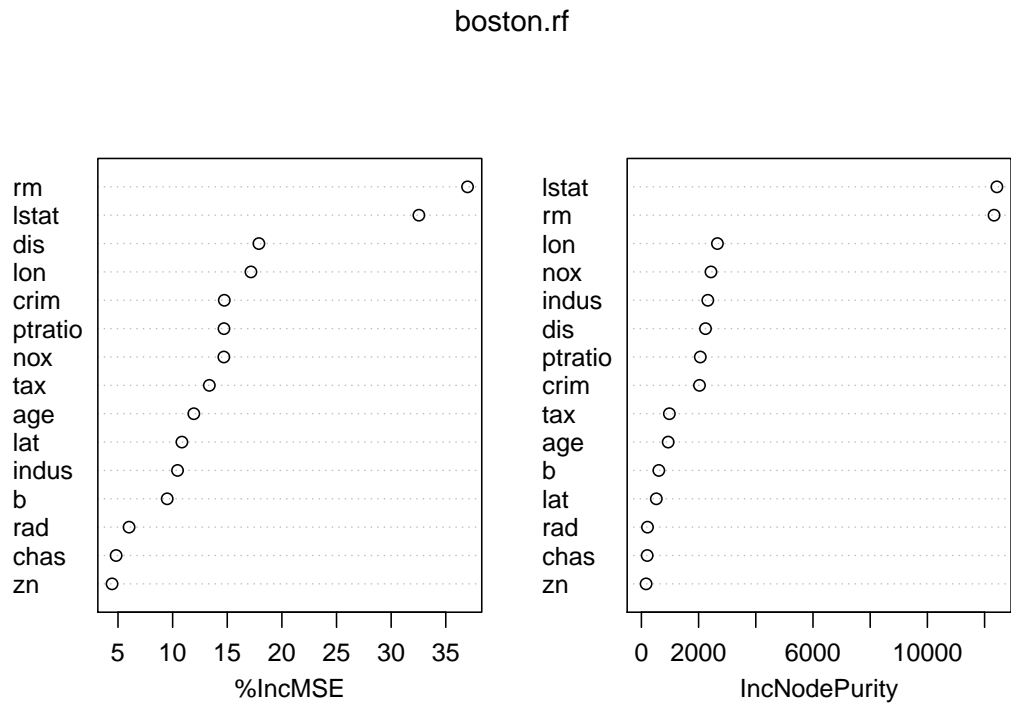


Figure 1: Dotchart of variable importance scores for the Boston housing data based on a random forest with 500 trees.

```
partial(pred.var = "lstat") %>%  
plotPartial(smooth = TRUE, lwd = 2, ylab = expression(f(lstat)))
```

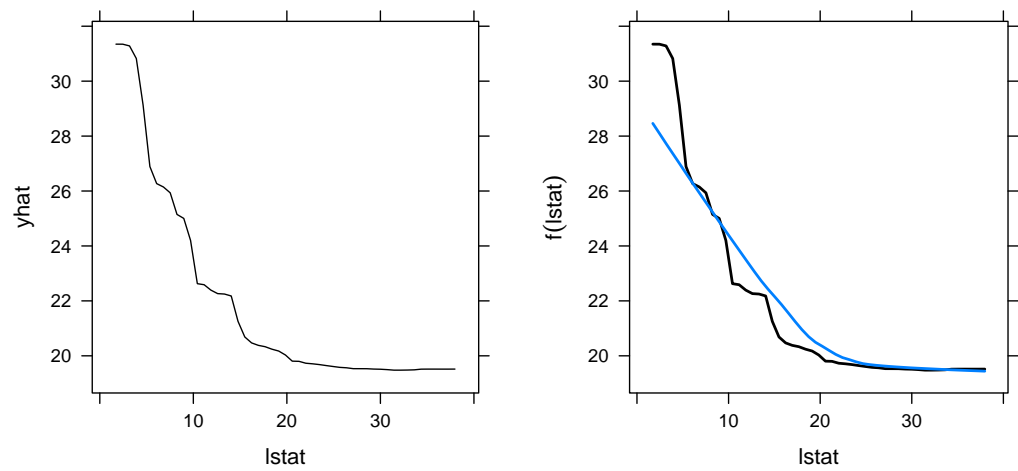


Figure 2: Partial dependence of cmedv on lstat based on a random forest. *Left:* Default plot. *Right:* Customized plot obtained using the plotPartial function.

