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 $\widehat{f}(x)$ 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 home values from $n = 506$ census tracts in the suburbs of Boston from the 1970 census; see Harrison and Rubinfeld (1978, Table IV) 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 there time and efforts was an interpretable prediction equation which is reproduced in Equation (1).

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

Nowadays, many 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 since these algorithms typically do not produce simple prediction formulas like Equation (1). These models can still provide insight into the data, but it is not in the form of simple equations. For example, quantifying predictor importance has become a crucial task in the analysis of "big data", and many supervised learning algorithms, like tree-based methods, can naturally assign variable importance scores to all of the predictors in the training data.

While determining predictor importance is a crucial task in any supervised learning problem, 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 $\widehat{f}(x)$. If we partition x into an interest set, z_s, and its compliment, $z_c = x \setminus z_s$, then the "partial dependence" of the response on z_s is defined as

$$f_s(z_s) = E_{z_c}\left[\widehat{f}(z_s, z_c)\right] = \int \widehat{f}(z_s, z_c)\, p_c(z_c)\, dz_c, \tag{2}$$

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

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

where $z_{i,c}$ $(i = 1, 2, \ldots, n)$ 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 $\{x_{11}, x_{12}, \ldots, x_{1k}\}$. The partial dependence of the response on x_1 can be constructed as follows:

```
1. For $i \in \{1, 2, \ldots, k\}$:
    (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 $i = 1, 2, \ldots, k$.
```

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, the algorithm can be parallelized quite easily (more on this in Section 2.2.4). It 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** (Ridgeway, 2015), among others; these are 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., generalized additive models 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 Milborrow, **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. As of **plotmo** version 3.3.0, there is now support for constructing PDPs, but it is not the default. The main difference is that **plotmo**, rather than applying step 1. (a)-(c) in Algorithm 1, accumulates all the data at once thereby reducing the number of internal calls to `predict`. The trade-off is a slight increase in speed at the expense of using more memory. So, why use the **pdp** package? As will be discussed in the upcoming sections, **pdp**:

- contains only a few functions with relatively few arguments;
- does <u>not</u> produce a plot by default;
- can be used more efficiently with `"gbm"` objects (see Section 2.2.4);
- produces graphics based on **lattice** (Sarkar, 2008), which are far more flexible than base R graphics;
- defaults to using false color level plots for multivariate displays (see Section 2.2.2);
- contains options to mitigate the risks associated with extrapolation (see Section 2.2.4);
- has the option to construct PDPs in parallel (see Section 2.2.4);
- is extremely flexible in the types of PDPs that can be produced (see Section 2.2.6),

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. ICE plots display the estimated relationship between the response and a predictor of interest for each observation. Consequently, the

PDP for a predictor of interest can be obtained by averaging the corresponding ICE curves across all observations. In Section 2.2.6, it is shown how to obtain ICE curves using the **pdp** package (which is slightly faster than **ICEbox**). **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 centered ICE (c-ICE) plots and derivative ICE (d-ICE) plots is also available in **ICEbox**; c-ICE plots help visualize heterogeneity in the modeled relationship between observations, and d-ICE plots help to explore interaction effects.

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. *Effect displays*, available in the **effects** package (Fox, 2003), provide tabular and graphical displays for the terms in parametric models while holding all other predictors at some constant value—similar in spirit to **plotmo**'s marginal model plots. 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`. The two most important functions exported by **pdp** are:

- `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 labeled `yhat`[1] and contains the values of the partial dependence function $\bar{f}_s\left(z_s\right)$. If `plot = TRUE`, then `partial` makes an internal call to `plotPartial` (with fewer plotting options) and returns the PDP in the form of a **lattice** plot (i.e., a `"trellis"` object). **Note:** it is recommended to call `partial` with `plot = FALSE` and store the results; this allows for more flexible plotting, and the user will not have to waste time calling `partial` again if the default plot is not sufficient.

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 (see Section 2.2.5 and Section 2.2.6 for examples).

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 **grid**-based graphical objects on

[1]There is one exception to this. When a function supplied via the `pred.fun` argument returns multiple predictions, the second to last and last columns will be labeled `yhat` and `yhat.id`, respectively (see Section 2.2.6).

a single plot (these include graphics produced using **lattice** (hence, **pdp**) and **ggplot2**). This is demonstrated in multiple examples throughout this paper.

