# Fitting Explainable Boosting Machines in R with the ebm Package

*by Brandon M. Greenwell*

**Abstract** With tabular data, there's certainly no shortage of machine learning algorithms for building models with top-notch performance. These sophisticated models typically pay a heavy price in transparency and often require computationally expensive ad-hoc methods to interpret them. Furthermore, these ad-hoc interpretations tend to only be approximate in nature and require strong assumptions. Explainable boosting machines are a modern machine learning algorithm that can provide first-rate performance while remaining completely interpretable by offering exact explanations for the model's output. Despite their availability in Python, the current implementation in R lacks many several important features, like interactive graphics and support for a wide range of loss functions. We introduce the **ebm** package, which bridges this gap by providing a complete interface to the Python implementation of explainable boosting machines. This new R package enables users to easily fit glassbox models with automatic interaction detection and that are easy to interpret using simple plotting functions to produce power visualizations. With the **ebm** package, R users can now build highly accurate and interpretable models, making it a valuable tool for data scientists and statisticians who require both performance and transparency in their predictive modeling tasks.

## Introduction

Generalized linear models (GLMs) (Nelder and Wedderburn, 1972) have been a bedrock of statistical modeling for over fifty years. GLMs extend the classic linear model to accommodate non-Gaussian response distributions, as well as some degree of non-linearity in the model structure. In particular, GLMs have three components:

1. A random component, which specifies a distribution for the response $Y$ conditional on a set of predictors $x$. In a traditional linear model, for instance, $Y|x$ is often assumed to have a Gaussian (or normal) distribution with constant variance $\sigma^2$.

2. A systematic component, which specifies a linear combination of the regression coefficients and the predictors and is often called the *linear predictor*: $\eta = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p = x^\top \beta$. (Note that linear here refers to the fact that $\eta$ is linear in the regression coefficients $\beta$.)

3. A link function, which specifies how the linear predictor (systematic component) is linked to the conditional mean response $E(Y|x)$ (random component). In logistic regression, for instance, this is the logit function.

In essence, GLMs have the following basic form:

$$g\left(E\left[Y|x\right]\right) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j, \tag{1}$$

where $Y \sim$ some exponential family distribution, $g$ is the link function described above, and $\beta = (\beta_0, \beta_1, \ldots, \beta_p)^\top$ are fixed but unknown regression coefficients to be estimated from the data (typically, via some form of maximum likelihood estimation). The definitive reference for GLMs has always been the monograph by McCullagh and Nelder (1989). Dunn and Smyth (2018) provide a more modern treatment of GLMs, which also focuses on R.

GLMs are often considered as *glassbox* models because of their inherently interpretable structure. For instance, the coefficients ($\beta$) from a GLM offer exact explanations of the model's output, which are often human interpretable (especially for additive models with no interaction effects). This is in contrast to *blackbox* models, where explanations of the model's output are generally approximate in nature and often computationally expensive to acquire. However, the simple structure of the systematic component of a GLM often limits their predictive performance. Generalized additive models (GAMs) (Hastie and Tibshirani, 1990) extend GLMs by replacing the systematic component of (1) with a sum of nonlinear smooth functions to produce

$$g\left(E\left[Y|x\right]\right) = \beta_0 + \sum_{j=1}^{p} f_j\left(x_j\right). \tag{2}$$

The term functions $f_j$ are general enough that they can absorb the intercept, and hence, sometimes we may omit $\beta_0$ in (2). Model (2) is quite flexible in how the response depends on the predictors. Further,

GAMs maintain explainability since the model's output can be understood by simply plotting the individual term contributions (sometimes called shape functions) $f_j\left(x_j\right)$. GAMs can also include interactions effects. For example, a pairwise interaction effect between $x_i$ and $x_j$ can be accommodated by including a bivariate term of the form $f\left(x_i, x_j\right)$. The terms in a GAM can be represented by a variety of functions. Traditionally, the terms in a GAM are represented using some form of smoothing splines. For a thorough overview of GAMs, see Wood (2017).

Lou et al. (2013) introduced generalized additive models plus interactions (GA²Ms) which automatically search for and include important pairwise interaction terms using an algorithm called FAST[1]. In short, FAST is a novel, computationally efficient method for ranking all possible pairs of feature candidates for inclusion in the model. The resulting model consists of univariate (i.e., main effect) terms and a small number of pairwise interactions. Since GA²Ms are low-dimensional, they can still be visualized and interpreted relatively easily by users while also potentially being more accurate. A GA²M has the general form

$$g\left(E\left[Y|\boldsymbol{x}\right]\right) = \sum f_i\left(x_i\right) + \sum f_{ij}\left(x_i, x_j\right) \quad (i \neq j). \tag{3}$$

Here we've simply absorbed the bias term (or intercept) into the shape functions $f_i$. In contrast to tradition GAMs (2), the shape functions in (3) are not necessarily smooth. In particular, modern GA²Ms often rely on tree-based methods (Greenwell, 2022), which tend to produce more rigid step functions.

# 1 Explainable boosting machines

Explainable boosting machines (Nori et al., 2019), or EBMs for short, are a modern class of GA²Ms, that can offer both competitive accuracy and explicit transparency through exact explainability, which are often considered to be two opposing goals of a machine learning (ML) model. For example, full-complexity models, like random forests (Breiman, 2001), tend to be highly competitive in terms of accuracy, but are readily less transparent and explainable (due in part to the high-order interaction effects often captured by such black-box models). Essentially, with EBMs, it's possible to "have your cake and eat it too."

In short, EBMs have the general form

$$g\left(E\left[Y|\boldsymbol{x}\right]\right) = \theta_0 + \sum f_i\left(x_i\right) + \sum f_{ij}\left(x_i, x_j\right) \quad (i \neq j), \tag{4}$$

where,

- $g$ is a link function that allows the model to handle various response types (e.g., the logit link for logistic regression or Poisson deviance for modeling counts and rates);
- $\theta_0$ is a constant intercept (or bias term);
- $f_i$ is the term contribution (or shape function) for predictor $x_i$ (i.e., it captures the main effect of $x_i$ on $E\left[Y|\boldsymbol{x}\right]$);
- $f_{ij}$ is the term contribution for the pair of predictors $x_i$ and $x_j$ (i.e., it captures the joint effect, or pairwise interaction effect of $x_i$ and $x_j$ on $E\left[Y|\boldsymbol{x}\right]$).

Similar to GA²Ms (3), the pairwise interaction terms are determined automatically using the FAST algorithm described in Lou et al. (2013). The primary difference between EBMs and GA²Ms is in how the shape functions are estimated. In particular, EBMs use *cyclic gradient boosting* (Nori et al., 2019; Wick et al., 2019) to estimate the shape function for each feature (and selected pairs of interactions) in a round-robin fashion. A low learning rate is used to help ensure that the order in which the feature effects are estimated does not matter. Apparently, estimating the shape function for each feature one-at-a-time in this manner also helps mitigate potential issues caused by multicollinearity.

In contrast to GA²Ms (3), EBMs also utilize bagging on two different levels, called inner bagging and outer bagging. These additional steps result in increased computation time, but can result in increased accuracy, smoother graphs, and estimated standard errors for the model's outputs. (e.g., for displaying the uncertainty associated with each main effect term in (4)).

Another benefit to using EBMs in practice, as discussed in Wang et al. (2022), is that they're highly editable! As with any ML model, EBMs can learn from and exploit undesirable patterns in the data which can result in potentially harmful predictions if the model is deployed or used to impact

---

[1] As far as I can tell, FAST is not an acronym for anything in particular.

decision making. With the flexibility and transparency of EBMs, in conjunction with tools like GAM Changer (Wang et al., 2022), fitted EBM models can easily and responsibly be edited to fix or remove problematic patterns.

However, in contrast to the more common gradient boosting machine (Friedman, 2001, 2002), or GBM for short, which can ignore irrelevant inputs, EBMs include at least one term in the model for each feature: one main effect ($f_i$) for each predictor, and a term for each pairwise interaction effect selected using the FAST algorithm. This is due to the cyclic nature of the underlying boosting framework. For example, an EBM applied to a training set with $p = 300$ features will result in a model with at least 300 terms, sometimes substantially more!

While EBMs are considered *glass-box* models, an EBM with, say, hundreds or thousands of terms, starts to become much less transparent and explainable. Moreover, the larger the fitted model (i.e., the more terms there are), then the more time it will take the EBM to make predictions, making larger models less fit for deployment and production. We'll return to this issue in a later section with a real application.

### Software

EBMs are implemented as part of the **InterpretML** framework (Nori et al., 2019), which includes both a Python and an R interface called **interpret**. While the Python implementation is fully functional and rich with features (e.g., interactive graphics), the R version (Jenkins et al., 2024) currently lags behind. For instance, the official R interface doesn't currently support regression settings, monotonic constraints, or local explanations. Further, the plotting functionality of the **interpret** R package only supports visualizing a single main effect term using static plots (i.e., no interactive visualizations, no visualizations for pairwise interactions, feature importance plots, or support for visualizing local explanations). Because of these serious drawbacks, we've developed the **ebm** package (Greenwell, 2025), which is a full-featured R interface to the **interpret** Python library powered by **reticulate** (Ushey et al., 2024). In the next section, we'll discuss the **ebm** package in detail and provide several examples that cannot currently be replicated through use of the official **interpret** package in R.

## 2 The ebm package

The **ebm** package is an R wrapper around the explainable boosting functionality of the **interpret** Python library, powered by **reticulate**. The main features of this package, especially when compared to the official **interpret** package in R, include:

- Support for all the objective (i.e., loss) functions available in the Python library (e.g., Poisson deviance for modeling counts).
- Full support for all arguments when fitting EBMs, such as monotonic constraints, alternative missing value strategies, and various regularization parameters.
- Static and interactive graphics for explaining the model at both the global and local level.
- Convenient methods like `print()`, `plot()`, and `predict()`.
- A convenient formula interface for specifying the model structure (e.g., `log(y) ~ x1 + x2 + x3` or `y ~ . - x10`).
- Support for loading serialized or "pickled" EBM models, including those fit directly with the corresponding Python library.
- Support for merging multiple EBMs into one model.
- Built-in parallelism.