Multi-predictor PDPs

The benefit of using partial is threefold: (1) it is a generic function that can be used for various types of fitted models (not just random forests), (2) it will allow for any number of predictors to be used (e.g., multivariate displays), and (3) it can utilize any of the parallel backends supported by the `foreach` package (Analytics and Weston, 2015c); we discuss parallel execution in a later section. For example, the following code chunk uses the random forest model to assess the joint effect of lstat and rm on

cmedv. The `grid.arrange` function is then used to display three PDPs, which make use of various `plotPartial` options¹, on the same graph. The result is displayed in Figure 3.

```

rwb <- colorRampPalette(c("red", "white", "blue"))
pd <- partial(boston.rf, pred.var = c("lstat", "rm"))
grid.arrange(
  # Figure 3 (left)
  plotPartial(pd),
  # Figure 3 (middle)
  plotPartial(pd, contour = TRUE, col.regions = rwb),
  # Figure 3 (right)
  plotPartial(pd, levelplot = FALSE, zlab = "cmedv", drape = TRUE,
              colorkey = TRUE, screen = list(z = -20, x = -60)),
  ncol = 3
)

```

Note: the default color map for level plots is the color blind-friendly `matplotlib` (Hunter, 2007) ‘`viridis`’ color map provided by the `viridis` package (Garnier, 2016).

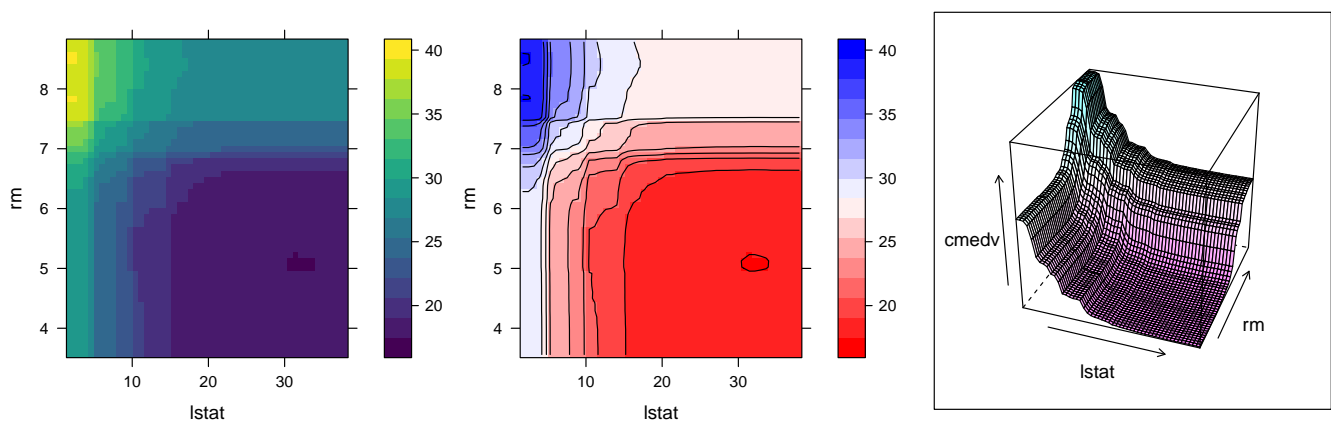


Figure 3: Partial dependence of `cmedv` on `lstat` and `rm` based on a random forest. *Left:* Default plot. *Middle:* With contour lines and a different color palette. *Right:* Using a 3-D surface.

Avoiding extrapolation

It is not wise to draw conclusions from PDPs in regions outside the area of the training data. Here we describe two ways to mitigate the risk of extrapolation in PDPs: rug displays and convex hulls. Rug displays are one-dimensional plots added to the axes. Both `partial` and `plotPartial` have a `rug` option that, when set to `TRUE`, will display the deciles of the distribution (as well as the minimum and maximum values) for the predictors on the horizontal and vertical axes. The following snippet of code produces the left display in Figure 4.

```

# Figure 4 (left)
partial(boston.rf, pred.var = "lstat", plot = TRUE, rug = TRUE)

```

In two or more dimensions, plotting the convex hull is more informative; it outlines the region of the predictor space that the model was trained on. When `chull = TRUE`, the convex hull of the first two dimensions of z_s (i.e., the first two variables supplied to `pred.var`) is computed; for example, if you set `chull = TRUE` in the call to `partial` only the region within the convex hull of the first two variables is plotted. Over interpreting the PDP outside of this region is considered extrapolation and is ill-advised. The right display in Figure 4 was produced using:

```

# Figure 4 (right)
partial(boston.rf, pred.var = c("lstat", "rm"), plot = TRUE, chull = TRUE)

```

¹See Section 2.2.4 for an example of how to add a label to the colorkey in these types of graphs.

