Brightbox: a short introduction

Benjamin Rollert and Ronny Li

2017-06-28

Brightbox contains functions that tackle the problem of inspecting internals for any blackbox supervised learner. The package is designed to work with the caret package as well as any model that is an ensemble of `caret` learners. As of writing, the approaches implemented in the package are **partial dependency plots** (`run_partial_dependency`) and **marginal variable importance** (`calculate_marginal_vimp`).

Installation

```
> devtools::install_github('breather/brightbox')
```

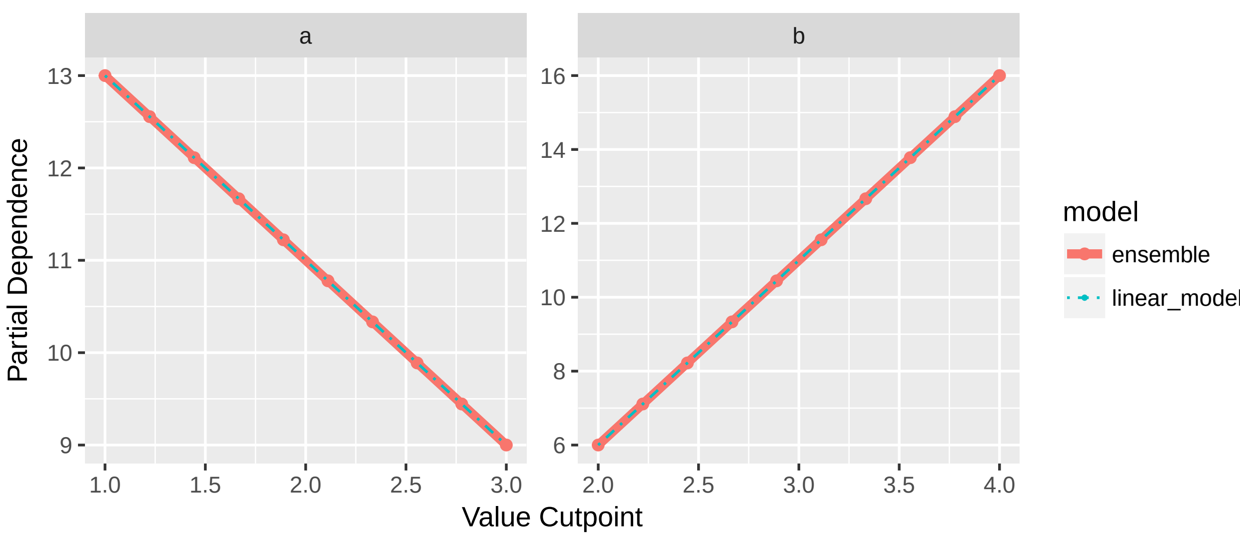
Partial Dependency Plots

Introduction

Partial dependency plots are a technique for visualizing the effect of a single feature on the response, marginalizing over the values of all other features. They are the visual equivalent of a coefficient in linear regression (and in fact, the partial dependency plot for a coefficient in a linear model will be a straight line).

```
# Example of partial dependency plots for a linear model
library(data.table)
dt <- data.table(a = 1:3, b = 2:4, c = c(8, 11, 14))
lm1 <- lm(c ~ a + b - 1, dt)
lm1$coefficients
#>  a  b
#> -2  5
```

```
# Note that the slopes of the plotted lines match the coefficients
library(brightbox)
pd <- run_partial_dependency(feature_dt = dt[, c("a", "b")],
                             model_list = list(linear_model = lm1))
```



The advantage of partial dependency plots shine when there is a non-linear relationship between the feature in question and the response. In such cases, a linear model will hide the true relationship due to its underlying assumptions. Ideally we would like to use a more flexible model with fewer assumptions but maintain interpretability.

The next section will work through an example dataset to demonstrate `brightbox`'s features with respect to partial dependency plots.