Currently supported models are described in Table 1. In these cases, `partial` should be able to automatically determine an appropriate value for the argument `type` (i.e., `"regression"` or `"classification"`). In other situations, the user may need to specify this argument. This allows `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).

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"`
Generalized linear model	**stats**	`"glm"`, `"lm"`
Linear model	**stats**	`"lm"`
Nonlinear least squares	**stats**	`"nls"`
Multivariate adaptive regression splines (MARS)	**earth** (Milborrow, 2016)	`"earth"`
Projection pursuit regression	**stats**	`"ppr"`
Random forest	**randomForest**	`"randomForest"`
	party	`"RandomForest"`
	partykit	`"cforest"`
	ranger (Wright, 2016)	`"ranger"`
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`.

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.

Another important argument to `partial` is `train`. If `train = NULL` (the default), `partial` tries to extract the original training data from the fitted model object. For objects that typically store a copy of the training data (e.g., objects of class `"BinaryTree"`, `"RandomForest"`, and `"train"`), this is straightforward. Otherwise, `partial` will attempt to extract the call stored in `object` (if available) and use that to evaluate the training data in the same environment from which `partial` was called. This can cause problems when, for example, the training data have been changed after fitting the model, but before calling `partial`. Hence, it is good practice to always supply the training data via the `train` argument in the call to `partial`[2]. If `train = NULL` and the training data can not be extracted from the fitted model, the user will be prompted with an informative error message (this will occur, for example, when using `partial` with `"ksvm"` and `"xgb.Booster"` objects):

```
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`.
```

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

[2]For brevity, we ignore this option in most of the examples in this paper.

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)]
```

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

```
library(randomForest)  # for randomForest, partialPlot, and varImpPlot functions
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.

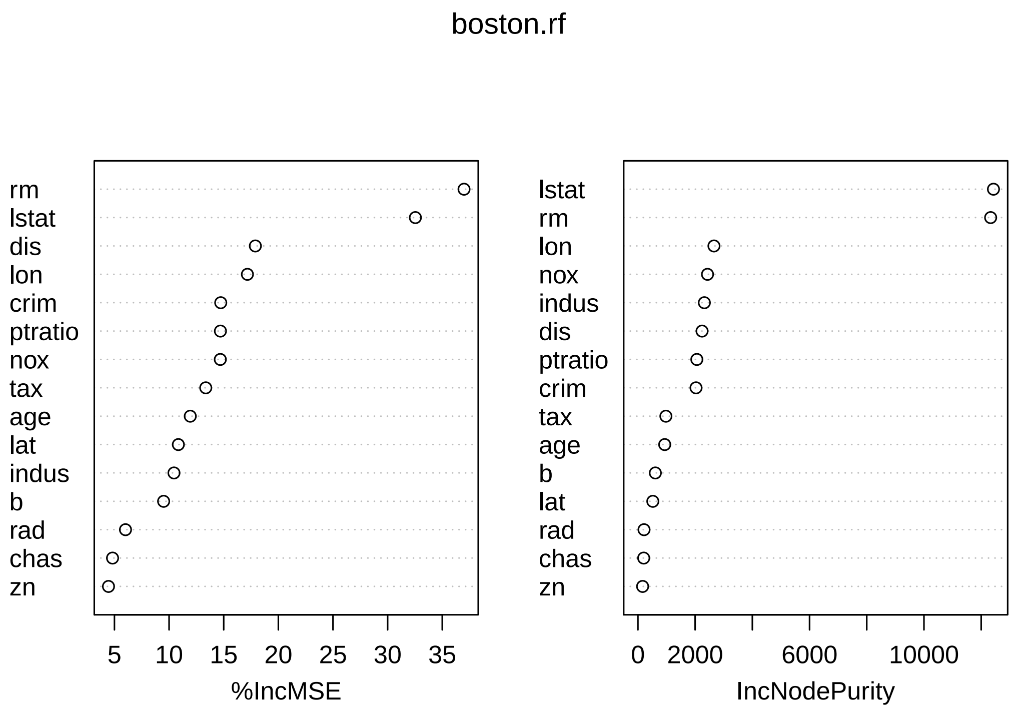

Figure 1: Dotchart of variable importance scores for the Boston housing data based on a random forest with 500 trees.

