The Stata Journal

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press, and Stata is a registered trademark of StataCorp LP.

The Stata Journal (2005)
5, Number 3, pp. 330–354

Boosted regression (boosting): An introductory tutorial and a Stata plugin

Matthias Schonlau
RAND

Abstract. Boosting, or boosted regression, is a recent data-mining technique that has shown considerable success in predictive accuracy. This article gives an overview of boosting and introduces a new Stata command, `boost`, that implements the boosting algorithm described in Hastie, Tibshirani, and Friedman (2001, 322). The plugin is illustrated with a Gaussian and a logistic regression example. In the Gaussian regression example, the R^2 value computed on a test dataset is $R^2 = 21.3\%$ for linear regression and $R^2 = 93.8\%$ for boosting. In the logistic regression example, stepwise logistic regression correctly classifies 54.1% of the observations in a test dataset versus 76.0% for boosted logistic regression. Currently, `boost` accommodates Gaussian (normal), logistic, and Poisson boosted regression. `boost` is implemented as a Windows C++ plugin.

Keywords: st0087, boost, boosted regression, boosting, data mining

1 Introduction

Economists and analysts in the data-mining community differ in their approach to regression analysis. Economists often build a model from theory and then use the data to estimate parameters of their model. Because their model is justified by theory, economists are sometimes less inclined to test how well the model fits the data. Data miners tend to favor the "kitchen sink" approach in which all or most available regressors are used in the regression model. Because the choice of x-variables is not supported by theory, validation of the regression model is very important. The standard approach to validating models in data mining is to split the data into a training and a test dataset.

The concept of a training dataset versus a test dataset is central to many data-mining algorithms. The model is fitted on the training data. The fitted model is then used to make predictions on the test data. Assessing the model on test data rather on the training data ensures that the model is not overfitted and is generalizable. If the regression model has tuning parameters (e.g., ridge regression, neural networks, boosting), good values for the tuning parameters are usually found by running the model several times with different values for the tuning parameters. The performance of each model is assessed on the test dataset, and the best model (according to some criterion) is chosen.

In this paper, I review boosting or boosted regression and supply a Stata plugin for Windows. In the same way that generalized linear models include Gaussian, logistic, and other regressions, boosting also includes boosted versions of Gaussian, logistic, and other regressions. Boosting is a highly flexible regression method. It allows

the researcher to specify the x-variables without specifying the functional relationship to the response. Traditionally, data miners have used boosting in the context of the "kitchen sink" approach to regression but it is also possible to use boosting in a more-targeted manner, i.e., using only variables motivated by theory. Because it is more flexible, a boosted model will tend to fit better than a linear model and therefore inferences made based on the model may have more credibility. There is mounting empirical evidence that boosting is one of the best modeling approaches ever developed. Bauer and Kohavi (1999) performed an extensive comparison of boosting with several other competitors on 14 datasets and found boosting to be "the best algorithm". Friedman, Hastie, and Tibshirani (2000) compare several boosting variants with the CART (classification and regression tree) method and find that all the boosting variants outperform the CART algorithm on eight datasets. The success of boosting in terms of predictive accuracy has been subject to much speculation. Part of the mystery that surrounds boosting is due to the fact that different scientific communities, computer scientists and statisticians, have contributed to its development. I will first give a glimpse into the fascinating history of boosting. The remainder of the paper is structured as follows: section 2 contains the syntax of the `boost` command; section 3 contains options for that command; section 4 explains boosting and gives a historical overview; section 5 uses a toy example to show how the boost command can be used for normal (Gaussian) regression; section 6 features a logistic regression example; section 7 gives runtime benchmarks for the boosting command; and section 8 concludes with some discussion.

2 The boost command

2.1 Installation

To install the `boost` plugin, copy the `boost.hlp` and `boost.ado` files into one of the ado directories, e.g., `c:\ado\personal`. A list of valid directories can be obtained by typing `adopath` within Stata. Copy the `boost.dll` file into a directory of your choosing (e.g., the same directory as the ado-file). Unlike an ado-file, a plugin must be explicitly loaded:

```
capture program boost_plugin, plugin using("C:\ado\personal\boost.dll")
```

The `capture` command prevents this line from resulting in an error if the plugin was already loaded.