### Environment set up

First and foremost, the **ebm** package works through **reticulate** to interface with the underlying Python library **interpret**. Therefore, it's necessary to ensure your environment is configured properly so that all the necessary dependencies are installed and available (i.e., Python itself, along with any required libraries). Personally, I prefer to use **pip** (The Python Packaging Authority, 2025) and **pyenv** (Pyenv project team, 2025) to manage my Python environments, but virtual environments and conda environments are safer and considered best practice. Therefore, in this section, I'll briefly cover setting up a simple virtual environment with the necessary dependencies. For further details and other available options, see **reticulate**'s extensive documentation. In particular, read more about configuring Python, setting up virtual environments, and installing Python libraries in the the following vignettes:

```
vignette("versions", package = "reticulate")  # Python version configuration
vignette("python_packages", package = "reticulate")  # managing envs and deps
```

A Python virtual environment is just an isolated directory with its own libraries and interpreter that are independent of the system-wide Python installation. Virtual environments effectively allow you to manage Python versions and dependencies separately across different projects and development environments.[2] Here, we'll create a new virtual environment called "r-ebm" and install the required Python package **interpret**. Note that it's not necessary to do this through **reticulate**. As noted above, I prefer to do this through the command line with **pip** and **pyenv**. Furthermore, tools for creating and managing virtual environments are part of the standard Python library (as well as external packages), so it's possible to set up virtual environments manually if you know what you're doing.

Assuming you have a valid Python installation available, the code chunk below will create a virtual environment called "r-ebm" and install the **interpret** library into it. Note that the **ebm** package also includes an `install_interpret()` function for convenience, which defaults to creating/using a virtual environment called "r-ebm" by default (see `?ebm::install_interpret` for details).

```r
library(reticulate)

# Create a new environment
virtualenv_create("r-ebm")

# Install the interpret module
virtualenv_install("r-ebm", packages = "interpret")
# Can also call `ebm::install_interpret("r-ebm")`

# Switch to the created virtual environment, if not already there
# use_virtualenv("r-ebm", required = TRUE)

# Load the interpret.glassbox sub-package
interpret <- import("interpret")
print(interpret$`__version__`)
names(interpret$glassbox)
```

```
#> [1] "0.6.9"
```

```
#>  [1] "APLRClassifier"                "APLRRegressor"
#>  [3] "ClassificationTree"            "DecisionListClassifier"
#>  [5] "ExplainableBoostingClassifier" "ExplainableBoostingRegressor"
#>  [7] "LinearRegression"              "LogisticRegression"
#>  [9] "merge_ebms"                    "RegressionTree"
```

Note that "ExplainableBoostingClassifier" and "ExplainableBoostingRegressor" are the core modules from **interpret.glassbox** that power the **ebm** package. In this article, I used Python version .

## A quick comparison of GAMs, GBMs, and EBMs

TBD.

## Predicting and explaining policy ownership (the CoIL 2000 challenge)

In this section, we briefly describe a binary classification example using data from the CoIL 2000 Challenge (van der Putten and van Someren, 2004). The data set, which we'll obtain from the R package **kernlab** (Karatzoglou et al., 2004), consists of 9822 customer records containing 86 variables, including product usage data and socio-demographic data derived from zip area codes. The goal of the challenge was to be able to answer the following question: "Can you predict who would be interested in buying a caravan insurance policy and give an explanation of why?" Hence, being able to explain your model's predictions was key to being successful in this challenge.

To start, we'll load in the data and split into train/test sets using the same partitioning as the original competition (see `?kernlab::ticdata` for variable descriptions and further details):

```r
data("ticdata", package = "kernlab")
ticdata$CARAVAN <- ifelse(ticdata$CARAVAN == "insurance", 1L, 0L)
tictrn <- ticdata[1:5822, ]     # training data (N = 5822)
tictst <- ticdata[-(1:5822), ]  # test data (N = 4000)
```

---

[2]A similar concept for R is provided by the **renv** package (**?**).

Since there's currently no easy way to set the positive class in ebm(), we manually encoded the response factor as a 0/1 outcome, where $Y = 1$ corresponds to CARAVAN = "insurance" (i.e., that the customer did by caravan insurance). This is important since it determines how we'll interpret the model's output later. In the future, I plan to add a convenience argument to make this simpler (e.g., pos_class = "insurance"). Note that R's internal function relevel() from package **stats** won't work here since the conversion happens under the hood on the Python side.

```
library(ebm)

# Fit a default EBM classifier
(tic_ebm <- ebm(CARAVAN ~ ., data = tictrn, objective = "log_loss"))

#>
#> Call:
#> ebm(formula = CARAVAN ~ ., data = tictrn, objective = "log_loss")
#>
#> Python object:
#> ExplainableBoostingClassifier(early_stopping_tolerance=0,
#>                               interaction_smoothing_rounds=100,
#>                               learning_rate=0.04, max_leaves=2, min_hessian=0.0,
#>                               smoothing_rounds=500)
#>
#> Number of features: 85
#> Number of terms: 162
#> Number of interactions: 77
#> Intercept: -3.364
#> Objective: log_loss
#> Link function: logit
```

A proper call to ebm() produces an object of class "EBM" that also inherits the class of the underlying Python object. The main "EBM" class is used to define various methods, like print() (which results in the printed output above), predict(), plot(), and more. In this example, a default call to ebm() produced an EBM with 162 terms (85 main effects, one for each feature, and 77 pairwise interactions) plus an intercept. Further, notice that EBMs use the logit link function for binary outcomes, similar to logistic regression. This is important to call out since the link function determines the scale for interpreting the main effects, pairwise interactions, and local explanations from the model, which we'll discuss shortly.

EBMs are random in nature due to random subsampling (used for early stopping during boosting) and bagging. For reproducibility, you might be tempted to call set.seed() prior to the analysis, but this will have no effect! Since the randomness occurs on the Python side, we have to use the provided random_state argument in the call to ebm(). The default is random_state = 42L which makes the output reproducible from run to run. Setting random_state = NULL generates non-repeatable sequences and therefore the results will not be reproducible.

Predictions can be obtained using the generic predict() method. The type of prediction is controlled via the type argument, which defaults to type = "response". For classification, this results in a matrix of predicted probabilities. Other options useful options here are type = "link" (i.e., predictions on the logit scale) and type = "class" for predicting class labels. Each of these are shown below for the test set:

```
# Compute predictions on the probability scale
head(probs <- predict(tic_ebm, newdata = tictst))

#>                 0          1
#> [1,] 0.9683955 0.03160452
#> [2,] 0.6963962 0.30360384
#> [3,] 0.8479792 0.15202082
#> [4,] 0.9576798 0.04232017
#> [5,] 0.9856882 0.01431179
#> [6,] 0.9835051 0.01649488

# Compute predictions on the link (i.e., logit) scale
head(predict(tic_ebm, newdata = tictst, type = "link"))

#> [1] -3.422340 -0.830195 -1.718839 -3.119250 -4.232256 -4.088073
```

```
# Look at confusion matrix from default predicted class labels
labels <- predict(tic_ebm, newdata = tictst, type = "class")
table("Observed" = tictst$CARAVAN, "Predicted" = labels)

#>         Predicted
#> Observed    0    1
#>        0 3759    3
#>        1  234    4
```

As mentioned earlier, EBMs can also capture uncertainty in the model's output via the outer bagging procedure, which is turned on by default (see the outer_bags parameter in ?ebm::ebm). This will be evident later when we interpret the main effects of the model through various plots. When outer bagging is used, you can also request standard errors for the predictions by setting se_fit = TRUE in the call to predict(). Note, however, that standard errors are only provided for predictions on the link scale, so we need to specify type = "link" as well, as shown below:

```
# Compute predictions on the link (i.e., logit) scale with standard errors
head(predict(tic_ebm, newdata = tictst, type = "link", se_fit = TRUE))

#>           [,1]      [,2]
#> [1,] -3.422340 0.1186275
#> [2,] -0.830195 0.1567391
#> [3,] -1.718839 0.1081985
#> [4,] -3.119250 0.2821688
#> [5,] -4.232256 0.2747545
#> [6,] -4.088073 0.1793779
```

The winning entry, by Charles Elkan (Elkan, 2001), correctly identified 121 caravan policy holders among their 800 top predictions based on the test set. The EBM fitted above does exactly the same according to the cumulative gains computed below:

```
ord <- order(probs[, 2L], decreasing = TRUE)
sum(tictst$CARAVAN[ord[1:800]])  # number of 1s if we sort by highest prob

#> [1] 121
```

The code chunk below uses the well-known **rms** package (Harrell Jr, 2023) to construct a flexible smooth calibration curve based on the predicted probabilities for the test set, along with some relevant performance statistics. For instance, the area under the receiver operating characteristic curve based on the test data for this model is 0.734. In this case, the model slightly overpredicts across the range of probabilities.

```
par(las = 1)
rms::val.prob(probs[, 2L], y = tictst$CARAVAN, cex = 0.5)  # See Figure 1
```
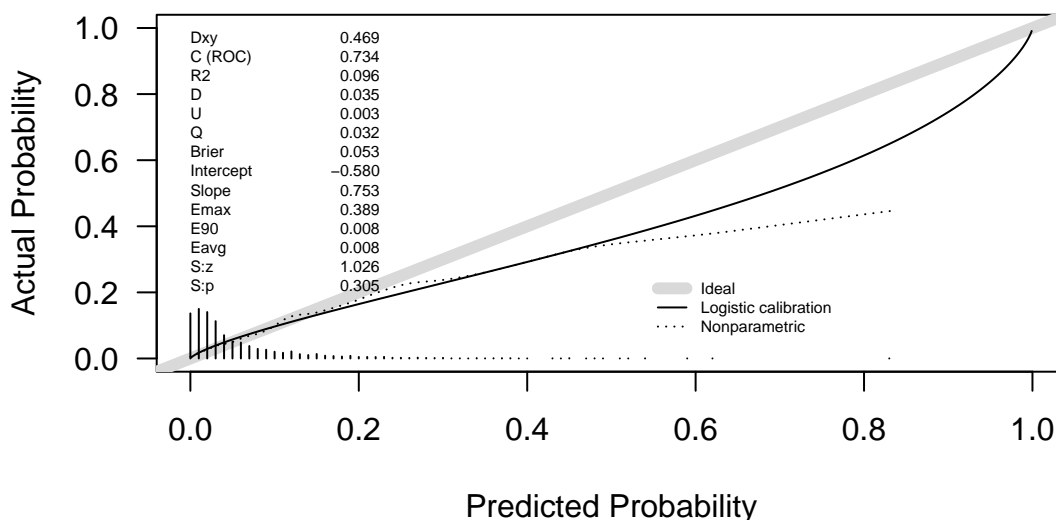


**Figure 1:** Calibration curve and validation statistics for the fitted EBM model based on the the test set.

```
#>           Dxy         C (ROC)              R2                D        D:Chi-sq
#>   4.687465e-01   7.343733e-01   9.569165e-02   3.511593e-02   1.414637e+02
#>           D:p               U         U:Chi-sq             U:p               Q
#>            NA   3.028332e-03   1.411333e+01   8.616484e-04   3.208759e-02
#>         Brier       Intercept           Slope            Emax             E90
#>   5.346921e-02  -5.796053e-01   7.534482e-01   3.888433e-01   7.959127e-03
#>          Eavg             S:z             S:p
#>   7.931846e-03   1.025679e+00   3.050429e-01
```

After training a model, it's often desirable to have a way to persist the model for future use without having to retrain it. In R, we can typically save the result of a fitted model using saveRDS(). Alternatively, some packages provide a more native solution, like **xgboost**'s xgb.save() function (Chen et al., 2024). These solutions will not work here since the connection to the underlying Python object is lost once the session is finished. The proper way to handle this is to use **reticulate**'s py_save_object() and py_load_object() functions to save and load the underlying Python object. This works by serializing the fitted EBM object using Python's **pickle** module. Advanced **reticulate** users could also rely on rely on the **joblib** library for serialization, or even exporting the fitted model to ONNX (Open Neural Network Exchange) via the **ebm2onnx** Python package, depending on the need. For more details on persisting a fitted model in Python, visit https://scikit-learn.org/stable/model_persistence.html. Here we'll show how to properly save and load a fitted EBM model using both **pickle** (through **reticulate**'s built-in functions mentioned above) and **joblib**.