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)  # for partial, plotPartial, and grid.arrange functions
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"
  partial(pred.var = "lstat") %>%
  plotPartial(smooth = TRUE, lwd = 2, ylab = expression(f(lstat)))
```

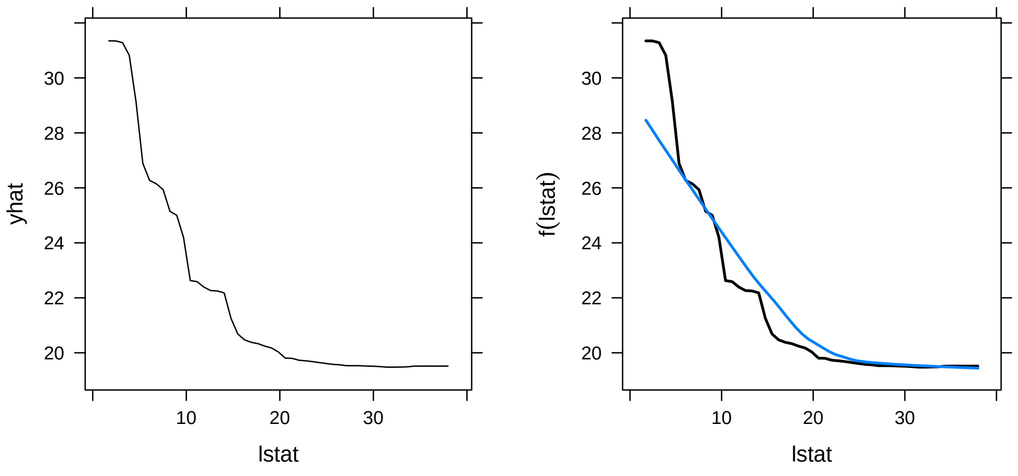

Figure 2: Partial dependence of `cmedv` on `lstat` based on a random forest. *Left*: Default plot. *Right*: Customized plot obtained using the `plotPartial` function.

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 used to display three PDPs, which make use of various `plotPartial` options[3], on the same graph. The results are displayed in Figure 3.

```
# Compute partial dependence data for lstat and rm
pd <- partial(boston.rf, pred.var = c("lstat", "rm"))

# Default PDP
pdp1 <- plotPartial(pd)

# Add contour lines and use a different color palette
rwb <- colorRampPalette(c("red", "white", "blue"))
pdp2 <- plotPartial(pd, contour = TRUE, col.regions = rwb)

# 3-D surface
pdp3 <- plotPartial(pd, levelplot = FALSE, zlab = "cmedv", drape = TRUE,
                    colorkey = TRUE, screen = list(z = -20, x = -60))

# Figure 3
grid.arrange(pdp1, pdp2, pdp3, 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).

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

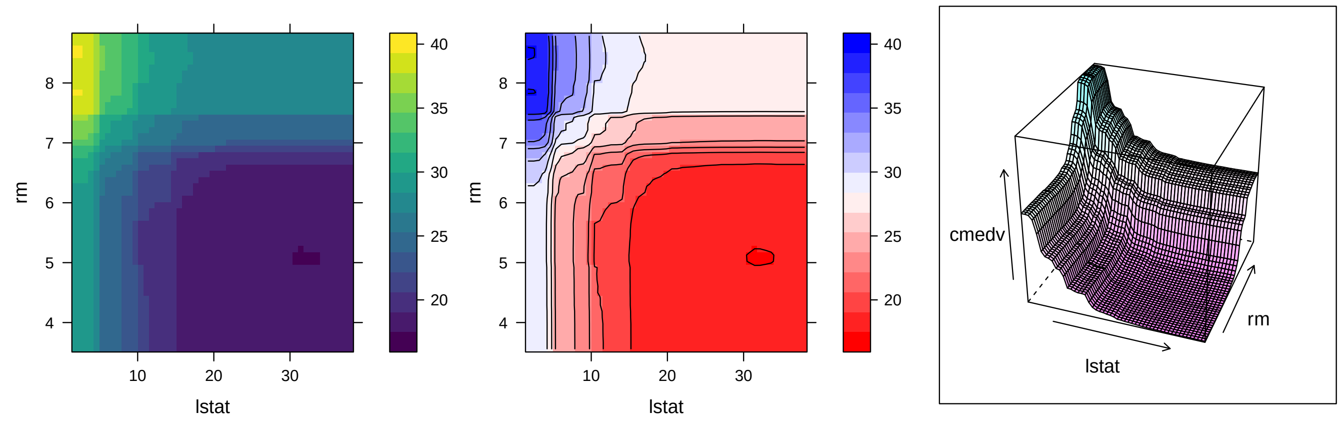

Figure 3: Partial dependence of `cmedv` on `lstat` and `rm` based on a random forest. *Left*: Default plot. *Middle*: With contour lines and a different color palette. *Right*: Using a 3-D surface.