2.2 Syntax

`boost` *varlist* [*if*] [*in*], distribution(*string*) maxiter(#) [influence predict(*varname*) shrink(#) bag(#) trainfraction(#) interaction(#) seed(#)]

2.3 Description

`boost` determines the number of iterations that maximizes the likelihood or, equivalently, the pseudo-R^2. The pseudo-R^2 is defined as $R^2 = 1 - L1/L0$, where $L1$ and $L0$ are the log likelihood of the full model and intercept-only model, respectively.

Unlike the R^2 given in `regress`, the pseudo-R^2 is an out-of-sample statistic. Out-of-sample R^2s tend to be lower than in-sample-R^2s.

Output and return values

The standard output consists of the best number of iterations, *bestiter*; the R-squared value computed on the test dataset, `test_R2`; and the number of observations used for the training data, `trainn`. `trainn` is computed as the number of observations that meet the `in`/`if` conditions times `trainfraction()`. These statistics can also be retrieved using `ereturn`. In addition, `ereturn` also stores the training R-squared value, `train_R2`, as well as the log-likelihood values from which `train_R2` and `test_R2` are computed.

2.4 Options

`distribution(`*string*`)` specifies the distribution of the inefficiency term. Possible distributions are `normal`, `logistic`, and `poisson`.

`maxiter(#)` specifies the maximum number of trees to be fitted. The actual number used, *bestiter*, can be obtained from the output as `e(bestiter)`. When *bestiter* is too close to `maxiter()`, the maximum likelihood iteration may be larger than `maxiter()`. In that case, it is useful to rerun the model with a larger value for `maxiter()`. When `trainfraction(1.0)`, all `maxiter()` observations are used for prediction (*bestiter* is missing because it is computed on a test dataset).

`influence` displays the percentage of variation explained (for nonnormal distributions, the percentage of log likelihood explained) by each input variable. The influence matrix is saved in `e(influence)`.

`predict(`*varname*`)` predicts and saves the predictions in the variable *varname*. To allow for out-of-sample predictions, `predict()` ignores `if` and `in`. For model fitting only, observations that satisfy `if` and `in` are used, and predictions are made for all observations.

`shrink(#)` specifies the shrinkage factor. `shrink(1)` corresponds to no shrinkage. As a general rule of thumb, reducing the value for `shrink()` requires increasing the value of `maxiter()` to achieve a comparable cross-validation R^2. The default is `shrink(0.01)`.

`bag(#)` specifies the fraction of training observations that is used to fit an individual tree. `bag(0.5)` means that half the observations are used for building each tree. To use all observations, specify `bag(1.0)`. The default is `bag(0.5)`.

`trainfraction(#)` specifies the percentage of data to be used as training data. The remainder, the test data, is used to evaluate the best number of iterations. The default is `trainfraction(0.8)`.

`interaction(#)` specifies the maximum number of interactions allowed. For example, `interaction(1)` means that only main effects are fitted; `interaction(2)` means that main effects and two-way interactions are fitted; and so forth. The number of interactions equals the number of terminal nodes in a tree plus 1. `interaction(1)` means that each tree has 2 terminal nodes; `interaction(2)` means that each tree has 3 terminal nodes; and so forth. The default is `interaction(5)`.

`seed(#)` specifies the random-number seed to generate the same sequence of random numbers. Random numbers are only used for bagging. Bagging uses random numbers to select a random subset of the observations for each iteration. The default is `seed(0)`. The `boost seed()` option is unrelated to Stata's `set seed` command.

3 Boosting

Boosting was invented by computational learning theorists and later reinterpreted and generalized by statisticians and machine learning researchers. Computer scientists tend to think of boosting as an "ensemble" method (a weighted average of predictions of individual classifiers), whereas statisticians tend to think of boosting as a sequential regression method. To understand why statisticians and computer scientists think about the essentially same algorithms in different ways, both approaches are discussed: section 4.1 discusses an early boosting algorithm from computer science; section 4.2 describes regression trees, the most-commonly used base learner in boosting; and section 4.3 describes Friedman's gradient boosting algorithm, which is the algorithm I have implemented for the Stata plugin. The remaining sections talk about variations of the algorithm that are relevant to my implementation (section 4.4), how to evaluate boosting algorithms via a cross-validated R^2 (section 4.5), the influence of variables (section 4.6), and advice on how to set the boosting parameters in practice (section 4.7).