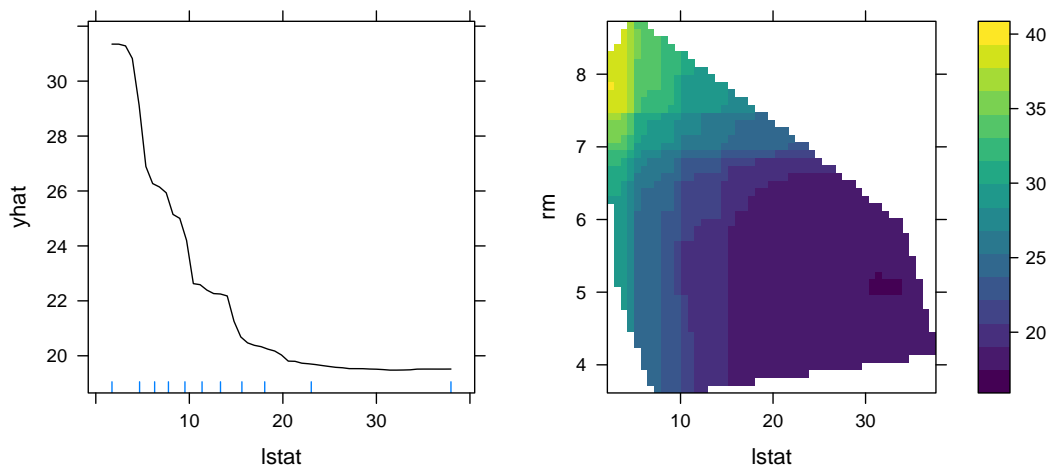


Figure 4: Examples of PDPs with the addition of a rug display (left) and a convex hull (right).

Addressing computational concerns

Constructing PDPs can be quite computationally expensive. Several strategies are available to ease the computational burden in larger problems. For example, there is no need to compute partial dependence of `cmdev` using each unique value of `rm` in the training data (which would require $k = 446$ passes over the data!). We could get very reasonable results using a reduced number of points. Current options are to use a grid of equally spaced values in the range of the variable of interest; the number of points can be controlled using the `grid.resolution` option in the call to `partial`. Alternatively, a user-specified grid of values (e.g., containing specific quantiles of interest) can be supplied through the `pred.grid` argument. To demonstrate, the following snippet of code computes the partial dependence of `cmdev` on `rm` using each option, then uses `grid.arrange` to display all three PDPs on the same page; the results are displayed in Figure 5.

```
grid.arrange(
  # Figure 5 (left)
  partial(boston.rf, "rm", plot = TRUE),
  # Figure 5 (middle)
  partial(boston.rf, "rm", grid.resolution = 30, plot = TRUE),
  # Figure 5 (right)
  partial(boston.rf, "rm", pred.grid = data.frame(rm = 3:9), plot = TRUE),
  ncol = 3
)
```

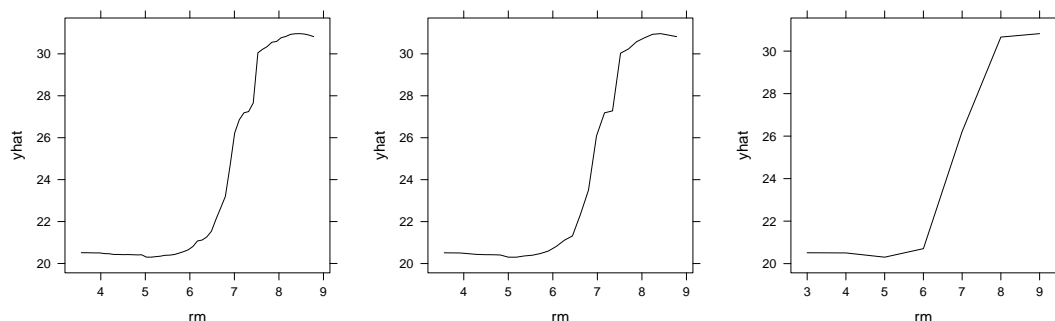


Figure 5: Partial dependence of `cmdev` on `rm`. *Left:* Default plot. *Middle:* Using a reduced grid size. *Right:* Using a user-specified grid.

The partial function relies on the `plyr` package (Wickham, 2011), rather than R's built-in for loops. This makes it easy to request progress bars (e.g., `progress = "text"`) or run `partial` in parallel. In fact, `partial` can use any of the parallel backends supported by the `foreach` package. To use this

functionality, we must first load and register a supported parallel backend [e.g., **doMC** (Analytics and Weston, 2015a) or **doParallel** (Analytics and Weston, 2015b)].

To illustrate, we will use the Los Angeles ozone pollution data described in Breiman and Friedman (1985). The data contain daily measurements of ozone concentration (ozone) along with eight meteorological quantities for 330 days in the Los Angeles basin in 1976. The data are available from <http://statweb.stanford.edu/~tibs/ElemStatLearn/datasets/LAozone.data>. Details, including variable information, are available from <http://statweb.stanford.edu/~tibs/ElemStatLearn/datasets/LAozone.info>. The following code chunk loads the data into R:

```
ozone <- read.csv(paste0("http://statweb.stanford.edu/~tibs/ElemStatLearn/",
                        "datasets/LAozone.data"), header = TRUE)
```

Next, we use the multivariate adaptive regression splines (MARS) algorithm introduced in Friedman (1991) to model ozone concentration as a nonlinear function of the eight meteorological variables plus day of the year; we allow for up to three-way interactions.

```
ozone.mars <- earth(ozone ~ ., data = ozone, degree = 3)
summary(ozone.mars)
```

The MARS model produced a generalized R^2 of 0.79, similar to what was reported in Breiman and Friedman (1985). A single three-way interaction was found involving the predictors

- wind: wind speed (mph) at Los Angeles International Airport (LAX)
- temp: temperature ($^{\circ}$ F) at Sandburg Air Force Base
- dpgr: the pressure gradient (mm Hg) from LAX to Dagget, CA

To understand this interaction, we can use a PDP. However, since the partial dependence between three continuous variables can be computationally expensive, we will run `partial` in parallel.