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)
```

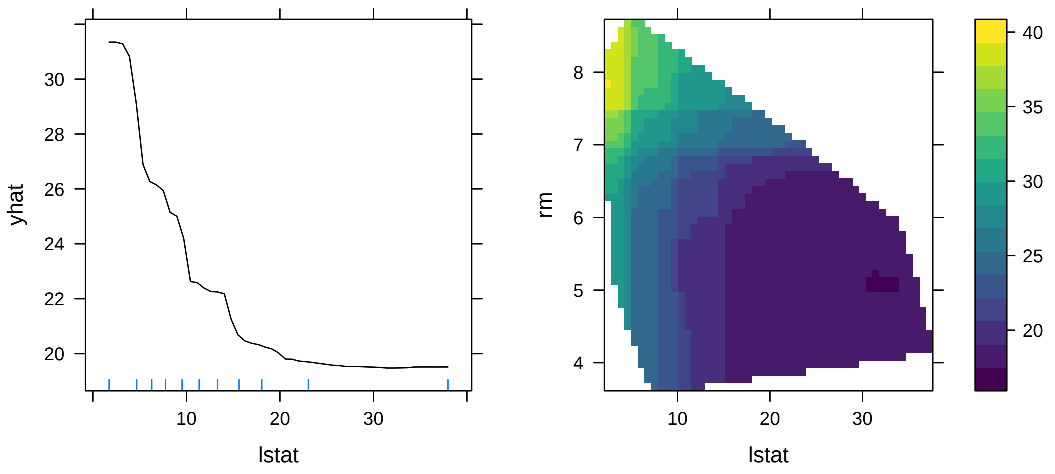

Figure 4: Examples of PDPs with the addition of a rug display (left) and a convex hull (right).

Addressing computational concerns

Constructing PDPs can be quite computationally expensive[4] Several strategies are available to ease the computational burden in larger problems. For example, there is no need to compute partial dependence of `cmedv` 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 `cmedv` on `rm` using each option; `grid.arrange` is used to display all three PDPs on the same graph, side by side. The results are displayed in Figure 5.

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

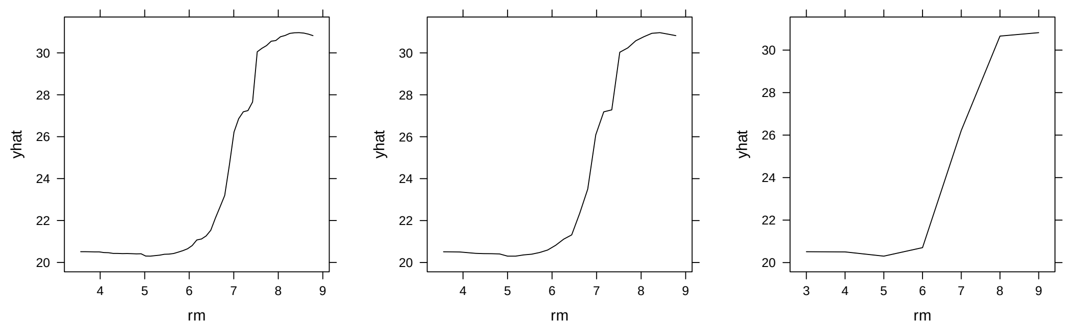

Figure 5: Partial dependence of `cmedv` on `rm`. *Left*: Default plot. *Middle*: Using a reduced grid size. *Right*: Using a user-specified grid.

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.

```
library(earth)  # for earth function (i.e., MARS algorithm)
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

[4]The exception is regression trees based on single-variable splits which can make use of the efficient weighted tree traversal method described in Friedman (2001), however, only the **gbm** package seems to make use of this approach; consequently, **pdp** can also exploit this strategy when used with **gbm** models (see `?partial` for details).