3.1 Boosting and its roots in computer science

Boosting was invented by two computer scientists at AT&T Labs (Freund and Schapire 1997). Below I describe an early algorithm, the "AdaBoost" algorithm, because it illustrates why computer scientists think of boosting as an ensemble method, that is, a method that averages over multiple classifiers.

AdaBoost (see algorithm 1) works only in the case where the response variable takes only one of two values. Whether the values are 0,1 or -1,1 is not important—the algorithm could be modified easily. Let C_1 be a binary classifier (e.g., logistic regression) that predicts whether an observation belongs to the class -1 or 1. The classifier is fitted to the data as usual, and the misclassification rate is computed. This first classifier C_1 receives a classifier weight that is a monotone function of the error

rate it attains. In addition to classifier weights, there are also observation weights. For the first classifier, all observations were weighted equally. The second classifier, C_2 (e.g., the second logistic regression), is fitted to the same data, however, with changed observation weights. Observation weights corresponding to observations misclassified by the previous classifier are increased. Again, observations are reweighted, a third classifier C_3 (e.g., a third logistic regression) is fitted, and so forth. Altogether *iter* classifiers are fitted where *iter* is some predetermined constant. Finally, using the classifier weights the classifications of the individual classifiers are combined by taking a weighted majority vote. The algorithm is described in more detail in algorithm 1.

Initialize weights to be equal $w_i = 1/n$.

For $m = 1$ to *iter* classifiers (C_m):

(a) Fit classifier C_m to the weighted data.

(b) Compute the (weighted) misclassification rate r_m.

(c) Let the classifier weight $\alpha_m = \log\{(1 - r_m)/r_m\}$.

(d) Recalculate weights $w_i = w_i \exp\{\alpha_m \mathbf{I}(y_i \neq C_m)\}$.

Majority vote classification: $\text{sign}\left\{ \sum_{m=1}^{M} \alpha_m C_m(x) \right\}$.

Algorithm 1: The AdaBoost algorithm for classification into two categories.

Early on, researchers attributed the success and innovative element of this algorithm to the fact that observations that are repeatedly misclassified are given successively larger weights. Another unusual element is that the final "boosted" classifier consists of a majority-weighted vote of all previous classifiers. With this algorithm, the individual classifiers do not need to be particularly complex. On the contrary, simple classifiers tend to work best.

3.2 Regression trees

The most-commonly used simple classifier is a regression tree (e.g., CART, Breiman et al. 1984). A regression tree partitions the space of input variables into rectangles and then fits a constant (e.g., estimated by an average or a percentage) to each rectangle. The partitions can be described by a series of if-then statements, or they can be visualized by a graph that looks like a tree. Figure 1 gives an example of such a tree using age and indicator variables for race/ethnicity. The tree has six splits and, therefore, seven leaves (terminal nodes). Each split represents an if-then condition. Each observation descends the set of if-then conditions until a leaf is reached. If the if-then condition is true, the observation goes to the right branch; otherwise, it goes to the left branch. For example, in figure 2, any observation over 61 years of age would be classified in

the right-most leaf regardless of the person's race/ethnicity. The leaf with the largest values of the response, 0.8, corresponds to observations between 49 and 57 years of age who are neither Hispanic nor black. As the splits on age show, the tree can split on the same variable several times. If the six splits in the tree in figure 1 had split on six different variables, the tree would represent a 6-level interaction because six variables would have to be considered jointly in order to obtain the predicted value.