Walkthrough

```r
# First, load the data
library(data.table)
library(mlbench)
data(BostonHousing, package = "mlbench")
head(BostonHousing)
#>      crim zn indus chas   nox    rm  age    dis rad tax ptratio      b
#> 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90
#> 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90
#> 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83
#> 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63
#> 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90
#> 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12
#>   lstat medv
#> 1  4.98 24.0
#> 2  9.14 21.6
#> 3  4.03 34.7
#> 4  2.94 33.4
#> 5  5.33 36.2
#> 6  5.21 28.7
```

```r
# Split into features and response
boston_dt <- data.table(BostonHousing)
x <- boston_dt[ , -"medv", with = FALSE]
y <- boston_dt$medv
# Prep the data (numeric columns are friendlier)
x[, chas := as.numeric(chas)]
```

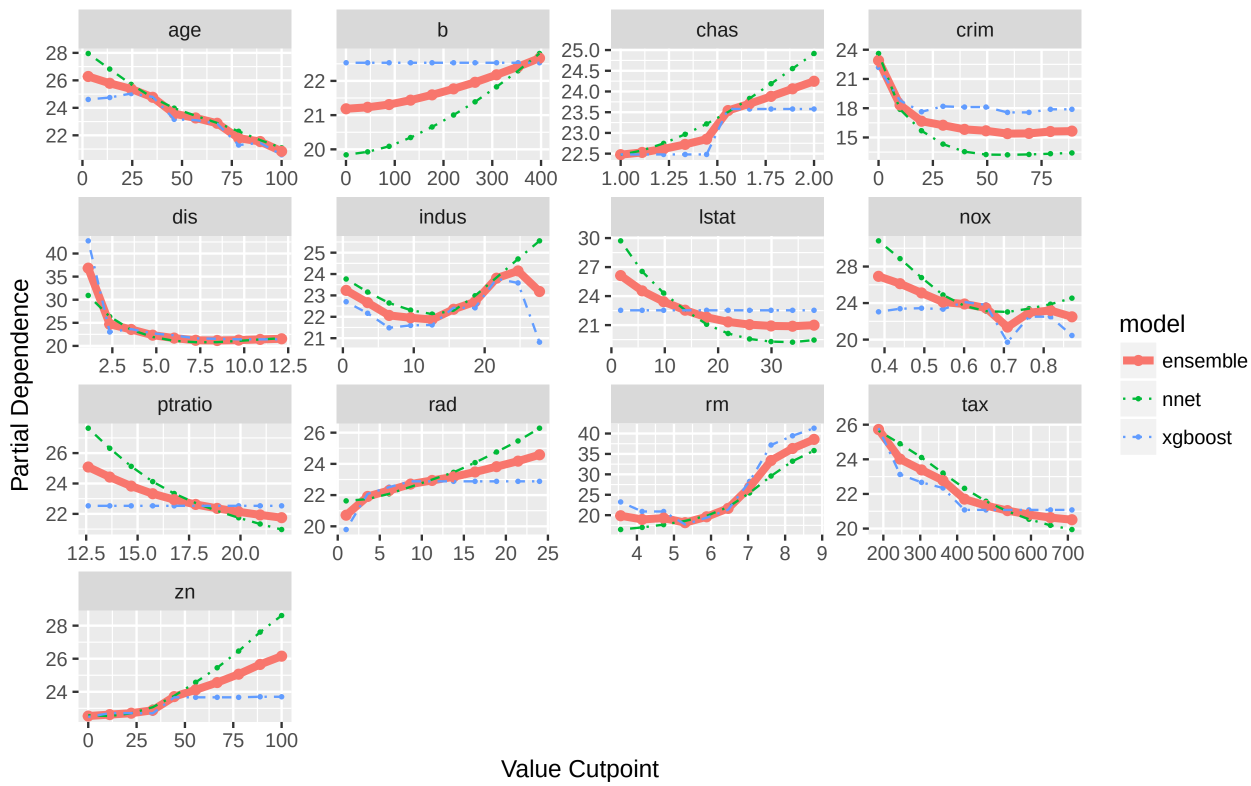
The dataset `BostonHousing` contains housing data for 506 census tracts of Boston from the 1970 census. `medv` is the response variable representing the median value of houses in each census tract (in USD 1000's). See `??BostonHousing` for additional details.