Note that a serialized "EBM" object get stripped of the "EBM" class which is used for calling various methods, like plot(), which we'll discuss later. To overcome this issue, the **ebm** package provides an as.ebm() function for coercing an appropriate Python object to an "EBM" object that can be used with various methods. This of course also means that you can import EBM models fitted with the **interpret** Python package directly into R for use with print(), predict(), plot(), etc.

```
# Save a fitted EBM model
reticulate::py_save_object(tic_ebm, filename = "tic_ebm.pkl")

# Load a pickled EBM model
(ebm_obj <- reticulate::py_load_object("tic_ebm.pkl"))

#> ExplainableBoostingClassifier(early_stopping_tolerance=0,
#>                               interaction_smoothing_rounds=100,
#>                               learning_rate=0.04, max_leaves=2, min_hessian=0.0,
#>                               smoothing_rounds=500)

(tic_ebm <- as.ebm(ebm_obj))  # adds "EBM" class for using print(), plot(), etc.

#>
#> Call:
#> NULL
#>
#> Python object:
#> ExplainableBoostingClassifier(early_stopping_tolerance=0,
#>                               interaction_smoothing_rounds=100,
#>                               learning_rate=0.04, max_leaves=2, min_hessian=0.0,
#>                               smoothing_rounds=500)
#>
#> Number of features: 85
#> Number of terms: 162
#> Number of interactions: 77
#> Intercept: -3.364
#> Objective: log_loss
#> Link function: logit
```

Note that the original function call prints as NULL since it's impossible to determine from the specific object we called as.ebm() on.

In the specific case of an EBM, it may be more useful to use the **joblib** Python package, which provides the joblib.dump() and joblib.load() functions for serializing and deserializing a Python object, respectively; this approach is more efficient on objects that internally store large **numpy** arrays (Harris et al., 2020). This is usually the case for a fitted EBM model since all of the term contributions, etc. are stored as **numpy** arrays. As of this writing, **reticulate** does not support **joblib** through any convenience function (e.g., like py_save_object()), so we'll need to interface with the module directly,

as shown below (note that you may have to install **joblib** using one of the methods discussed earlier for installing/managing Python libraries):

```
joblib <- reticulate::import("joblib")  # assumes the joblib pkg is available

# "Dump" the fitted EBM model into a pickle file
joblib$dump(tic_ebm, filename = "tic_ebm_joblib.pkl")

#> [1] "tic_ebm_joblib.pkl"

# Load a pickled EBM model
(tic_ebm <- as.ebm(joblib$load("tic_ebm_joblib.pkl")))

#>
#> Call:
#> NULL
#>
#> Python object:
#> ExplainableBoostingClassifier(early_stopping_tolerance=0,
#>                               interaction_smoothing_rounds=100,
#>                               learning_rate=0.04, max_leaves=2, min_hessian=0.0,
#>                               smoothing_rounds=500)
#>
#> Number of features: 85
#> Number of terms: 162
#> Number of interactions: 77
#> Intercept: -3.364
#> Objective: log_loss
#> Link function: logit
```

Fitted "EBM" objects can be understood and visualized using the generic `plot()` method. By default, `plot()` relies on **ggplot2** (Wickham, 2016) for variable importance and main effect plots; for visualizing pairwise interactions, it relies on **lattice**'s `levelplot()` function (Sarkar, 2008). For convenience, we'll also use **patchwork** (Pedersen, 2024) in these examples for configuring the layout of displays with multiple plots.

Below, we use the `plot()` method to display a few global summaries of the fitted model. First, we display the overall term importance scores. By default, all terms in the model are plotted so it's a good idea to set the `n_terms` argument to something more reasonable, especially if the model contains lots of terms. Here we tell it to display the top 15 terms in the model. The importance scores are computed as the mean absolute score from each term in the model.

```
library(ggplot2)
library(patchwork)

theme_set(theme_bw())  # my preferred theme for ggplot2

# Plot term importance scores (see Figure 2)
plot(tic_ebm, n_terms = 15)
```

The default behavior is to produce a Cleveland dot plot, but you can switch to a bar plot by setting `geom = "bar"` or `geom = "col"` in the call to `plot()`. Setting `horizontal = TRUE` will also produce a plot that's been flipped on its axes. See `?ebm::plot.EBM` for details.

Next, we'll plot the main effect for PPERSAUT, the highest rated term in the model, by setting `term = "PPERSAUT"` in the call to `plot()`; note that this produces a plot of the corresponding shape function, $f$(PPERSAUT), from (4). It's a good idea to also include some information regarding the distribution of the variable of interest to help avoid extrapolation and over interpreting the estimated main effects. The results are displayed in Figure 3 (left side).

```
plot(tic_ebm, "PPERSAUT", horizontal = TRUE) +  # See Figure 3
  ggplot(tictrn, aes(PPERSAUT)) +  # add distribution plot
  geom_bar() +
  scale_x_discrete(drop = FALSE) +  # don't drop zero counts
  xlab("") +
  coord_flip()
```
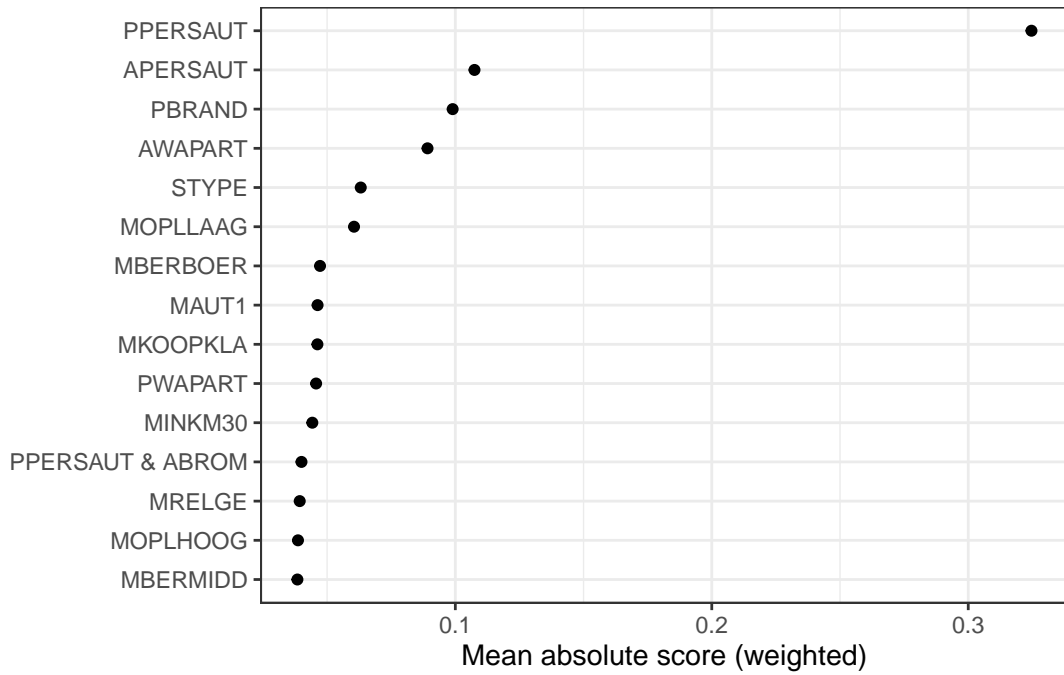
**Figure 2:** Term importance scores from the fitted EBM model computed as the mean absolute contribution from each term in the model. Only the top 15 terms are displayed here.
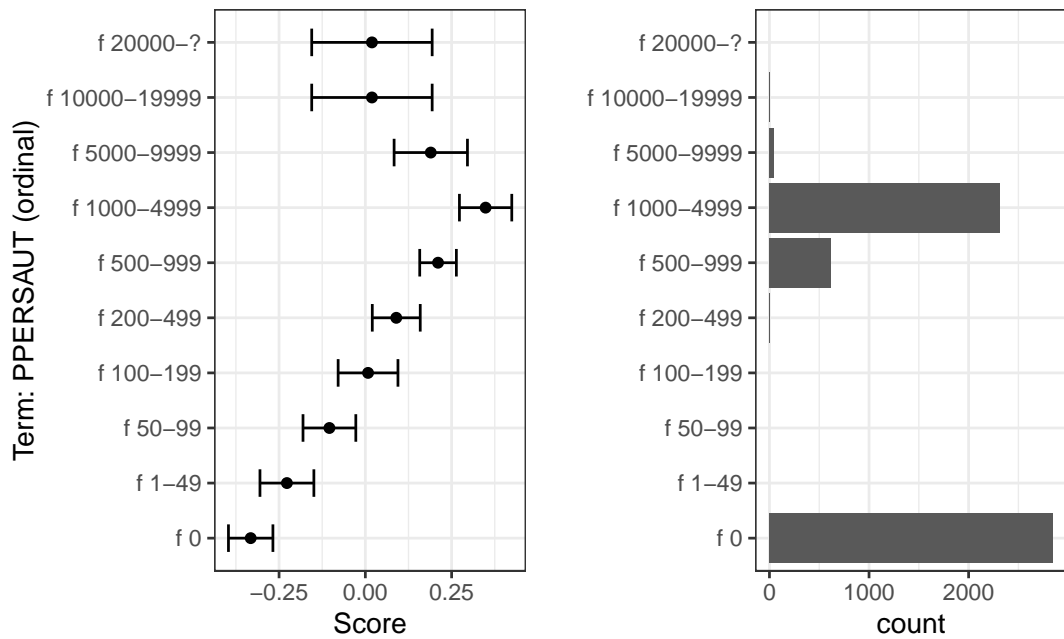


**Figure 3:** Term contribution for PPERSAUT from the fitted EBM model. Left: Main effect with standard error bars. Right: Distribution of PPERSAUT in the training data.

Note that PPERSAUT is an ordered factor representing the contribution level for car policies. The left side of Figure 3 shows a generally increasing relationship between contribution to car policies and the log odds of buying a caravan insurance policy. The distribution of PPERSAUT in the training data is also shown on the right side of Figure 3. You can similarly display heatmaps for pairwise interactions, but we'll reserve this for an example in a later section of this paper. Recall that EBMs provide estimates of uncertainty in the model's output whenever outer bagging is used. This is also displayed in the main effect plots; in Figure 3, for example, these are displayed as error bars in the dot chart.