Setting up a parallel backend is rather straightforward. To demonstrate, the following snippet of code sets up the partial function to run in parallel on both Windows and Unix-like systems using the **doParallel** package.

```
library(doParallel) # load the parallel backend
cl <- makeCluster(4) # use 4 workers
registerDoParallel(cl) # register the parallel backend
```

Now, to run `partial` in parallel, all we have to do is invoke the `parallel = TRUE` and `paropts` options and the rest is taken care of by the internal call to **plyr** and the parallel backend we loaded². This is illustrated in the code chunk below which obtains the partial dependence of ozone on wind, temp, and dpgr in parallel. The last three lines of code simply add a label to the colorkey. The result is displayed in Figure 6. **Note:** it is considered good practice to shut down the workers by calling `stopCluster` when finished.

```
partial(ozone.mars, pred.var = c("wind", "temp", "dpgr"), plot = TRUE,
        chull = TRUE, parallel = TRUE, paropts = list(.packages = "earth")) # Figure 6
stopCluster(cl) # good practice
```

```
# Add a label to the colorkey
lattice::trellis.focus("legend", side = "right", clipp.off = TRUE, highlight = FALSE)
grid::grid.text("ozone", x = 0.2, y = 1.05, hjust = 0.5, vjust = 1)
lattice::trellis.unfocus()
```

It is important to note that when using more than two predictor variables, `plotPartial` produces a trellis display. The first two variables given to `pred.var` are used for the horizontal and vertical axes, and additional variables define the panels. If the panel variables are continuous, then shingles³ are produced first using the equal count algorithm (see, for example, `?lattice::equal.count`). Hence, it will be more effective to use categorical variables to define the panels in higher dimensional displays when possible.

Classification problems

For classification problems, partial dependence functions are on a scale similar to the logit; see, for example, Hastie et al. (2009, pp. 369–370). Suppose the response is categorical with K levels, then for

²Notice we have to pass the names of external packages that the tasks depend on via the `paropts` argument; in this case, "earth". See `?plyr::adply` for details.

³A shingle is a special Trellis data structure that consists of a numeric vector along with intervals that define the "levels" of the shingle. The intervals may be allowed to overlap.

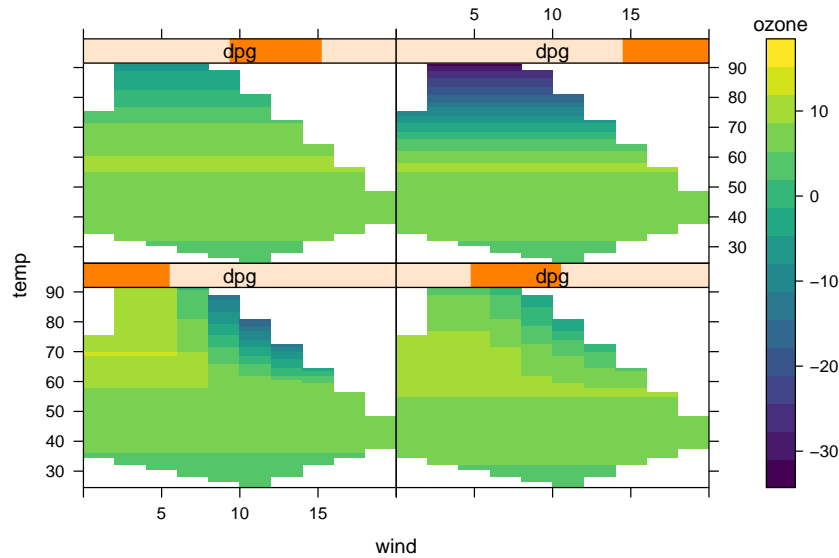


Figure 6: Partial dependence of ozone on wind, temp, and dpg. Since dpg is continuous, it is first converted to a shingle; in this case, four groups with 10% overlap.

each class we compute

$$f_k(x) = \log[p_k(x)] - \frac{1}{K} \sum_{k=1}^K \log[p_k(x)], \quad k = 1, 2, \dots, K, \quad (4)$$

where $p_k(x)$ is the predicted probability for the k -th class. Plotting $f_k(x)$ helps us understand how the log-odds for the k -th class depends on different subsets of the predictor variables.

To illustrate, we consider Edgar Anderson's iris data from the **datasets** package. The iris data frame contains the sepal length, sepal width, petal length, and petal width (in centimeters) for 50 flowers from each of three species of iris: setosa, versicolor, and virginica. We fit a support vector machine with a Gaussian radial basis function kernel to the data using the **svm** function in the **e1071** package (the tuning parameters were determined using 5-fold cross-validation).

```
library(e1071)
iris.svm <- svm(Species ~ ., data = iris, kernel = "radial", gamma = 0.75,
  cost = 0.25, probability = TRUE)
```

Note: the partial function has to be able to extract the predicted probabilities for each class, so it is necessary to set `probability = TRUE` in the call to `svm`.