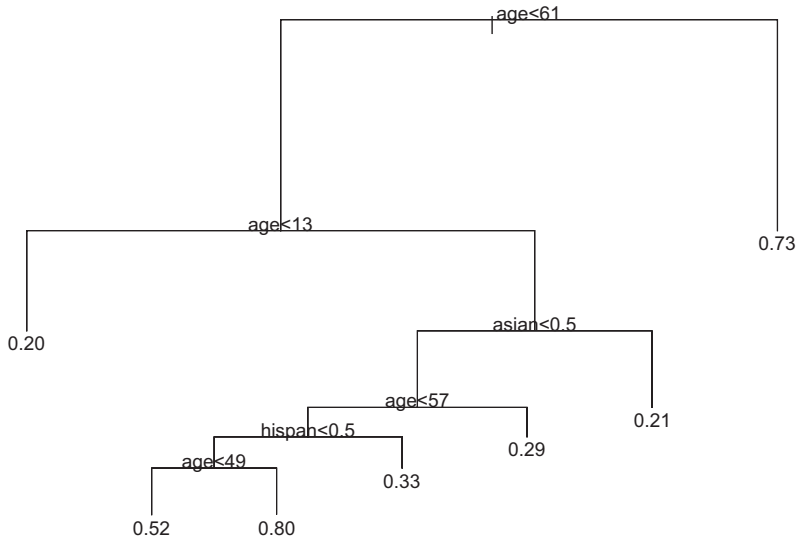

Figure 1: Example of a regression tree with six splits

A regression tree with only two terminal nodes (i.e., a tree with only one split) is called a tree stump. It is hard to imagine a simpler classifier than a tree stump—yet tree stumps work surprisingly well in boosting. Boosting with stumps fits an additive model, and many datasets are well approximated with only additive effects. Fewer complex classifiers can be more flexible and harbor a greater danger of overfitting (i.e., fitting well only to the training data).

3.3 Friedman's gradient boosting algorithm

Early researchers had some difficulty explaining the success of the AdaBoost algorithm. The computational-learning-theory roots of the AdaBoost puzzled statisticians who have traditionally worked from likelihood-based approaches to classification and, more gener-

ally, to regression. Then Friedman, Hastie, and Tibshirani (2000) were able to reinterpret this algorithm in a likelihood framework, enabling the authors to form a boosted logistic regression algorithm, a formulation more familiar to the statistical community. Once the connection to the likelihood existed, boosting could be extended to generalized linear models and further still to practically any loss criterion (Friedman 2001; Ridgeway 1999). This meant that a boosting algorithm could be developed for all the error distributions in common practice: Gaussian, logistic, Poisson, Cox models, etc. With the publication of Hastie, Tibshirani, and Friedman's (2001) book on statistical learning, modern regression methods such as boosting caught on in the statistics community.

The interpretation of boosting in terms of regression for a continuous, normally distributed response variable is as follows: The average y-value is used as a first guess for predicting all observations. This is analogous to fitting a linear regression model that consists of the intercept only. The residuals from the model are computed. A regression tree is fitted to the residuals. For each terminal node, the average y-value of the residuals that the node contains is computed. The regression tree is used to predict the residuals. (In the first step, this means that a regression tree is fitted to the difference between the observation and the average y-value. The tree then predicts those differences.) The boosting regression model—consisting of the sum of all previous regression trees—is updated to reflect the current regression tree. The residuals are updated to reflect the changes in the boosting regression model; a tree is fitted to the new residuals, and so forth. This algorithm is summarized in more detail in algorithm 2.

---

1) Initialization: Set initial guess to $\overline{y}$.

2) For all regression trees $m = 1$ to M:

   2a) Compute the residuals based on the current model

   $$r_{mi} = y_i f_{m-1}(x_i)$$

   where i indexes observations.
   Note that f_{m-1} refers to the sum of all previous regression trees.

   2b) Fit a regression tree (with a fixed number of nodes) to the residuals.

   2c) For each terminal node of the tree, compute the average residual.
   The average value is the estimate for residuals that fall in the corresponding node.

   2d) Add the regression tree of the residuals to the current best fit,
   $f_m = f_{m-1} +$ last regression tree of residuals.

Algorithm 2: Friedman's gradient boosting algorithm for a normally distributed response

Each term of the regression model thus consists of a tree. Each tree fits the residuals of the prediction of all previous trees combined. To generalize algorithm 2 to a more

general version with arbitrary distributions requires that "average y-value" be replaced with a function of y-values that is dictated by the specific distribution and that the residual (step 2a) be a "deviance residual" (McCullagh and Nelder 1989).

The basic boosting algorithm requires the specification of two parameters. One is the number of splits (or the number of nodes) that are used for fitting each regression tree in step 2b of algorithm 2. The number of nodes equals the number of splits plus one. Specifying one split (tree stump) corresponds to an additive model with only main effects. Specifying two splits corresponds to a model with main effects and two-way interactions. Generally, specifying J splits corresponds to a model with up to J-way interactions. When J x-variables need to be considered jointly for a component of a regression model, this is a J-way interaction. Hastie, Tibshirani, and Friedman (2001) suggest that $J = 2$, in general, is not sufficient and that $4 \leq J \leq 8$ generally works well. They further suggest that the model is typically not sensitive to the exact choice of J within that range. In the `boost` program, J is specified as an option, `interaction(`J`)`.