One attractive feature of the **interpret** library in Python is the use of interactive graphics and dashboards (e.g., via Plotly (Plotly Technologies Inc., 2015)). This feature is not available in the associated **interpret** R package. However, since **ebm** exposes the full functionality of the Python version through **reticulate**, the plot() method can also be used to produce the same HTML-based

interactive graphics. These can be viewed either in the browser (the default) or in an interactive viewer like the one built into RStudio or VS Code. Below we reproduce the main effect term for PPERSAUT using an interactive Plotly/HTML-based graphic that will open by default in a browser. This will work for any of the visualizations discussed in this paper. Note, however, that while **ebm** does support multiclass outcomes, only interactive visualizations are available in these cases (i.e., no static **ggplot2**-based plots for multiclass outcomes). Of course, these plots can always be constructed quite easily following the approach discussed at the end of this section.

```
plot(tic_ebm, term = "PPERSAUT", interactive = TRUE)  # See Figure 4
```
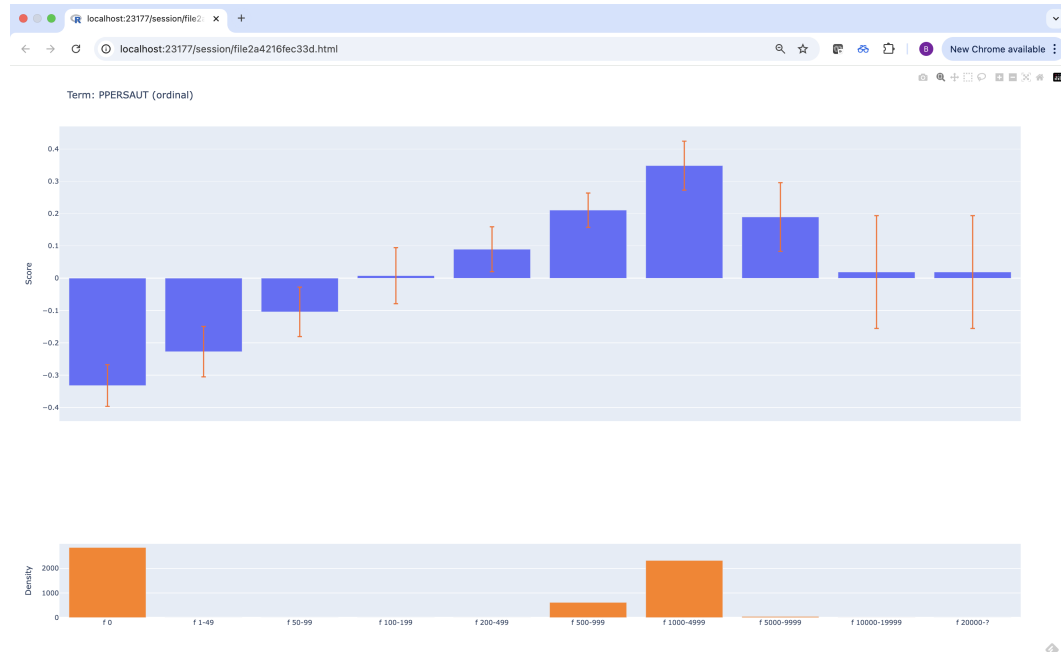


**Figure 4:** Screenshot of an interactive visualization for the main effect of PPERSAUT displayed in a Google Chrome browser.

Let's now look at the second most important term, APERSAUT, which corresponds to the main effect of a continuous feature representing the number of car policies owned. The main effect for APERSAUT is displayed in the left side of Figure 5. We might expect there to be an increasing monotonic relationship between APERSAUT and the log odds of buying a caravan insurance policy. In fact, it's often desirable that the shape functions $f_j\left(x_j\right)$ in (4) be constrained in some way. This may be due to business considerations, or because of some a priori belief about the underlying relationships between the predictors and the response. In such cases, we may wish to enforce a type of constraint called monotonicity on some of the terms in the model. This means, for example, that the relationship between $x_j$ and $y$, as modeled through $f_j\left(x_j\right)$, should always be increasing or decreasing. For instance, a monotonic increasing relationship would ensure that $f_j\left(x_j\right) \geq f_j\left(x_j'\right)$ whenever $x_j \geq x_j'$ (and vice versa for a decreasing monotonic relationship).

To illustrate the idea, we'll enforce a decreasing monotonic relationship between APERSAUT and the log odds of purchasing a caravan insurance policy. While we can enforce monotonic constraints via the monotone_constraints argument in the call to ebm(), the **interpret** authors generally recommend forcing monotonicity instead by post-processing the graphs of the fitted model using isotonic regression (**?**). This is the recommended approach as it prevents the model from compensating for the monotonicity constraints by learning non-monotonic effects in other highly-correlated features. We can follow this route by calling the $monotonize() method that's bound to the fitted "EBM" object. Note that calling this method will modify the original object in place, so to preserve the original unconstrained model, we'll first call the $copy() bound method to make a deep copy. Additionally, we lose any uncertainty associated with the term from the outer bagging in the original model.

```
tic_ebm_mono <- as.ebm(tic_ebm$copy())  # make a deep copy to leave original intact
tic_ebm_mono$monotonize("APERSAUT", increasing = FALSE)

#> ExplainableBoostingClassifier(early_stopping_tolerance=0,
#>                               interaction_smoothing_rounds=100,
```

```
#>                                     learning_rate=0.04, max_leaves=2, min_hessian=0.0,
#>                                     smoothing_rounds=500)

# Display results side by side (see Figure 5)
plot(tic_ebm, term = "APERSAUT") + ggtitle("Original") +
  plot(tic_ebm_mono, term = "APERSAUT") + ggtitle("Monotonized")
```
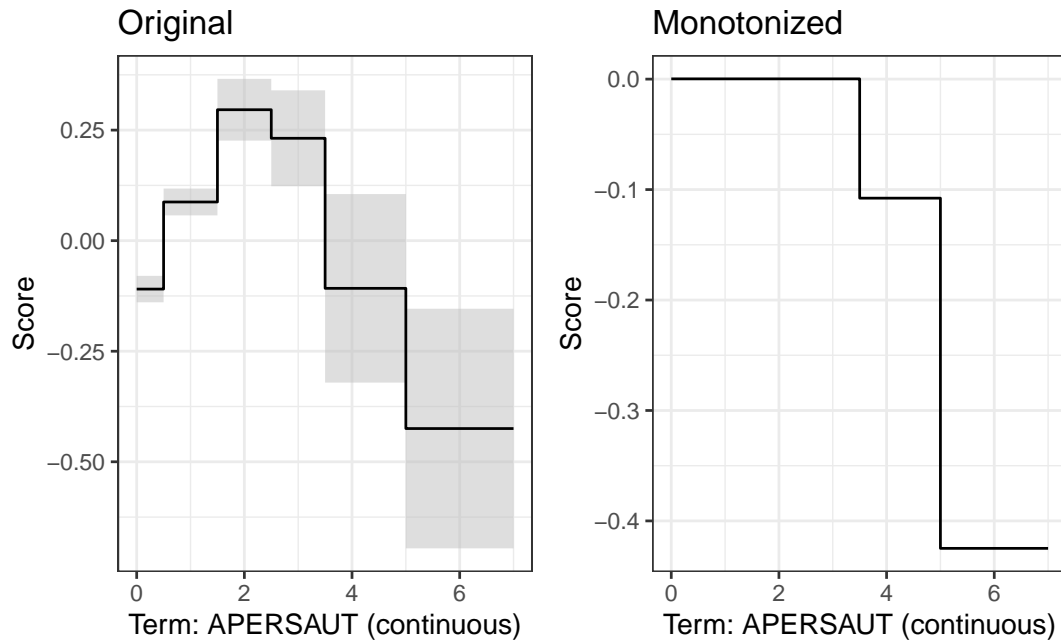


**Figure 5:** Term contribution for `APERSAUT` from the fitted EBM model. Left: Main effect with piecewise standard error band from the original unconstrained model. Right: Main effect with forced decreasing monotonicity through post-pocessing the left graph using isotonic regression.

While the **ebm** package does do not expose 100% of the functionality available in the corresponding Python library through simple convenience functions, the above example shows that users can pretty much do anything they need by interacting directly with the underlying Python objects (the magic happens through **reticulate**). For example, to reproduce the original HTML-based visualization, which will be displayed in a browser, you can do the following:

```
idx <- as.integer(which(tic_ebm$term_names_ == "APERSAUT") - 1L)
plt <- tic_ebm$explain_global()$visualize(idx)
plt$show()  # should open in a browser
# Can also use `plt$write_html("<path/to/file.html>")` to write to HTML file
```

Details aside, you can access the data to recreate any of the static plots in this article (or those produced by the `plot()` method in general), by coercing the relevant internal Python plotly object to an ordered dictionary, which **reticulate** will automatically convert to a list. For instance, the following snippet of code extracts the main data used in generating the left plot in Figure 5 (this is essentially what `plot()` does under the hood).[3]

```
plt$to_ordered_dict()$data[[2L]]  # main effect only (i.e., no error bounds)

#> $fill
#> [1] "tonexty"
#>
#> $fillcolor
#> [1] "rgba(68, 68, 68, 0.15)"
#>
#> $line
#> $line$color
#> [1] "rgb(31, 119, 180)"
#>
```

---

[3]Don't forget that Python indexing starts at 0!

```
#> $line$shape
#> [1] "hv"
#>
#>
#> $mode
#> [1] "lines"
#>
#> $name
#> [1] "Main"
#>
#> $type
#> [1] "scatter"
#>
#> $x
#> [1] 0.0 0.5 1.5 2.5 3.5 5.0 7.0
#>
#> $xaxis
#> [1] "x"
#>
#> $y
#> [1] -0.10947771  0.08747734  0.29608178  0.23149262 -0.10776385 -0.42483175
#> [7] -0.42483175
#>
#> $yaxis
#> [1] "y"
```

The visualizations discussed so far are examples of *global explanations*. We can also explain the model's output at the local (i.e., prediction) level. This is more commonly done for general ML models using techniques like SHAP (SHapley Additive exPlanations) (Lundberg and Lee, 2017) and LIME (Local Interpretable Model-agnostic Explanations) (Ribeiro et al., 2016). These post-hoc methods typically make strong assumptions about feature independence and so forth and only provide an approximation as to how each feature (or subset thereof) contributes to a model's output. With EBMs, we can visualize exactly how each term in the model contributes to a specific prediction (this is true for GAMs in general).