We will train two blackbox learners on this data and see if we can interpret the patterns each model learned.

```r
library(caret)
# Train an xgboost model
xgb <- train(x = x, y = y,
             method = "xgbTree", metric = "RMSE")
# Train a single-layer neural net
nn <- train(x = x, y = y,
            method = "nnet", metric = "RMSE",
            preProc = c("center", "scale"),
            tuneGrid = expand.grid(size = c(10, 15, 20), decay = c(0, 5e-4, 0.05)),
            linout = TRUE,
            trace = FALSE)
```

From the two trained models we can use Brightbox to inspect their internals and trivially, the internals of an ensemble of these models. In this example, we construct an ensemble that is the median of the xgboost and neural net models.

```r
library(brightbox)
# Generate partial dependency plots for each feature
pd <- run_partial_dependency(x, model_list = list(xgboost = xgb, nnet = nn),
                             ensemble_colname = "ensemble", ensemble_fcn = median)
```



In the returned plot we can see how `medv` changes with respect to each feature for every model. As a result, there are quite a few things we can learn by inspecting the plot.

1. We can easily see which features have a positive or negative relationship with the response and that the patterns are more complex than in the earlier example with linear regression (`rm` and `nox` for example).
2. There are some features and some ranges within features where the two models are roughly in agreement. From this we can conclude that the plotted signal in those ranges is rather strong. (`age`, `rm`, `dis`)
3. Some features have a wider range in the y-axis than others. This means that the feature had a larger impact on the response. In general, if a partial dependency plot is completely flat (zero slope) then the feature does not affect the model's predictions at all (an exception being when interaction effects are present).

As was hinted in points 2 and 3 above, partial dependency plots can help us determine feature importance by inspecting the partial dependence range and variance. `run_partial_dependency` returns a data.table with such calculations pre-computed.

```r
# pd was previously saved from run_partial_dependency
head(pd)
#>    feature feature_val   model prediction     vimp
#> 1:      rm    3.561000 xgboost   23.23486 20.39906
#> 2:      rm    4.140889 xgboost   20.89914 20.39906
#> 3:      rm    4.720778 xgboost   20.89914 20.39906
#> 4:      rm    5.300667 xgboost   17.73334 20.39906
```