The second parameter is the number of iterations or the number of trees to be fitted. If the number of iterations is too large, the model will overfit; i.e., it will fit the training data well but not generalize to other observations from the same population. If the number of iterations is too small, the model is not fitted as well, either. A suitable value for `maxiter()` can range from a few dozen to several thousand, depending on the value of a shrinkage parameter (explained below) and the dataset. The easiest way to find a suitable number of iterations is to check how well the model fits on a test dataset. In the `boost` program, the maximum number of iterations, `maxiter()`, is specified, and the number of iterations that maximizes the log likelihood on a test dataset, *bestiter*, is automatically found. The size of the test dataset is controlled by `trainfraction()`. For example, if `trainfraction(0.5)`, the last 50% of the data is used as test data.

3.4 Shrinkage and bagging

There are two commonly used variations on Friedman's boosting algorithm: "shrinkage" and "bagging". Shrinkage (or regularization) means reducing or shrinking the impact of each additional tree in an effort to avoid overfitting. The intuition behind this idea is that it is better to improve a model by taking many small steps than a smaller number of large steps. If one step turns out to be a mis-step, the damage can be more easily undone in subsequent steps. Shrinkage has been previously employed, for example, in ridge regression, where it refers to shrinking regression coefficients back to zero to reduce the impact of unstable regression coefficients on the model. Shrinkage is accomplished by introducing a parameter λ in step 2d of algorithm 2:

$$f_m = f_{m-1} + \lambda \cdot (\text{last regression tree of residuals})$$

where $0 < \lambda \leq 1$. The smaller λ, the greater the shrinkage is. The value $\lambda = 1$ corresponds to no shrinkage. Typically λ is 0.1 or smaller, with $\lambda = 0.01$ or $\lambda = 0.001$ being common. Smaller shrinkage values require larger number of iterations. In my experience, λ and the number of iterations typically satisfy $10 \leq \lambda \cdot \textit{bestiter} \leq 100$ for the best model. In other words, a decrease of λ by a factor of 10 implies an increase

of the number of iterations by a similar factor. In the `boost` program, λ is specified through the option `shrink(`λ`)`.

The second commonly used variation on the boosting algorithm is bagging. At each iteration, only a random fraction, `bag()`, of the residuals is selected. In that iteration, only the random subset of the residuals is used to build the tree. Unselected residuals are not used in that iteration at all. The randomization is thought to reduce the variation of the final prediction without affecting bias. Different random subsets may have different idiosyncrasies that will average out. While not all observations are used in each iteration, all observations are eventually used across all iterations. Friedman (2001) recommends bagging with 50% of the data. In the `boost` program, this can be specified as `bag(0.5)`.

3.5 Cross-validation and the pseudo-R^2

Highly flexible models are prone to overfitting. While it may still occur, overfitting is less of an issue in linear regression, where the restriction of linearity guards against this problem to some extent. To assess predictive accuracy with highly flexible models, it is important to separate the data the model was trained on from test data.

My boosting implementation splits the data into a training dataset and a test dataset. By default, the first 80% of the data are used as training data and the remainder are used as test data. This percentage can be changed through the use of the option `trainfraction()`. `trainfraction(0.5)`, for example, uses the first 50% of the data as training data. It is important that the observations are in random order before invoking the `boost` command because otherwise the test data may be different from the training data in a systematic way.

Cross-validation is a generalization of the idea of splitting the data into training and test datasets. For example, in five-fold cross-validation the dataset is split into 5 distinct subsets of 20% of the data. In turn, each subset is used as test data and the remainder as training data. This is illustrated graphically in figure 2. My implementation corresponds to the fifth row of figure 2. An example of how to rotate the 5 groups of data to accomplish five-fold cross-validation using the `boost` implementation is given in the help file for the `boost` command.

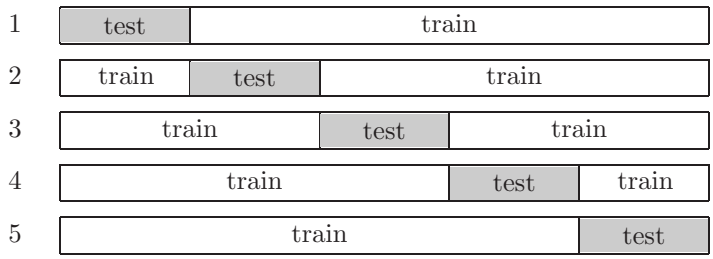

Figure 2: Illustration of five-fold cross-validation: Data are split into training data and test data in five different ways.