Below, we call the `plot()` method to explain the predicted output associated with the first instance of the test sample. For this customer, the EBM predicts a very low probability of purchasing a caravan insurance policy ($\hat{p}(x) = 0.0316$), and we'd like to know how each term in the model contributed to that prediction. Note that we need to specify a couple of additional arguments in the call to `plot()` here. In particular, we need to set `local = TRUE` and provide the input features as a data frame via the `X` argument. You can optionally provide the response values as shown below, but this only has an effect when setting `interactive = TRUE`, which results in displaying the associated predicted and observed response value at the top of the graph. The results are displayed in Figure 6.

```
newx <- subset(tictst[1L, ], select = -CARAVAN)
newy <- tictst$CARAVAN[1L]  # not useful unless `interactive = TRUE`
plot(tic_ebm, local = TRUE, X = newx, y = newy)  # See Figure 6
```

From Figure 6 we can see that the biggest drivers for this customer's low score were the fact that they did not appear to have or make any contribution toward existing car policies.

### Predicting ALS progression (the DREAM challenge prediction prize)

In this section, we'll look at a regression example using the ALS data from Efron and Hastie (2016, p. 349). A description of the data, along with the original source and download instructions, can be found at https://hastie.su.domains/CASI/data.html. These data concern 1822 observations on *amyotrophic lateral sclerosis* (ALS or Lou Gehrig's disease) patients. The goal is to predict ALS progression over time, as measured by the slope (or derivative) of a functional rating score (i.e., column dFRS), using 369 available predictors obtained from patient visits. The data were originally part of the DREAM-Phil Bowen ALS Predictions Prize4Life challenge. The winning solution (Küffner et al., 2015) used a Bayesian tree ensemble quite similar to a random forest, while Efron and Hastie (2016, chap. 17) analyzed the data using gradient tree boosting via the R package **gbm** (Ridgeway, 2024).

Below, we load in the ALS data from a URL and fit a default EBM. Note that the data already contain an indicator for the train/test splits, so we use that to split the data and discard the column
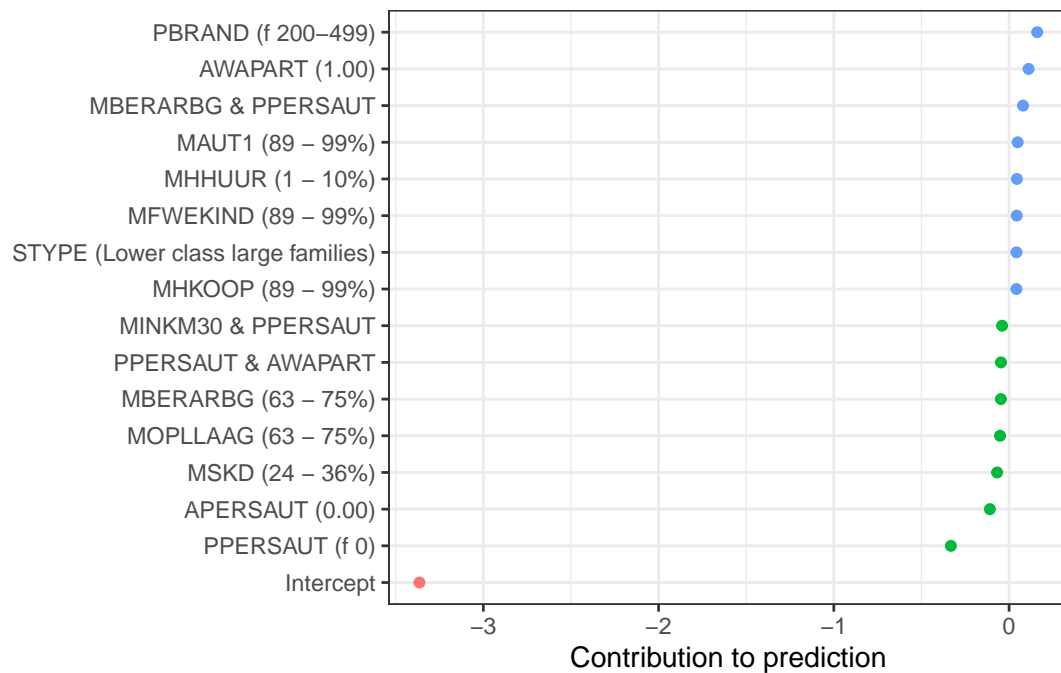
**Figure 6:** Local contributions from the fitted EBM model to the predicted outcome for the first customer in the test set.

after. We also compute the mean square error (MSE) on the test set for comparison to a simpler model obtained later.

```
# Laod data and split into train/test sets
url <- "https://hastie.su.domains/CASI_files/DATA/ALS.txt"
als <- read.table(url, header = TRUE)
alstrn <- subset(als, subset = !testset, select = -testset)  # training data
alstst <- subset(als, subset = testset, select = -testset)   # test data

# Fit a default EBM regressor
(als_ebm <- ebm(dFRS ~ ., data = alstrn, objective = "rmse"))

#>
#> Call:
#> ebm(formula = dFRS ~ ., data = alstrn, objective = "rmse")
#>
#> Python object:
#> ExplainableBoostingRegressor(early_stopping_tolerance=0)
#>
#> Number of features: 369
#> Number of terms: 702
#> Number of interactions: 333
#> Intercept: -0.6802
#> Objective: rmse
#> Link function: identity

# Compute MSE on the test set
pred <- predict(als_ebm, newdata = alstst)
(mse_tst <- mean((alstst$dFRS - pred)^2))

#> [1] 0.2669048
```

The fitted EBM model contains 702 terms (369 main effect terms, one for each predictor, and 333 pairwise interaction terms) plus an intercept. The MSE for this model on the test data was 0.267. The top 10 terms are displayed in Figure 7 below. Here, we see that `Onset.Delta` is by far the most important predictive feature in the model.

```
plot(als_ebm, n_terms = 10)  # See Figure 7
```
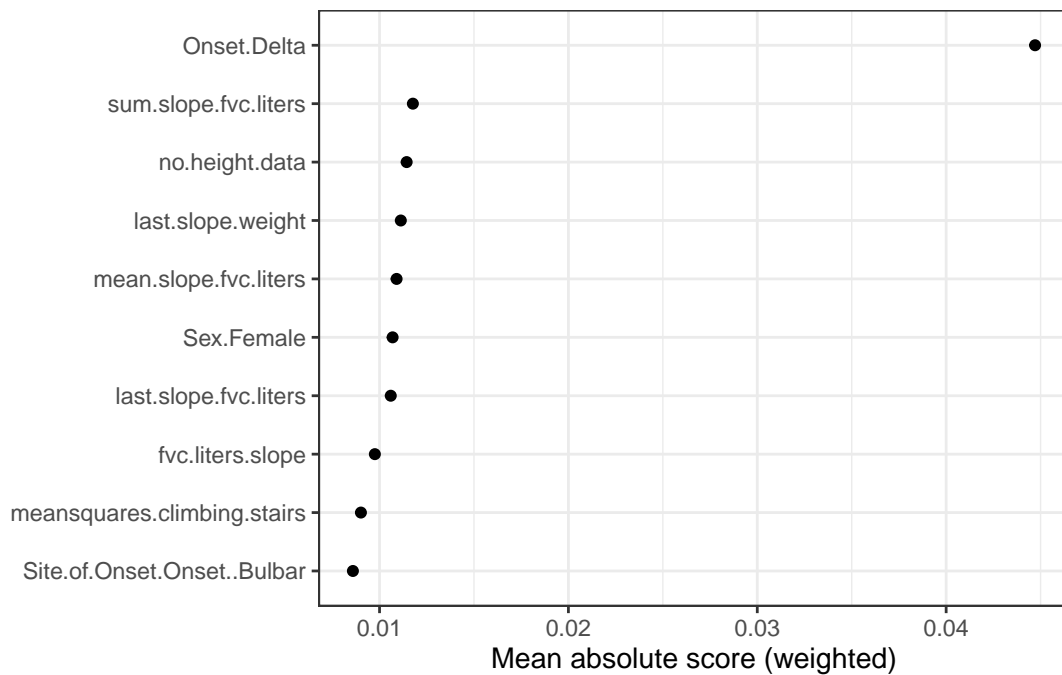
**Figure 7:** Term importance scores from the fitted EBM model computed as the mean absolute contribution from each term in the model. Only the top 10 terms are displayed.

As briefly mentioned in the previous example, we can also plot pairwise interaction effects using a heatmap (or contour plot). Below we plot the term contributions for both $f$ (`Onset.Delta`) (a main effect) and $f$ (`Onset.Delta`, `last.slope.weight`) (a pairwise interaction). The results are displayed in Figure 8. Note that since `plot()` relies on **lattice** for pairwise interaction plots, we use **gridExtra** (Auguie, 2017) to display the two **grid**-based plots together.

```
p1 <- plot(als_ebm, term = "Onset.Delta")  # main effect
p2 <- plot(als_ebm, term = c("Onset.Delta", "last.slope.weight"))  # interaction
gridExtra::grid.arrange(grobs = list(p1, p2), nrow = 1)  # See Figure 8
```
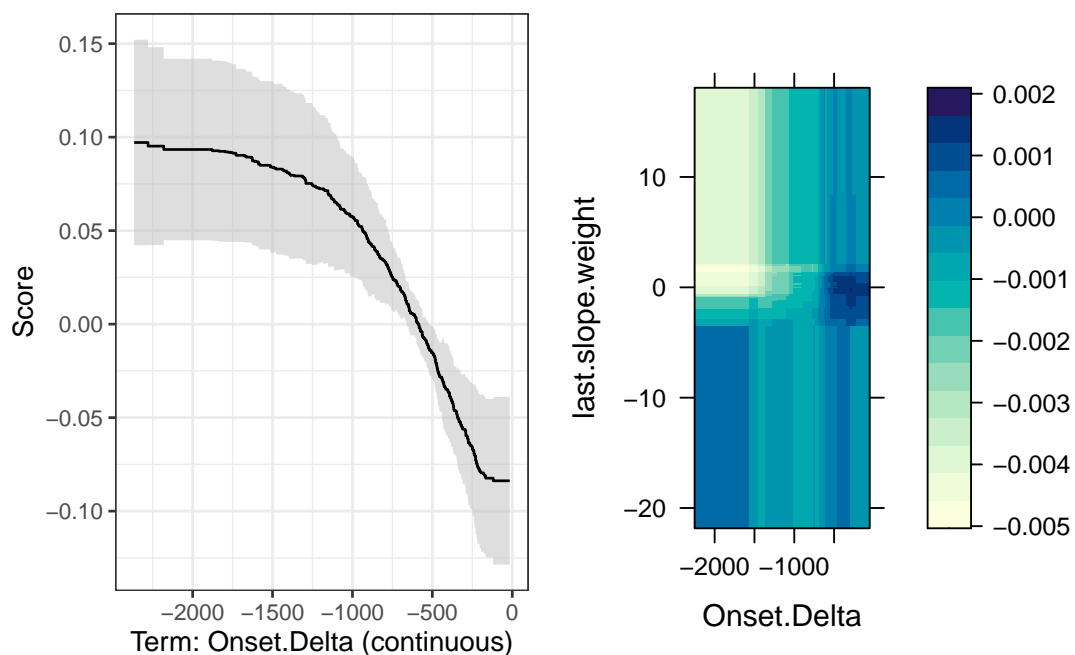


**Figure 8:** Term contributions from the fitted EBM model to the ALS data. Left: Main effect of `Onset.Delta`. Right: Pairwise interaction effect of `Onset.Delta` & `last.slope.weight`.