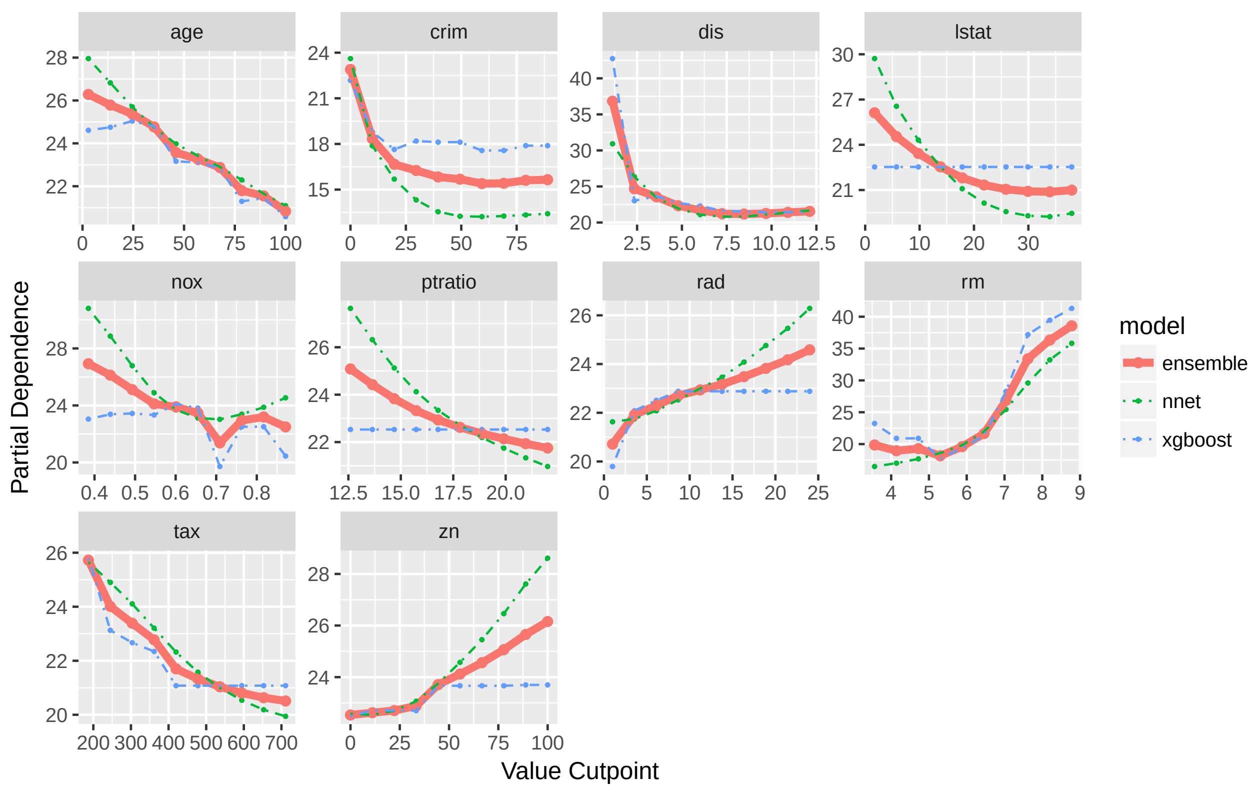
```
#> 5:       rm    5.880556 xgboost   19.38172 20.39906
#> 6:       rm    6.460444 xgboost   21.32710 20.39906
```

pd contains the data necessary to reconstruct the partial dependency plots (columns feature, feature_val, model, and prediction) as well as a variable importance column (vimp). Variable importance is calculated as the y-axis range for a given feature and model, with the ensemble model chosen by default (TODO: incorporate variance into the variable importance calculation).

```
# Inspect the variable importance of each feature
unique(pd[, list(feature, vimp)])
#>     feature      vimp
#>  1:      rm 20.399065
#>  2:     dis 15.656962
#>  3:    crim  7.502317
#>  4:     nox  5.554842
#>  5:     age  5.446153
#>  6:   lstat  5.239971
#>  7:     tax  5.212727
#>  8:     rad  3.871130
#>  9:      zn  3.622065
#> 10: ptratio  3.333818
#> 11:   indus  2.272505
#> 12:    chas  1.770995
#> 13:       b  1.480411
```

From here we may be interested in plotting just the 10 most important features.

```
pd_top <- run_partial_dependency(x, model_list = list(xgboost = xgb, nnet = nn),
                                 feature_cols = unique(pd[["feature"]])[1:10],
                                 ensemble_colname = "ensemble", ensemble_fcn = median)
```



Inspecting the variance of a partial dependence plot

We may wish to check the stability of the relationship between a feature and the partial dependence. How do we know that the relationship isn't just by chance, an artifact of a model overfitting?

We propose an approach in which many models are trained on different subsamples of the training data. Both visual tests and automated tests give us good insight as to the variance in the partial dependence relationship.

Let's start by training 50 xgboost models on random subsamples of the `BostonHousing` dataset and store the resulting models in a list:

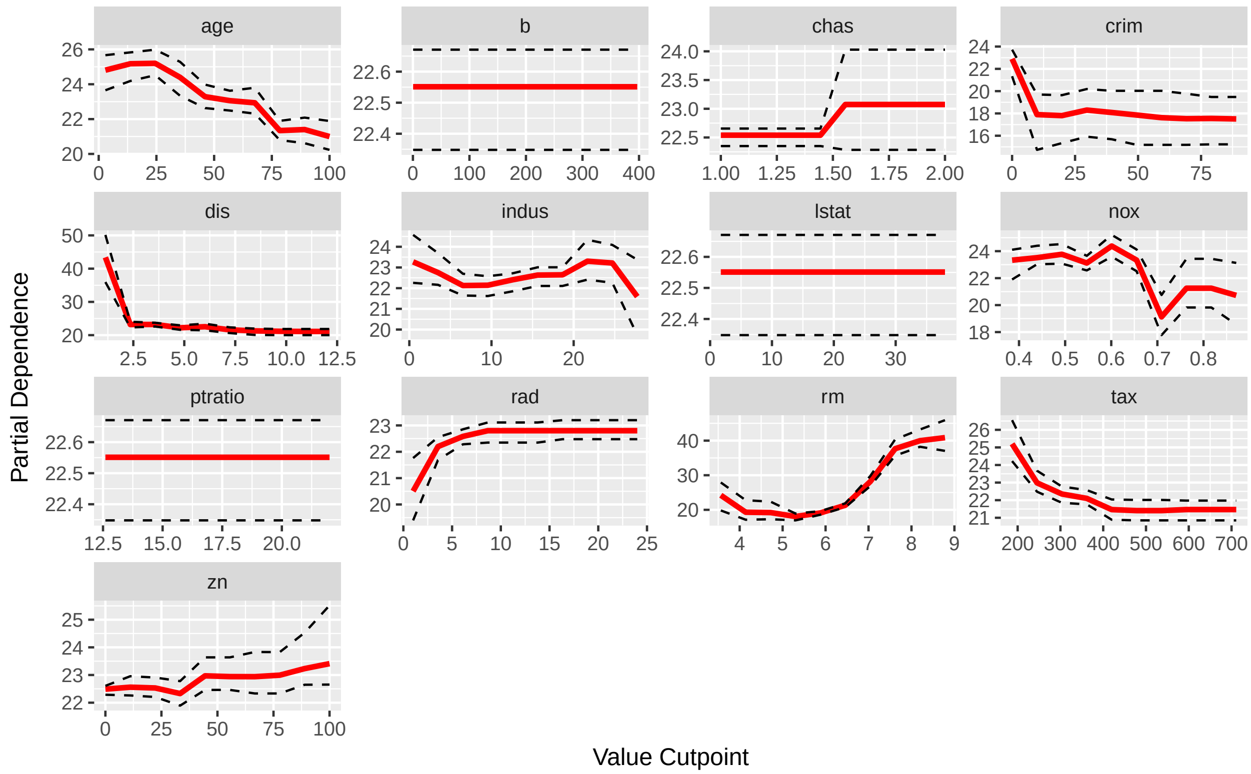
```
library(caret)
# Train 50 xgboost models
xgb_list <- list()
for (i in 1:50) {
  ind <- sample(x = 1:nrow(x), size = 404)  # randomly sample ~80% of rows
  xgb_list[[i]] <- train(x = x[ind], y = y[ind],
                         method = "xgbTree", metric = "RMSE",
                         trControl = trainControl(method = "none"),
                         tuneGrid = xgb$bestTune)
}
names(xgb_list) <- paste("xgb", 1:50, sep = "_")
```

Now that we have our 50 seperate models stored in list, we can try plotting the resulting output. Instead of plotting all 50 models in the same plot, which would turn into a mess, we plot specific quantiles of the model predictions, namely the 5th, 50th, and 95th quantiles.

```
# return list of partial dependency data.tables, one data.table for each feature
pd_list <- loop_calculate_partial_dependency(feature_dt = x, model_list = xgb_list)
pd <- do.call(rbind, pd_list)  # rbind data.tables into one long data.table

pd_quant <- pd[model != "ensemble",
               list(lcl = quantile(prediction, .05),
                    ensemble = median(prediction),
                    ucl = quantile(prediction, .95)),
                    by = c("feature", "feature_val")]

ggplot(data = pd_quant,
       aes(x = feature_val,
           y = prediction)) +
           geom_line(aes(y = ensemble), size = 1.2, color = "red") +
           geom_line(aes(y = ucl), size = 0.5, linetype = "dashed") +
           geom_line(aes(y = lcl), size = 0.5, linetype = "dashed") +
           facet_wrap(~feature, scales = "free") +
           labs(x = "Value Cutpoint", y = "Partial Dependence")
```



We can see that for variables like `dis`, `tax`, and `rm`, there is little spread between the quantiles, indicating very stable results regardless of the subsample the models were trained on. Conversely, we can conclude that variables like `crim` and `chas` have a high variance between models, at least for certain ranges of the variables; we should treat any inference about these variables with caution.