Next, we plot the partial dependence of Species on both Petal.Width and Petal.Length for each of the three classes. The result is displayed in Figure 7.

```
pd <- NULL
for (i in 1:3) {
  tmp <- partial(iris.svm, pred.var = c("Petal.Width", "Petal.Length"),
    which.class = i, grid.resolution = 101, progress = "text")
  pd <- rbind(pd, cbind(tmp, Species = levels(iris$Species)[i]))
}

# Figure 7
library(ggplot2)
ggplot(pd, aes(x = Petal.Width, y = Petal.Length, z = yhat, fill = yhat)) +
  geom_tile() +
  geom_contour(color = "white", alpha = 0.5) +
  scale_fill_distiller(name = "Logit", palette = "Spectral") +
  theme_bw() +
  facet_grid(~ Species)
```

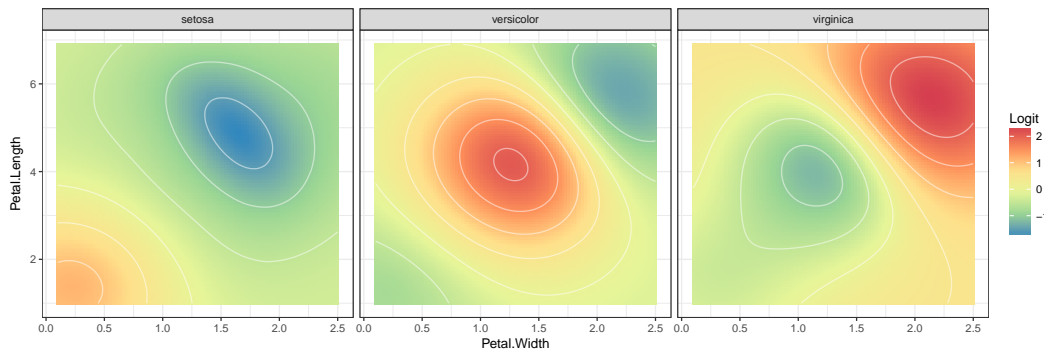


Figure 7: Partial dependence of species on petal width and petal length for the iris data.

User-defined prediction functions

PDPs are essentially just averaged predictions; see, for example, step 1. (c) in Algorithm 1. As pointed out by Goldstein et al. (2015), since traditional PDPs are based on the sample mean, strong interactions can conceal the complexity of the modeled relationship between the response and predictors of interest. This was part of the motivation behind Goldstein, Kapelner, Bleich, and Pitkin's ICE curves.

With `partial` it is possible to replace the mean in step 1. (c) with, say, the median, or obtain PDPs for classification problems on the raw probability scale. It is even possible to obtain ICE curves. This flexibility is due to the new `pred.fun` argument in `partial` (starting with `pdp` version 0.4.0). This argument accepts an optional prediction function that requires two arguments: `object` and `newdata`. The supplied prediction function must return either (i) a single prediction—typically an averaged prediction—or (ii) a vector of predictions. The two examples below illustrate.

Using the `pred.fun` argument it is possible to obtain PDPs for classification problems on the probability scale; we just need to write a function that extracts the predicted class probabilities of interest and then averages the result. The snippet of code below uses the fitted SVM from Section 2.2.5 to extract the average predicted probability of belonging to the Setosa class.

```
pred.prob <- function(object, newdata) { # see ?predict.svm
  pred <- predict(object, newdata, probability = TRUE)
  prob.setosa <- attr(pred, which = "probabilities")[, "setosa"]
  mean(prob.setosa)
}
```

Next, we simply pass this function via the `pred.fun` argument in the call to `partial`. The result is displayed in Figure 8.

```
# Figure 8
partial(iris.svm, pred.var = "Petal.Width", pred.fun = pred.prob, plot = TRUE,
       rug = TRUE)
```

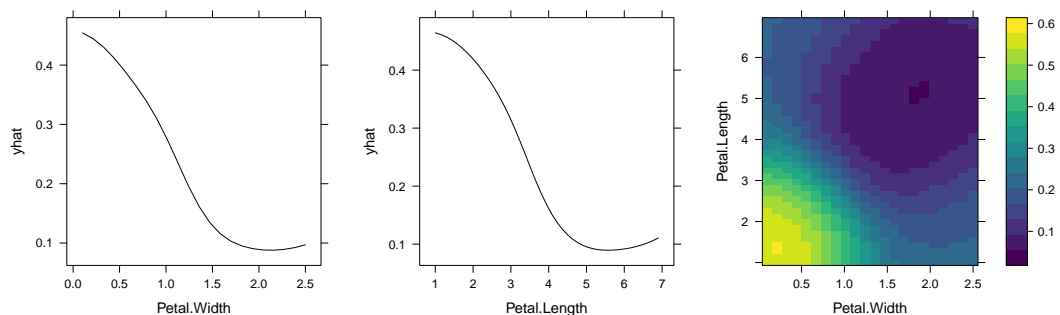


Figure 8: Partial dependence of Species on `Petal.Width` and `Petal.Length` plotted on the probability scale; in this case, the probability of belonging to the Setosa species.