From Figure 8, we can see that increasing `Onset.Delta` is associated with a decrease in the predicted

outcome (a lterger outcome is better here). This confirms that a longer time since diagnosis predicts a lower decline. The pairwise interaction plot suggests that the relationship between `Onset.Delta` and predicted dFRS is reversed between negative and nonnegative values of `last.slope.weight` (the difference in weight between the last two visits).

Compared to "black-box" models, like random forests and deep neural networks, EBMs are considered "glass-box" models that can be competitively accurate while also maintaining a higher degree of transparency and explainability. However, as discussed earlier, EBMs become readily less transparent and harder to interpret in high-dimensional settings with many predictor variables; they also become more difficult to use in production due to increases in scoring time. To that end, Greenwell et al. (2023) proposed a simple solution based on the least absolute shrinkage and selection operator (LASSO) (Tibshirani, 1996). Their approach can help introduce sparsity by reweighting the individual model terms and removing the less relevant ones, thereby allowing these models to maintain their transparency and relatively fast scoring times in higher-dimensional settings. The idea is similar in spirit to the importance sampled learning ensemble strategy proposed in Friedman et al. (2003). In short, post-processing a fitted EBM with many (i.e., possibly hundreds or thousands) of terms using the LASSO can help reduce the model's complexity and drastically improve scoring time.

We'll illustrate the basic idea using the previous model, which contains a total of 703 terms comprised of the main effects, pairwise interactions, and an intercept. In the next chunk, we'll construct data matrices consisting of the individual term contributions for both the train and test sets. For example, the $j$-th column of X_trn is given by the contribution of the $j$-th term of the model $\left\{ f_j \left( x_{ij} \right) \right\}_{i=1}^{n}$ over the training set. These data matrices will be used as input to the LASSO in the next step.

```
X_trn <- predict(als_ebm, newdata = alstrn, type = "terms")
X_tst <- predict(als_ebm, newdata = alstst, type = "terms")
colnames(X_trn) <- colnames(X_tst) <- als_ebm$term_names_
print(dim(X_trn))  # same rows as alstrn, but one column for each term in model

#> [1] 1197  702

# Sanity check (should be equivalent)
head(cbind(
  rowSums(X_trn) + c(als_ebm$intercept_),
  predict(als_ebm, newdata = alstrn)  # additive on link scale
))

#>            [,1]       [,2]
#> [1,] -0.7866218 -0.7866218
#> [2,] -0.1947158 -0.1947158
#> [3,] -0.5743338 -0.5743338
#> [4,] -1.1612496 -1.1612496
#> [5,] -0.8793495 -0.8793495
#> [6,] -0.5021240 -0.5021240
```

Next, we use the **glmnet** package (Friedman et al., 2010; Tay et al., 2023) to fit the entire LASSO path using the new training data comprised of the individual term contributions from the fitted EBM model. We then assess performance using the associated test set and collect the results.

```
library(glmnet)

# Fit the entire LASSO path using the term contributions, f(x), as inputs
lasso <- glmnet(X_trn, y = alstrn$dFRS, lower.limits = 0, standardize = FALSE)

# Assess performance of the LASSO fit using the independent test set
perf <- assess.glmnet(lasso, newx = X_tst, newy = alstst$dFRS)
perf <- do.call(cbind, args = perf)  # bind results into matrix

# Collect results and sort by number of non-zero coefficients/terms
res <- as.data.frame(cbind("n_terms" = lasso$df, perf, "lambda" = lasso$lambda))
head(res <- res[order(res$n_terms), ])

#>    n_terms       mse       mae      lambda
#> s0       0 0.3203724 0.4583318 0.010792530
#> s1       1 0.3132038 0.4529917 0.009833751
#> s2       1 0.3072654 0.4484656 0.008960148
```

```
#> s3       1 0.3023472 0.4444106 0.008164153
#> s4       1 0.2982750 0.4410089 0.007438872
#> s5       1 0.2949041 0.4381329 0.006778023
```

Next, we'll extract the value of the regularization parameter (lambda) associated with the smallest test MSE. The results displayed below show that a reweighted EBM with only 23 terms (including the intercept) obtains a test MSE of 0.262.

```
lambda <- res[which.min(res$mse), "lambda"]
res[which.min(res$mse), ]

#>     n_terms      mse       mae        lambda
#> s36      23 0.262219 0.4040236 0.0003789464
```

Finally, we can plot the results which are displayed in Figure 9. On the left, we show the LASSO coefficient path and on the right we show the test deviance as a function of the number of terms in the compressed model.

```
# Plot results (see Figure 9)
par(mfrow = c(1, 2), mar = c(4, 4, 0.1, 0.1), cex.lab = 0.95,
    cex.axis = 0.8, mgp = c(2, 0.7, 0), tcl = -0.3, las = 1)
plot(lasso, xvar = "lambda", col = adjustcolor("darkred", alpha.f = 0.3),
     xlab = "Log lambda")
abline(v = log(lambda), lty = 2, col = 1)
plot(res[, c("n_terms", "mse")], type = "l", las = 1,
     xlab = "Number of terms", ylab = "Test MSE")
abline(h = mse_tst, lty = 2, col = "darkred")
```
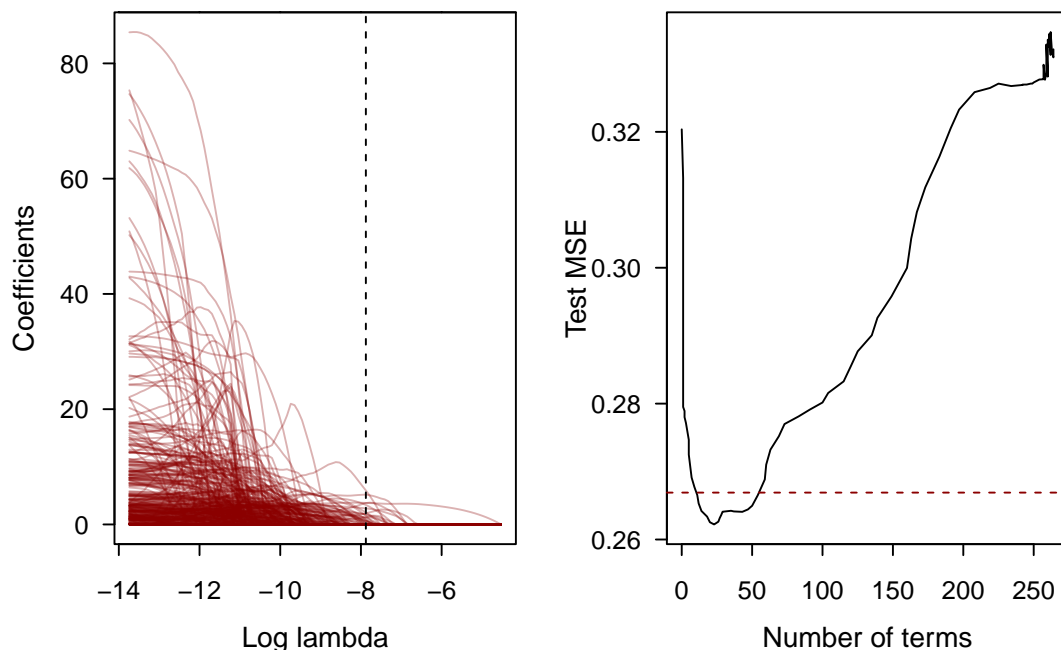


**Figure 9:** Results from post-processing the fitted EBM model using the LASSO. Left: LASSO coefficient path (one coefficient for each term in the EBM model). Right: Number of non-zero coefficients/EBM terms vs. the corresponding test MSE.

The test MSE for the compressed model is 0.262, which is actually better than the original model with all 703 terms! Quite impressive in this case.

Lastly, we can call the $scale() and $sweep() methods bound to the original EBM object to reweight and purge any unused terms from the compressed (i.e., reweighted) EBM model (the new term importance scores are displayed in Figure 10):

```
als_ebm_lasso <- as.ebm(als_ebm$copy())
weights <- coef(lasso, s = lambda)  # LASSO coefficients (most are zero!)
weights <- setNames(as.numeric(weights), nm = rownames(weights))
for (i in seq_along(weights[-1L])) {
```

```
  idx <- as.integer(i - 1L)  # Sigh, don't forget Python indexing starts at 0!
  als_ebm_lasso$scale(idx, factor = weights[i + 1])
}
als_ebm_lasso$intercept_ <- weights[1L]

# Remove unused terms!
als_ebm_lasso$sweep(terms = TRUE, bins = TRUE, features = FALSE)

#> ExplainableBoostingRegressor(early_stopping_tolerance=0)

length(als_ebm_lasso$term_names_)

#> [1] 23

plot(als_ebm_lasso)  # show new term importance scores (see Figure 10)
```
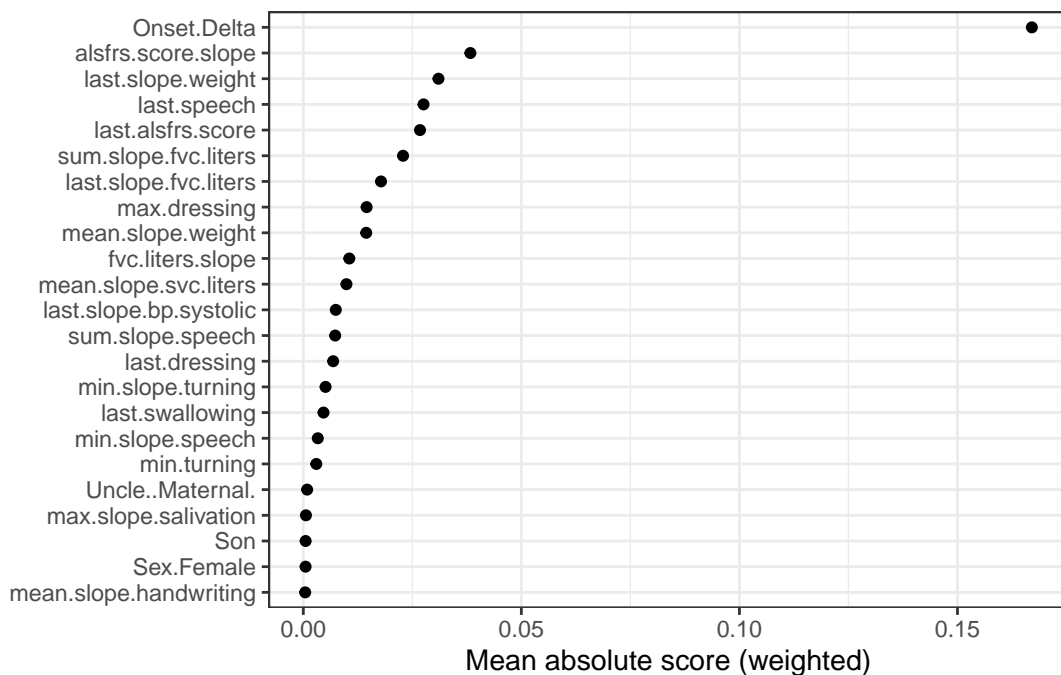


**Figure 10:** Term importance scores from the compressed EBM model computed as the mean absolute contribution from each term in the model.