- `wind`: wind speed (mph) at Los Angeles International Airport (LAX)
- `temp`: temperature (^oF) at Sandburg Air Force Base
- `dpg`: 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 loaded[5]. This is illustrated in the code chunk below which obtains the partial dependence of `ozone` on `wind`, `temp`, and `dpg` in parallel. The last three lines of code 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", "dpg"), 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()
```

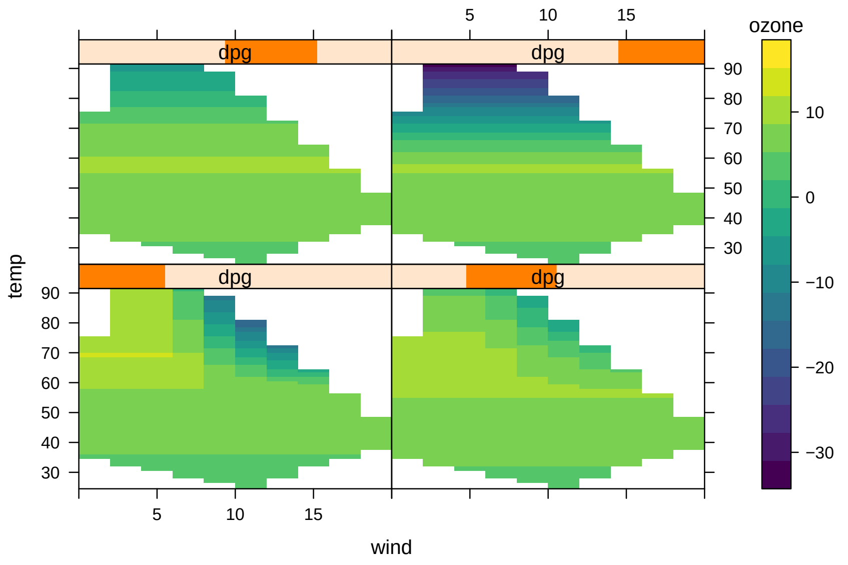

Figure 6: Partial dependence of `ozone` on `wind`, `temp`, and `dpg`. Since `dpg` is continuous, it is first converted to a shingle; in this case, four groups with 10% overlap.

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 shingles[6] 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.

[5]Notice 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.

[6]A 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.

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 each class we compute

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

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)  # for svm function
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 = "Centered\nlogit", palette = "Spectral") +
  theme_bw() +
  facet_grid(~ Species)
```

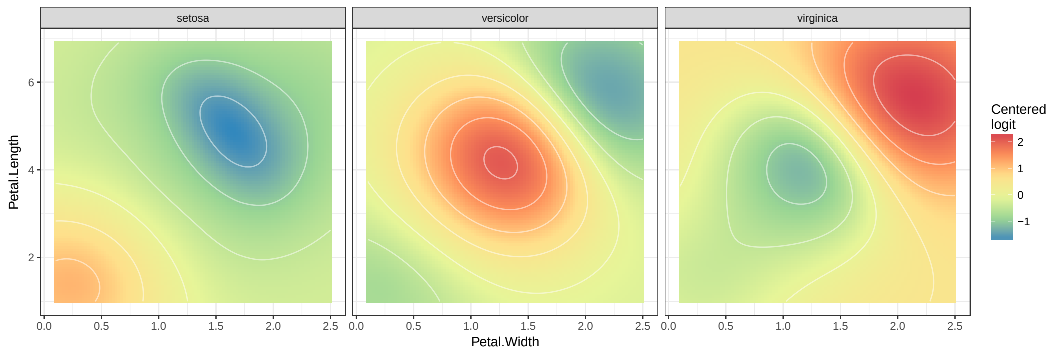

Figure 7: Partial dependence of `Species` on `Petal.Width` and `Petal.Length` for the iris data.

User-defined prediction functions

PDPs are essentially just averaged predictions; see, for example, step 1. (c) in Algorithm 1. Consequently, as pointed out by Goldstein et al. (2015), 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 plot procedure.