For regression problems, the default prediction function is essentially

```
pred.fun <- function(object, newdata) {
  mean(predict(object, newdata), na.rm = TRUE)
}
```

This corresponds to step 1. (c) in Algorithm 1. Suppose we would like ICE curves instead. To accomplish this we simply need to pass a prediction function that returns a vector of predictions, one for each observation in newdata. This is demonstrated for the Boston housing example in the code snippet below.

```
pred.ice <- function(object, newdata) { # all predictions
  predict(object, newdata)
}
lstat.ice <- partial(boston.randomForest, pred.var = "lstat",
  pred.fun = pred.ice) # Figure 9 (left)
```

Recall that `partial` constructs a data frame that also inherits from class "partial". When the function supplied to `pred.fun` returns multiple predictions, `partial` instead constructs a data frame that also inherits from class "partial.ice". The only difference is the addition of a new column called "obs" which indicates which observation a curve belongs to; in this example, there is one curve for each unique observation in boston. To further illustrate, the code below constructs ICE curves for specific quantiles of the predictions. The results, along with the traditional PDP for `lstat`, are displayed in Figure 9.

```
pred.ice.quan <- function(object, newdata) { # quantiles of predictions
  quantile(predict(object, newdata), probs = 1:9/10)
}
lstat.ice.quan <- partial(boston.randomForest, pred.var = "lstat",
  pred.fun = pred.ice.quan) # Figure 9 (middle)
```

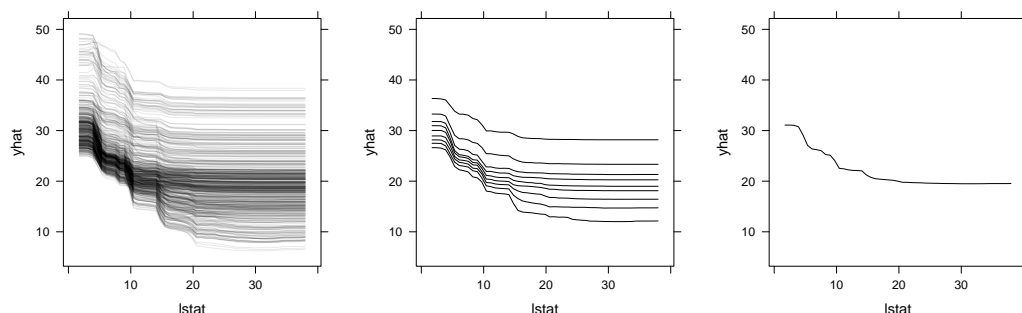


Figure 9: ICE curves and PDP depicting the relationship between `medv` and `lstat` for the Boston housing example. *Left:* ICE curves. *Middle:* Specific quantiles of ICE curves. *Right:* Standard PDP.

Using partial with the XGBoost library

To round out our discussion, we provide one last example using a recently popular (and successful!) machine learning tool. XGBoost, short for eXtreme Gradient Boosting, is a popular library providing optimized distributed gradient boosting that is specifically designed to be highly efficient, flexible and portable. The associated R package **xgboost** has been used to win a number of [Kaggle competitions](#). It has been shown to be many times faster than the well-known **gbm** package. However, unlike **gbm**, **xgboost** does not have built-in functions for constructing PDPs. Fortunately, the **pdp** package can be used to fill this gap.

For illustration, we return to the Boston housing data. The code chunk below uses **caret** to train an XGBoost model to the boston data frame from Section 2.2 over a grid of tuning values (see `?xgboost::xgboost` for an explanation of the various tuning parameters). The optimal combination of tuning parameter values is selected via 5-fold cross-validation.

```
library(caret)

# Setup for 5-fold cross-validation
ctrl <- trainControl(method = "cv", number = 5, verboseIter = TRUE)
```

```
# Grid of tuning values for XGBoost.
xgb.grid <- expand.grid(nrounds = c(1000, 2000, 3000),
                      max_depth = 1:4,
                      eta = c(0.001, 0.01, 0.1),
                      gamma = 0,
                      colsample_bytree = 1,
                      min_child_weight = 1,
                      subsample = c(0.5, 0.75, 1))

# Train an XGBoost model
set.seed(202)
boston.xgb <- train(x = data.matrix(subset(boston, select = -cmedv)),
                   y = boston$cmedv,
                   method = "xgbTree",
                   metric = "RMSE",
                   trControl = ctrl,
                   tuneGrid = xgb.grid)
```