And just as a sanity check, we can see that the compressed EBM model does produce the right predictions by comparing it to the output from the corresponding LASSO model:

```
# Compare predictions between LASSO and compressed EBM (should be the same!)
head(cbind(
  "EBM (Compressed)" = predict(als_ebm_lasso, newdata = alstst),
  "LASSO" = predict(lasso, newx = X_tst, s = lambda, type = "response")[, 1L]
))

#>      EBM (Compressed)      LASSO
#> [1,]       -0.2869531 -0.2869531
#> [2,]       -0.3862899 -0.3862899
#> [3,]       -0.4750451 -0.4750451
#> [4,]       -0.2647092 -0.2647092
#> [5,]       -0.8570664 -0.8570664
#> [6,]       -1.0047789 -1.0047789
```

Note that Greenwell et al. (2023) provide a similar compression analysis applied to the CoIL 2000 example described in the previous example by interfacing directly with the **interpret** Python library using **reticulate**.

**Parallelism in EBMs**

The core implementation of EBMs through the **interpret** Python library takes advantage of **joblib** (Varoquaux et al., 2024) to provide multi-core and multi-machine parallelization. Note that this differs from traditional parallelization in R through packages like **parallel** (R Core Team, 2023). Here the parallelization is taken care of completely on the Python side.

This can be controlled in the call to ebm() via the n_jobs argument. Note that negative integers are interpreted using **joblib**'s formula: n_cpus + 1 + n_jobs. For example, setting n_jobs = -2 will result in using all threads except for one. The default is n_jobs = -1 which results in using all available CPUs.

Below is a brief example using the Bikeshare data available in the **ISLR2** (James et al., 2022) package. Note that EBMs are parallelized at the outer bags level, which are used to generate error bounds and help with smoothing the graphs.

```
keep <- c("workingday", "temp", "weathersit", "mnth", "hr", "bikers")
bikeshare <- ISLR2::Bikeshare[, keep]

# Use all CPUs (n_jobs = -1; this is the default
system.time({
  ebm1 <- ebm(bikers ~ ., data = bikeshare, objective = "poisson_deviance",
              inner_bags = 20, outer_bags = 16, n_jobs = -1)
})

#>    user  system elapsed
#>   0.052   0.007   7.288

# Use one CPU (n_jobs = 1)
system.time({
  ebm2 <- ebm(bikers ~ ., data = bikeshare, objective = "poisson_deviance",
              inner_bags = 20, outer_bags = 16, n_jobs = 1)
})

#>    user  system elapsed
#>  45.445   0.517  45.966

# Use all CPUs, but no bagging
system.time({
  ebm3 <- ebm(bikers ~ ., data = bikeshare, objective = "poisson_deviance",
              inner_bags = 0, outer_bags = 1, n_jobs = -1)
})

#>    user  system elapsed
#>   0.024   0.002   1.100

# Plot results (see Figure 11)
plot(ebm2, term = "temp") + plot(ebm3, term = "temp")
```

**Merging several EBMs into a single model**

The underlying **interpret** Python library allows you to merge multiple EBM models trained on similar data sets that have the same set of features. This can be useful for a number of reasons. For instance, you can fit several independent EBM models to implement your own outer bagging strategy. Or perhaps you've fit several EBMs over time as data have become available and you want to merge several of the models into one.

In the code chunk below, we fit several EBMs using R's built in mtcars data set (see ?datasets::mtcars for details) and then merge them into a single EBM model. The merged EBM model retains the glass-box interpretability of the component EBMs, allowing us to view both global and local explanations of the combined model. Note that outer bagging is turned off for the individual component EBMs fitted below, but the merged model displays uncertainty in the learned graph for cyl (i.e., the number of cylinders). The results are displayed in Figure **??**.

```
# Generate list of EBMs with different random seeds
ebms <- lapply(1:3, FUN = function(i) {
  ebm(mpg ~ ., data = mtcars, outer_bags = 1, random_state = i, obj = "rmse")
```
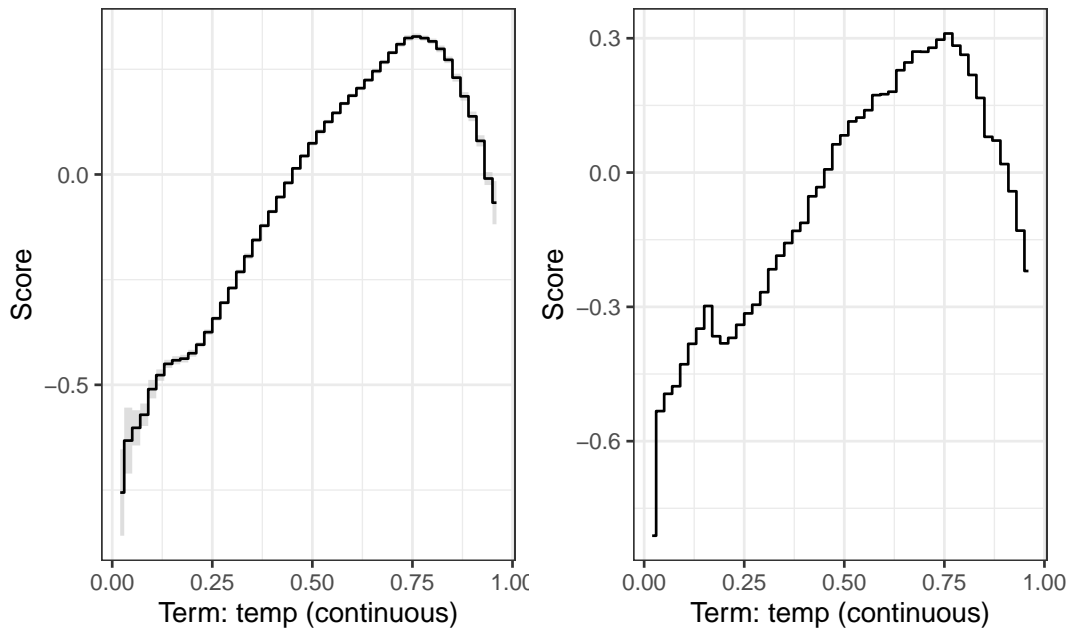
**Figure 11:** Main effect of temperature in Celsius. Left: Outer and inner bagging. Right: No bagging.

```
})

# Merge EBMs into one
merged <- do.call(merge, args = ebms)

# Display plots for "cyl" term in 2x2 grid (see Figure 10)
cyl_plots <- lapply(c(ebms, merged), FUN = plot, term = "cyl")
gridExtra::grid.arrange(grobs = cyl_plots, nrow = 2)  # See Figure 12
```
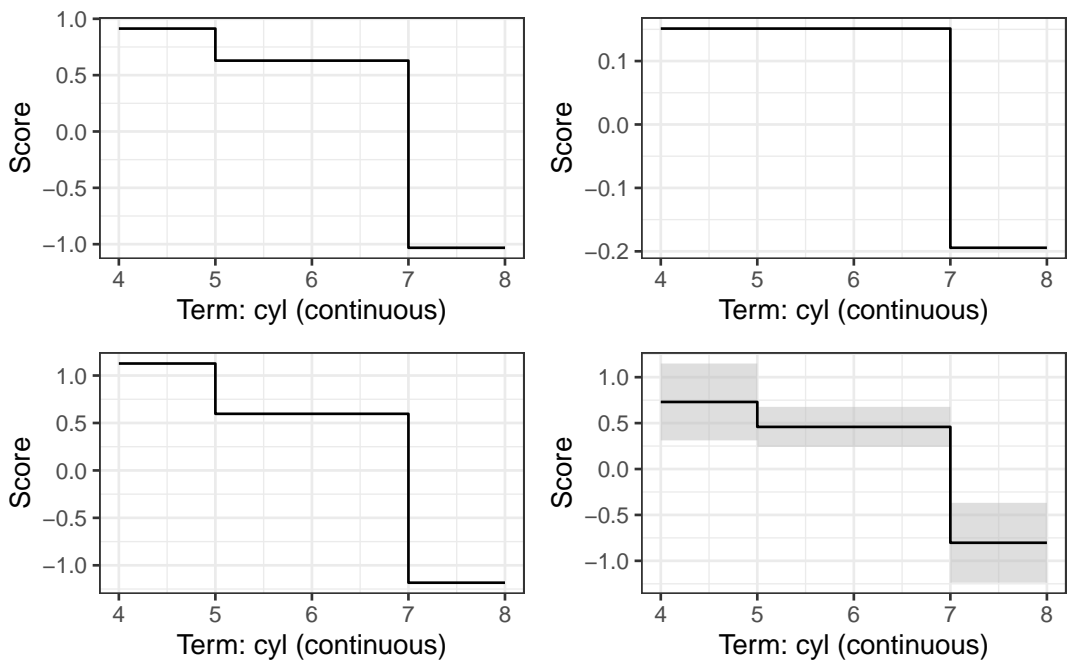


**Figure 12:** Main effect of the number of cylinders. Top left, top right, and bottom left correspond to the main effect from EBMs with different random seeds and no bagging. Bottom right corresponds to the main effect from the merged EBM.

**Tuning strategies**

Recommended tuning strategies for EBMs are discussed in the **interpret** library's FAQ: https://interpret.ml/docs/faq.html. We outline the essential points here. Similar to random forests, the default hyperparameters for EBMs tend to perform reasonably well on most problems; hence, we mostly stuck with the defaults for the examples in this paper. A reasonable strategy is to train an initial EBM model using the defaults and looking through the learned shape functions to catch unexpected or abnormal behavior—oftentimes, these graphs can help indicate which parameters should be tuned. Here are some general recommendations from the **interpret** developers:

- For accuracy, it's generally recommend to set `outer_bags` and `inner_bags` to 25 or more each. This will significantly slow down the training time of the algorithm, and may be unfeasible on larger data sets, but tends to produce smoother graphs with marginally higher accuracy. Note that the algorithm is parallelized at the outer bags level, so typically you'll pay a steeper price in terms of computing time for larger values of the `inner_bags` parameter.