With `partial` it is possible to replace the mean in step 1. (c) of Algorithm 1 with any other function (e.g., the median or trimmed mean), or obtain PDPs for classification problems on the 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 a single prediction or a vector of predictions. Returning the mean of all the predictions will result in the traditional PDP. Returning a vector of predictions (i.e., one for each observation) will result in a set of ICE curves. The 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 computes the predicted class probability of interest averaged across all observations. The function below can be used with the fitted SVM from the iris example of 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 following chunk of code construct PDPs for `Petal.Width` and `Petal.Length` on the probability scale. The results are displayed in Figure 8.

```
# PDPs for Petal.Width and Petal.Length on the probability scale
pdp.pw <- partial(iris.svm, pred.var = "Petal.Width", pred.fun = pred.prob,
                  plot = TRUE)
pdp.pl <- partial(iris.svm, pred.var = "Petal.Length", pred.fun = pred.prob,
                  plot = TRUE)
pdp.pw.pl <- partial(iris.svm, pred.var = c("Petal.Width", "Petal.Length"),
                     pred.fun = pred.prob, plot = TRUE)

# Figure 8
grid.arrange(pdp.pw, pdp.pl, pdp.pw.pl, ncol = 3)
```

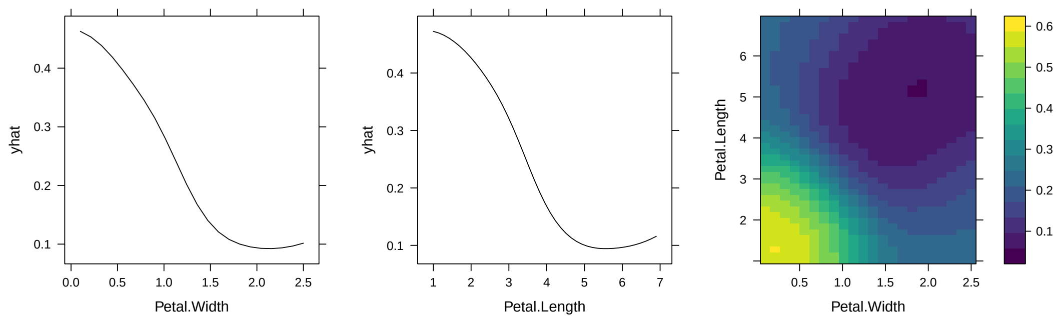

Figure 8: Partial dependence of `Species` on `Petal.Width` and `Petal.Length` plotted on the probability scale; in this case, the probability of belonging to the setosa species.

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 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`. The code snippet below illustrates this for the Boston housing example using the predictor `rm`. The result is displayed in Figure 9. **Note:** when the function supplied to `pred.fun` returns multiple predictions, the data frame returned by `partial` includes an additional column, `yhat.id`, that indicates which curve a point belongs to; in the following code chunk, there will be one curve for each observation in `boston`.

```
# Use partial to obtain ICE curves
pred.ice <- function(object, newdata) predict(object, newdata)
```

```
age.ice <- partial(boston.rf, pred.var = "age", pred.fun = pred.ice)

# Figure 9
plotPartial(age.ice, rug = TRUE, train = boston, alpha = 0.3)
```

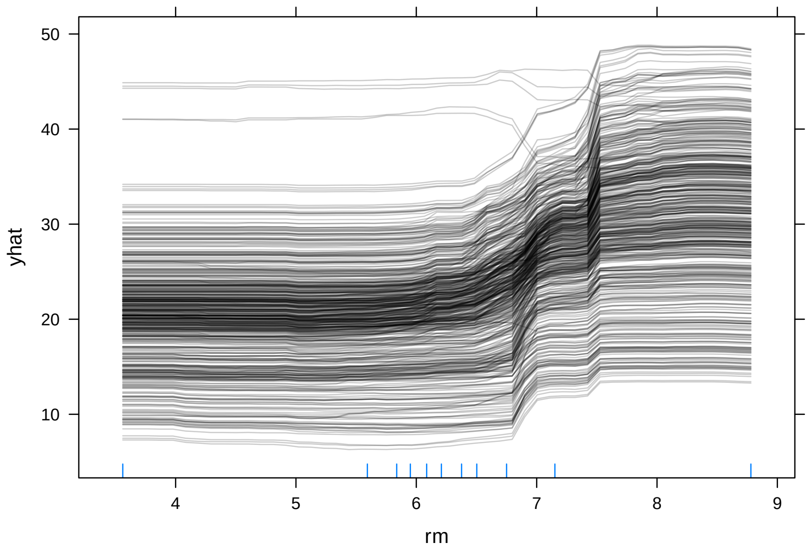

Figure 9: ICE curves depicting the relationship between cmedv and rm for the Boston housing example. Each curve corresponds to a different observation.