The model with the smallest root-mean-squared error (RMSE) had `nrounds = 3000`, `max_depth = 3`, `eta = 0.01`, and `subsample = 0.75`; the final model achieved a cross-validated RMSE and R^2 of 2.789 and 0.906, respectively. The following snippet of code computes the partial dependence of `cmedv` on both `rm` and `lstat`, individually and together, and then uses `grid.arrange` to display all three PDPs on the same page. The results are displayed in Figure 10.

```
grid.arrange(
  # Figure 8 (left)
  partial(boston.xgb, pred.var = "rm", plot = TRUE, rug = TRUE),
  # Figure 8 (middle)
  partial(boston.xgb, pred.var = "lstat", plot = TRUE, rug = TRUE),
  # Figure 8 (right)
  partial(boston.xgb, pred.var = c("lstat", "rm"), plot = TRUE, chull = TRUE),
  ncol = 3
)
```

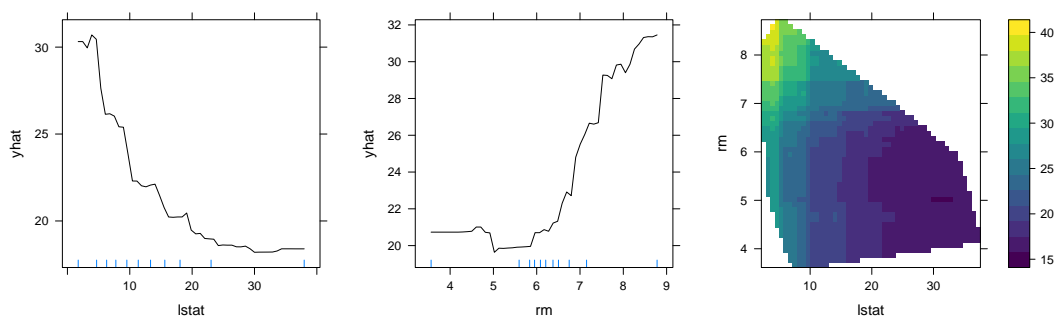


Figure 10: PDPs for the top two most important variables in the Boston housing data using XGBoost. Compare this to the random forest results displayed in Figures 4-5.

Since `train` defaults to storing a copy of the training data as part of the `"train"` object there is no need to supply it in the call to `partial`. However, this is not the case when using the **xgboost** package directly. For example, the following code fits the same model using the `xgboost` function.

```
set.seed(203) # for reproducibility
boston.xgb <- xgboost(data = data.matrix(subset(boston, select = -cmedv)),
                    label = boston$cmedv, objective = "reg:linear",
                    nrounds = 2000, max_depth = 3, eta = 0.01)
```

To use `partial` with the `"xgb.Booster"` class, we need to supply the original training data (minus the response) in the call to `partial`. The following snippet of code computes the partial dependence of `cmedv` on `rm` (plot not shown). (Make sure you are using version 0.6-0 or later of **xgboost**: <https://github.com/dmlc/xgboost/tree/master/R-package>.) **Note:** while `xgboost` requires the training data to be an object of class `"matrix"`, `"dgCMatrix"`, or `"xgb.DMatrix"`, `partial` requires a `"data.frame"`.

```
partial(boston.xgb, pred.var = "rm", plot = TRUE, rug = TRUE,
        train = subset(boston, select = -cmedv))
```

Summary

PDPs can be used to graphically examine the dependence of the response on low cardinality subsets of the features, accounting for the average effect of the other predictors. In this paper, we showed how to construct PDPs for various types of black box models in R using the **pdp** package. We also briefly discussed related approaches available in other R packages. Suggestions to avoid extrapolation and high execution times were discussed and demonstrated via examples.

In terms of future development, **pdp** can be expanded in a number of ways. For example, it would be useful to have the ability to construct PDPs for black box survival models—like conditional random forests with censored response. It would also be worthwhile to implement the partial dependence-based H -statistic (Friedman and Popescu, 2008) for assessing the strength of interaction between predictors.

Acknowledgments

TBD.