While the above is a useful visual test, we can also get an automated ranking of variable importance normalized by model variance. The function `calculate_pd_vimp_normed` performs the following operations:

1. Computes the same score as `calculate_pd_vimp`
2. Computes the standard deviation of the model predictions at each value cutpoint, returning a vector of standard devation scores
3. Returns a summary value (i.e. a vector of length 1) of the standard deviation vector returned in step 2. The default summary function is to take the `median` of the standard deviation vector.
4. Divides the score computed by `calculate_pd_vimp` in step 1 by the summary value of the standard deviation in step 3.

We simply pass `pd_list` to the function `calculate_pd_vimp_normed` using `lapply` to get the variance adjusted ranking of all variables in the dataset:

```
# apply calculate_pd_vimp_normed to each element of pd_list
vimp_normed_vec <- sapply(pd_list, calculate_pd_vimp_normed)
#Print descending order of variable importance
print(vimp_normed_vec[order(-vimp_normed_vec)])
#>       dis        rm       tax       age       rad       nox      crim
#> 37.501116 14.827416  9.584020  9.460136  9.298281  8.441780  3.839901
#>     indus        zn      chas   ptratio         b     lstat
#>  3.668547  2.780694  1.583559  0.000000  0.000000  0.000000
```

As we can see, the ranking is consistent with what we can visually identify as being the most stable relationships.

Partial dependency functions

While the function `run_partial_dependency` is the primary interface for partial dependency plots, it is composed of the following functions which can be useful if you want to avoid redundant calculations.

- `calculate_partial_dependency`
- `loop_calculate_partial_dependency`
- `facet_plot_fcn`
- `loop_plot_fcn`
- `plot_partial_dependency`
- `calculate_pd_vimp`

Marginal Variable Importance

(TODO: change this to a walkthrough instead)

```
calculate_marginal_vimp(x, y, method, loss_metric, resampling_indices, tuneGrid, trControl,
                        vars = names(x), allow_parallel = FALSE, ...)
```

Arguments `x`: data.table containing predictor variables
`y`: vector containing target variable
`method`: character string defining method to pass to caret
`loss_metric`: character. Loss metric to evaluate accuracy of model
`resampling_indices`: a list of integer vectors corresponding to the row indices used for each resampling iteration
`tuneGrid`: a data.frame containing hyperparameter values for caret. Should only contain one value for each hyperparameter. Set to NULL if caret method does not have any hyperparameter values.
`trControl`: trainControl object to be passed to caret `train`.
`vars`: character vector specifying variables for which to determine marginal importance Defaults to all predictor variables in x.
`allow_parallel`: boolean for parallel execution. If set to TRUE, user must specify parallel backend in their R session to take advantage of multiple cores. Defaults to FALSE.
`...`: additional arguments to pass to caret `train`

Description

A caret model is trained on training data according to resampling indices specified by the user, and error is calculated on out of sample data. Variable importance is determined by calculating the change in out of sample model performance when a variable is removed relative to baseline out of sample performance when all variables are included.

- **for** $b = 1, ..., B$ **do**
    1. Draw a bootstrap sample of the data
    2. Fit the model and calculate its prediction error Err_b, using the OOB data
    3. Fit a second model, but without variable v, and calculate its prediction error, $Err_{v,b}^{marg}$
- **end for**
- Calculate the marginal VIMP by averaging: $\Delta_v^{marg} = \sum_{b=1}^{B} [Err_{v,b}^{marg} - Err_b]/B$

The workhorse function for Marginal Variable Importance is implemented in `calculate_marginal_vimp` (source).