The curves in Figure 9 indicate some heterogeneity in the fitted model (i.e., some of the curves depict the opposite relationship). Such heterogeneity can be easier to spot using c-ICE curves; see Equation (4) on page 49 of Goldstein et al. (2015). Using **dplyr** (Wickham and Francois, 2016), it is rather straightforward to post-process the output from partial to obtain c-ICE curves (similar to the construction of *raw change scores* (Fitzmaurice et al., 2011, pg. 130) for longitudinal data). This is shown below.

```
# Post-process rm.ice to obtain c-ICE curves
library(dplyr)  # for group_by and mutate functions
rm.ice <- rm.ice %>%
  group_by(yhat.id) %>%
  mutate(yhat.centered = yhat - first(yhat))
```

Since the PDP is just the average of the corresponding ICE curves, it is quite simple to display both on the same plot. This can be accomplished by using the stat_summary function from the **ggplot2** package to average the ICE curves together. The code snippet below plots the ICE curves and c-ICE curves, along with their averages, for the predictor rm in the Boston housing example. The results are displayed in Figure 10.

```
# ICE curves with their average
p1 <- ggplot(rm.ice, aes(rm, yhat)) +
  geom_line(aes(group = yhat.id), alpha = 0.2) +
  stat_summary(fun.y = mean, geom = "line", col = "red", size = 1)

# c-ICE curves with their average
p2 <- ggplot(rm.ice, aes(rm, yhat.centered)) +
  geom_line(aes(group = yhat.id), alpha = 0.2) +
  stat_summary(fun.y = mean, geom = "line", col = "red", size = 1)

# Figure 10
grid.arrange(p1, p2, ncol = 2)
```

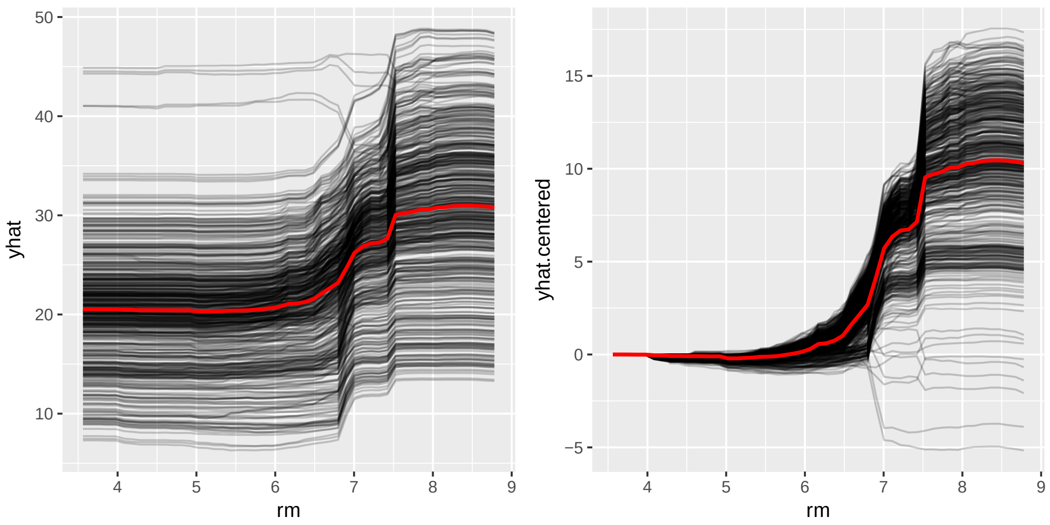

Figure 10: ICE curves (black curves) and their average (red curve) depicting the relationship between `cmedv` and `rm` for the Boston housing example. *Left*: Uncentered (here the red curve is just the traditional PDP). *Right*: Centered.