Bibliography

- E. Alfaro, M. Gámez, and N. García. *adabag: An R package for classification with boosting and bagging*. *Journal of Statistical Software*, 54(2):1–35, 2013. URL <http://www.jstatsoft.org/v54/i02/>. [p4]
- R. Analytics and S. Weston. *doMC: Foreach Parallel Adaptor for 'parallel'*, 2015a. URL <https://CRAN.R-project.org/package=doMC>. R package version 1.3.4. [p8]
- R. Analytics and S. Weston. *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*, 2015b. URL <https://CRAN.R-project.org/package=doParallel>. R package version 1.0.10. [p8]
- R. Analytics and S. Weston. *foreach: Provides Foreach Looping Construct for R*, 2015c. URL <https://CRAN.R-project.org/package=foreach>. R package version 1.4.3. [p5]
- B. Auguie. *gridExtra: Miscellaneous Functions for "Grid" Graphics*, 2016. URL <https://CRAN.R-project.org/package=gridExtra>. R package version 2.2.1. [p3]
- S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. URL <https://CRAN.R-project.org/package=magrittr>. R package version 1.5. [p4]
- L. Breiman and J. H. Friedman. Estimating optimal transformations for multiple regression and correlation. *Journal of the American Statistical Association*, 80(391):580–598, 1985. [p8]
- F. L. . E. Dimitriadou. *mlbench: Machine Learning Benchmark Problems*, 2010. URL <https://CRAN.R-project.org/package=mlbench>. R package version 2.1-1. [p3]
- J. Fox. Effect displays in R for generalised linear models. *Journal of Statistical Software*, 8(15):1–27, 2003. URL <http://www.jstatsoft.org/v08/i15/>. [p2]
- J. Fox and S. Weisberg. *An R Companion to Applied Regression*. Sage, Thousand Oaks CA, second edition, 2011. URL <http://socserv.socsci.mcmaster.ca/jfox/Books/Companion>. [p2]
- J. H. Friedman. Multivariate adaptive regression splines. *Annals of Statistics*, 19(1):1–67, 1991. [p8]
- J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2001. [p1]
- J. H. Friedman and B. E. Popescu. Predictive learning via rule ensembles. *Annals of Applied Statistics*, 2(3):916–954, 2008. [p13]
- J. H. Friedman and W. Stuetzle. Projection pursuit regression. *Journal of the American Statistical Association*, 76(376):817–823, 1981. [p3]

- S. Garnier. *viridis: Default Color Maps from 'matplotlib'*, 2016. URL <https://CRAN.R-project.org/package=viridis>. R package version 0.3.4. [p6]
- 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. [p2, 10]
- B. Greenwell. *pdp: An R Package for Constructing Partial Dependence Functions*, 2016. URL <https://CRAN.R-project.org/package=partial>. R package version 0.0.1. [p2]
- 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. [p1, 3]
- T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer New York, 2009. [p8]
- T. Hothorn and A. Zeileis. *partykit: A Laboratory for Recursive Partytioning*, 2016. URL <https://CRAN.R-project.org/package=partykit>. R package version 1.0-5. [p2]
- J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007. [p6]
- 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 <http://www.jstatsoft.org/v11/i09/>. [p4]
- M. Kuhn. *caret: Classification and Regression Training*, 2016. URL <https://CRAN.R-project.org/package=caret>. R package version 6.0-73. [p3]
- M. Kuhn, S. Weston, C. Keefer, and N. C. C. code for Cubist by Ross Quinlan. *Cubist: Rule- and Instance-Based Regression Modeling*, 2014. URL <https://CRAN.R-project.org/package=Cubist>. R package version 0.0.18. [p4]
- M. Kuhn, S. Weston, N. Coulter, and M. C. C. code for C5.0 by R. Quinlan. *C50: C5.0 Decision Trees and Rule-Based Models*, 2015. URL <https://CRAN.R-project.org/package=C50>. R package version 0.1.0-24. [p4]
- A. Liaw and M. Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002. URL <http://CRAN.R-project.org/doc/Rnews/>. [p2]
- D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, and F. Leisch. *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*, 2015. URL <https://CRAN.R-project.org/package=e1071>. R package version 1.6-7. [p4]
- S. Milborrow. *plotmo: Plot a Model's Response and Residuals*, 2015. URL <https://CRAN.R-project.org/package=plotmo>. R package version 3.1.4. [p2]
- S. Milborrow. *earth: Multivariate Adaptive Regression Splines*, 2016. URL <https://CRAN.R-project.org/package=earth>. R package version 4.4.4. [p4]
- A. Peters and T. Hothorn. *ipred: Improved Predictors*, 2015. URL <https://CRAN.R-project.org/package=ipred>. R package version 0.9-5. [p4]
- R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016. URL <https://www.R-project.org/>. [p3]
- D. Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer, New York, 2008. URL <http://lmdvr.r-forge.r-project.org>. ISBN 978-0-387-75968-5. [p3]
- D. Sarkar and F. Andrews. *latticeExtra: Extra Graphical Utilities Based on Lattice*, 2016. URL <https://CRAN.R-project.org/package=latticeExtra>. R package version 0.6-28. [p3]
- T. Therneau, B. Atkinson, and B. Ripley. *rpart: Recursive Partitioning and Regression Trees*, 2015. URL <https://CRAN.R-project.org/package=rpart>. R package version 4.1-10. [p4]
- C. S. Torsten Hothorn, Kurt Hornik and A. Zeileis. *party: A Laboratory for Recursive Partytioning*, 2015. URL <https://CRAN.R-project.org/package=party>. R package version 1.0-25. [p2]
- 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>. [p3]

H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009. ISBN 978-0-387-98140-6. URL <http://ggplot2.org>. [p3]

H. Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1): 1–29, 2011. URL <http://www.jstatsoft.org/v40/i01/>. [p7]

Brandon M. Greenwell
Infoscitex Corporation
4027 Colonel Glenn Highway
Suite 210
Dayton, OH 45431-1672
United States of America
greenwell.brandon@gmail.com