- If you believe the model might be overfitting (e.g., large difference between train and test error, or high degrees of instability in the graphs), consider reducing `max_bins` (the maximum number of bins continuous features are placed in) for smaller data sets (to clump more data together), and take a more aggressive approach to early stopping by decreasing `early_stopping_rounds` and increasing `early_stopping_tolerance`. Conversely, if you believe the model is underfitting, you can do the opposite of the previous suggestions (it might also be helpful to increase `max_rounds`, the total number of boosting rounds).

- If several pairwise interaction effects seem relatively important (i.e., are appearing higher up on the term importance list/plot), then try increasing the maximum number of allowed interactions via the `interaction` argument.

- For general tuning, it's recommended to sweep through `max_bins` with values in the range 32–1024, and `max_leaves` from 2–5. The potential improvements are typically marginal, but may help significantly on some data sets.

Detailed discussion on each hyperparameter available for tuning can be found here: https://interpret.ml/docs/hyperparameters.html.

## 3 Summary

EBMs are glassbox models that can provide both competitive accuracy and transparency through exact explainability. Static and interactive plots can be used to graphically examine a fitted EBM model at both the global and local level. For instance, a model can be explained on the global level by visualizing term importance scores, main effects, and important pairwise interactions.

In this paper, we showed how to fit EBMs using the **ebm** package, which provides a direct interface to Python's **interpret** library. The **ebm** package provides nearly all the functionality of the corresponding Python library, which is not the case for the associated **interpret** package in R. For instance, the provided examples showed how to fit both classification and regression models (including a model for counts using Poisson deviance). This article also showed how to explain an EBM's output at both the global and local level using either static or interactive plots. Further code examples also demonstrated how to use some of the more advanced functionality, like parallelism, merging several EBM models into one, and model compressing (i.e., feature/term reduction) by post-processing an EBM model with the LASSO. Further improvements to the **ebm** package can also be made by providing explicit support for converting fitted EBM models to ONNX (Bai et al., 2025) using the wonderful **ebm2onnx** package (Picard and Aouini, 2025), and model editing via GAM changer (Wang et al., 2022).

## 4 Acknowledgments

## Bibliography

B. Auguie. *gridExtra: Miscellaneous Functions for "Grid" Graphics*, 2017. URL https://CRAN.R-project.org/package=gridExtra. R package version 2.3. [p14]

J. Bai, F. Lu, K. Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2025. URL https://onnx.ai/. [p20]

L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. URL https://doi.org/10.1023/A:1010933404324. [p2]

T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen, R. Mitchell, I. Cano, T. Zhou, M. Li, J. Xie, M. Lin, Y. Geng, Y. Li, and J. Yuan. *xgboost: Extreme Gradient Boosting*, 2024. URL https://CRAN.R-project.org/package=xgboost. R package version 1.7.7.1. [p7]

P. K. Dunn and G. K. Smyth. *Generalized Linear Models With Examples in R*. Springer Texts in Statistics. Springer New York, 2018. ISBN 9781441901187. URL https://books.google.com/books?id=tBh5DwAAQBAJ. [p1]

B. Efron and T. Hastie. *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. Institute of Mathematical Statistics Monographs. Cambridge University Press, 2016. [p12]

C. Elkan. Magical thinking in data mining: lessons from coil challenge 2000. KDD '01, pages 426–431, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 158113391X. doi: 10.1145/502512.502576. URL https://doi.org/10.1145/502512.502576. [p6]

J. Friedman, R. Tibshirani, and T. Hastie. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010. doi: 10.18637/jss.v033.i01. [p15]

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

J. H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002. ISSN 0167-9473. doi: https://doi.org/10.1016/S0167-9473(01)00065-2. URL https://www.sciencedirect.com/science/article/pii/S0167947301000652. Nonlinear Methods and Data Mining. [p3]

J. H. Friedman, B. E. Popescu, et al. Importance sampled learning ensembles. *Journal of Machine Learning Research*, 94305:1–32, 2003. [p15]

B. M. Greenwell. *Tree-Based Methods for Statistical Learning in R*. Chapman & Hall/CRC Data Science Series. CRC Press, 2022. ISBN 9781000595338. URL https://books.google.com/books?id=SpRwEAAAQBAJ. [p2]

B. M. Greenwell. *ebm: Fit Interpretable Machine Learning Models*, 2025. URL https://github.com/bgreenwell/ebm, https://bgreenwell.github.io/ebm/. R package version 0.1.0. [p3]

B. M. Greenwell, A. Dahlmann, and S. Dhoble. Explainable boosting machines with sparsity – maintaining explainability in high-dimensional settings, 2023. URL https://arxiv.org/abs/2311.07452. [p15, 17]

F. E. Harrell Jr. *rms: Regression Modeling Strategies*, 2023. URL https://CRAN.R-project.org/package=rms. R package version 6.7-1. [p6]

C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi: 10.1038/s41586-020-2649-2. [p7]

T. Hastie and R. Tibshirani. *Generalized Additive Models*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1990. ISBN 9780412343902. URL https://books.google.com/books?id=qa29r1Ze1coC. [p1]

G. James, D. Witten, T. Hastie, and R. Tibshirani. *ISLR2: Introduction to Statistical Learning, Second Edition*, 2022. URL https://CRAN.R-project.org/package=ISLR2. R package version 1.3-2. [p18]

S. Jenkins, H. Nori, P. Koch, and R. Caruana. *interpret: Fit Interpretable Machine Learning Models*, 2024. URL https://CRAN.R-project.org/package=interpret. R package version 0.1.34. [p3]

A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis. kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9):1–20, 2004. doi: 10.18637/jss.v011.i09. [p4]

R. Küffner, N. Zach, R. Norel, J. Hawe, D. Schoenfeld, L. Wang, G. Li, L. Fang, L. Mackey, O. Hardiman, M. Cudkowicz, A. Sherman, G. Ertaylan, M. Grosse-Wentrup, T. Hothorn, J. van Ligtenberg, J. H. Macke, T. Meyer, B. Schölkopf, L. Tran, R. Vaughan, G. Stolovitzky, and M. L. Leitner. Crowdsourced analysis of clinical trial data to predict amyotrophic lateral sclerosis progression. *Nature Biotechnology*, 33(1):51–57, Jan 2015. ISSN 1546-1696. doi: 10.1038/nbt.3051. URL https://doi.org/10.1038/nbt.3051. [p12]

Y. Lou, R. Caruana, J. Gehrke, and G. Hooker. Accurate intelligible models with pairwise interactions. KDD '13, pages 623–631, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321747. doi: 10.1145/2487575.2487579. URL https://doi.org/10.1145/2487575.2487579. [p2]

S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf. [p12]

P. McCullagh and J. Nelder. *Generalized Linear Models, Second Edition*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1989. ISBN 9780412317606. URL https://books.google.com/books?id=h9kFH2_FfBkC. [p1]

J. A. Nelder and R. W. M. Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society. Series A (General)*, 135:370–384, 1972. ISSN 00359238, 23972327. doi: 10.2307/2344614. URL https://doi.org/10.2307/2344614. [p1]

H. Nori, S. Jenkins, P. Koch, and R. Caruana. Interpretml: A unified framework for machine learning interpretability, 2019. URL https://arxiv.org/abs/1909.09223. [p2, 3]

T. L. Pedersen. *patchwork: The Composer of Plots*, 2024. URL https://CRAN.R-project.org/package=patchwork. R package version 1.2.0. [p8]

R. Picard and Z. Aouini. *ebm2onnx: A Python package to convert EBM models to ONNX format*, 2025. URL https://github.com/microsoft/ebm2onnx. ebm2onnx package version 3.3.0. [p20]

Plotly Technologies Inc. Collaborative data science, 2015. URL https://plot.ly. [p9]

Pyenv project team. *pyenv: Simple Python version management.*, 2025. URL https://github.com/pyenv/pyenv. pyenv package version 2.5.3. [p3]

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2023. URL https://www.R-project.org/. [p18]

M. T. Ribeiro, S. Singh, and C. Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144, 2016. [p12]

G. Ridgeway. *gbm: Generalized Boosted Regression Models*, 2024. URL https://CRAN.R-project.org/package=gbm. R package version 2.1.9. [p12]

D. Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer, New York, 2008. ISBN 978-0-387-75968-5. URL http://lmdvr.r-forge.r-project.org. [p8]

J. K. Tay, B. Narasimhan, and T. Hastie. Elastic net regularization paths for all generalized linear models. *Journal of Statistical Software*, 106(1):1–31, 2023. doi: 10.18637/jss.v106.i01. [p15]

The Python Packaging Authority. *pip: The PyPA recommended tool for installing Python packages.*, 2025. URL https://pip.pypa.io/. pip package version 25.0.1. [p3]

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246. URL http://www.jstor.org/stable/2346178. [p15]

K. Ushey, J. Allaire, and Y. Tang. *reticulate: Interface to 'Python'*, 2024. URL https://CRAN.R-project.org/package=reticulate. R package version 1.35.0. [p3]

P. van der Putten and M. van Someren. A bias-variance analysis of a real world learning problem: The coil challenge 2000. *Machine Learning*, 57(1):177–195, Oct 2004. ISSN 1573-0565. doi: 10.1023/B:MACH.0000035476.95130.99. URL https://doi.org/10.1023/B:MACH.0000035476.95130.99. [p4]

G. Varoquaux, A. Gramfort, V. Michel, and B. Thirion. *joblib: running Python functions as pipeline jobs*, 2024. Python package version 1.4.2. [p18]

Z. J. Wang, A. Kale, H. Nori, P. Stella, M. E. Nunnally, D. H. Chau, M. Vorvoreanu, J. W. Vaughan, and R. Caruana. Interpretability, Then What? Editing Machine Learning Models to Reflect Human Knowledge and Values. In *Proceedings of the 28th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2022. URL https://interpret.ml/gam-changer. [p2, 3, 20]

F. Wick, U. Kerzel, and M. Feindt. Cyclic boosting - an explainable supervised machine learning algorithm. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 358–363. IEEE, Dec. 2019. doi: 10.1109/icmla.2019.00067. URL http://dx.doi.org/10.1109/ICMLA.2019.00067. [p2]

H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. URL https://ggplot2.tidyverse.org. [p8]

S. N. Wood. *Generalized Additive Models: An Introduction with R, Second Edition*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press, 2017. ISBN 9781498728348. URL https://books.google.com/books?id=HL-PDwAAQBAJ. [p2]

*Brandon M. Greenwell*
*University of Cincinnati*
*Department of Operations, Business Analytics, and Information Systems*
*Cincinnati, Ohio*
https://github.com/bgreenwell
*ORCiD:* *0000-0002-8120-0084*
greenwb@ucmail.uc.edu