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 example. The code chunk below uses **caret** to tune an **xgboost** model using 10-fold cross-validation. (After loading **caret**, use `getModelInfo("xgbTree")` for information on tuning **xgboost** models.) **Warning:** The following code chunk may take a few minutes to run.

```
# Tune an XGBoost model using 10-fold cross-validation
library(caret)  # functions related to classification and regression training
set.seed(202)  # for reproducibility
boston.xgb <- train(x = data.matrix(subset(boston, select = -cmedv)),
                    y = boston$cmedv, method = "xgbTree", metric = "Rsquared",
                    trControl = trainControl(method = "cv", number = 10),
                    tuneLength = 10)
```

The optimal model had a cross-validated R^2 of 0.902 (use `print(boston.xgb$bestTune)` to view the optimum tuning parameters). The next snippet of code computes the partial dependence of `cmedv` on both `rm` and `lstat`, individually and together. The results are displayed in Figure 11.

```
# PDPs for lstat and rm
pdp.lstat <- partial(boston.xgb, pred.var = "lstat", plot = TRUE, rug = TRUE)
pdp.rm <- partial(boston.xgb, pred.var = "rm", plot = TRUE, rug = TRUE)
pdp.lstat.rm <- partial(boston.xgb, pred.var = c("lstat", "rm"),
                        plot = TRUE, chull = TRUE)

# Figure 11
grid.arrange(pdp.lstat, pdp.rm, pdp.lstat.rm, ncol = 3)
```

The `train` function creates objects of class `"train"`, whereas the `xgboost` function creates objects of class `"xgb.Booster"`. 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` in this example. However, this is not the case when using the **xgboost** package directly. To illustrate, we fit the same model using the `xgboost` function with the tuning parameters found previously using **caret**.

```
library(xgboost)  # for xgboost function
set.seed(203)  # for reproducibility
```

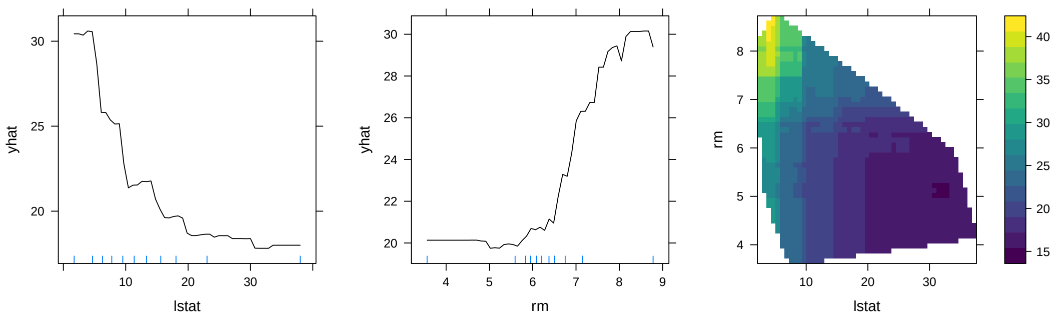

Figure 11: PDPs for the top two most important variables in the Boston housing data using **xgboost**. Compare this to the random forest results displayed in Figures 4-5.

```
boston.xgb <- xgboost(data = data.matrix(subset(boston, select = -cmedv)),
                      label = boston$cmedv, objective = "reg:linear",
                      nrounds = 100, max_depth = 5, eta = 0.3, gamma = 0,
                      colsample_bytree = 0.8, min_child_weight = 1,
                      subsample = 0.9444444)
```

To use `partial` with `"xgb.Booster"` objects, 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"` that does not contain the response column.

```
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. [p6]

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. [p6]

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. [p4]

G. Fitzmaurice, N. Laird, and J. Ware. *Applied Longitudinal Analysis*. Wiley Series in Probability and Statistics. Wiley, 2011. [p12]

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/. [p3]

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. [p3]

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, 8]

J. H. Friedman and B. E. Popescu. Predictive learning via rule ensembles. *Annals of Applied Statistics*, 2 (3):916–954, 2008. [p14]

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, 12]

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, 4]

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. [p10]

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. [p4]

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]

G. Ridgeway. *gbm: Generalized Boosted Regression Models*, 2015. URL https://CRAN.R-project.org/package=gbm. R package version 2.1.1. [p2]

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. [p2]

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. [p4]

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/. [p8]

H. Wickham and R. Francois. *dplyr: A Grammar of Data Manipulation*, 2016. URL https://CRAN.R-project.org/package=dplyr. R package version 0.5.0. [p12]

M. N. Wright. *ranger: A Fast Implementation of Random Forests*, 2016. URL https://CRAN.R-project.org/package=ranger. R package version 0.6.0. [p4]

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