For each test dataset, a pseudo-R^2 is computed on the test dataset. The pseudo-R^2 is defined as

$$\text{pseudo}-R^2 = 1 - L1/L0$$

where $L1$ and $L0$ are the log likelihoods of the model under consideration and an intercept-only model, respectively (see Long and Freese 2003 for a discussion on pseudo-R^2). In the case of Gaussian (normal) regression, the pseudo-R^2 turns into the familiar R^2 that can be interpreted as "fraction of variance explained by the model". For Gaussian regression, it is sometimes convenient to compute R^2 as

$$R^2 = \frac{\text{Var}(y) - \text{MSE}(y, \widehat{y})}{\text{Var}(y)} \tag{1}$$

where Var(.) and MSE(.,.) refer to variance and mean squared error, respectively. To avoid cumbersome language, this is referred to as "test R^2" in the remainder of this paper.

The training R^2 and test R^2 are computed on training and test data, respectively. The training R^2 is always between 0 and 1. Usually, the test R^2 is also between 0 and 1. However, if the log likelihood based on the intercept-only model is greater than the one for the model under consideration, the test R^2 is negative. A negative test R^2 is a sign that the model is strongly overfitted. The `boost` command computes both training and test R^2 values and makes them available in `e(test_R2)` and `e(train_R2)`.

3.6 Influence of variables and visualization

For a linear regression model, the effect of x-variables on the response variable is summarized by their respective coefficients. The boosting model is complex but can be interpreted with the right tools. Instead of using regression coefficients, boosting has the concept of "influence of variables" (Friedman 2001). Each split on a variable in a regression tree increases the log likelihood of the regression tree model. In Gaussian regression, the increase in log likelihood is proportional to the increase in sums of squares explained by the model. The sum of log likelihood increasing across all trees due to a given variable yields the influence of that variable. For example, suppose that there

are 1,000 iterations or regression trees and that each regression tree has one split (i.e., a tree stump, `interaction(1)`, main effects only). Further, suppose that 300 of the regression trees split on variable x_i. Then the sum of the increase in log likelihood due to these 300 trees represents the influence of x_i. The influences are standardized such that they add up to 100%.

Because a regression tree does not separate main effects and interactions, influences are only defined for variables—not for individual main effect or interaction terms. Influences only reveal the sum of squares explained by the variables; they say nothing about how the variable affects the response. The functional form of a variable is usually explored through visualization. Visualization of any one (or any set) of the variables is achieved by predicting over a range or grid of these variables. Suppose that the effect of x_i on the response is of interest. There are several options. The easiest is to predict the response for a suitable range of x_i values while holding all other variables constant (for example, at their mean or their mode). A second option is to predict the response for a suitable range of x_i values for each observation in the sample. Then the sample-averaged prediction for a given x_i value can be computed. A third option is to numerically integrate out other variables with a suitable grid of values to get what is usually referred to as a main effect.

In linear regression, the coefficient of a variable needs to be interpreted in the context of its range. It is possible to artificially inflate coefficients by, for example, changing a measurement from kilograms to grams. The influence percentages are invariant to any one-to-one rescaling of the variables. For example, measuring a variable in miles or kilometers does not affect the influence, but it would affect the regression coefficient of a logistic regression. In the `boost` program, the influence of individual variables is displayed when option `influence` is specified. The individual values are stored in a Stata matrix and can be accessed through `e(influence)`. The help file gives an example.

3.7 Advice on setting boosting parameters

In what follows, I give some suggestions of how the user might think about setting parameter values.

- Maxiter: If `e(bestiter)` is almost as large as `maxiter()`, consider rerunning the model with a larger value for `maxiter()`. The iteration corresponding to the maximum likelihood estimate may be greater than `maxiter()`.

- Interactions: Usually a value between 3 and 7 works well. If you know that your model only has two-way interactions, I still suggest trying larger values for two reasons. The first reason is that, typically, the loss in test R^2 for specifying too low a number of interactions is far worse than that of specifying a larger number of interactions. The performance of the model only deteriorates noticeably when the number of interactions is much too large. The second reason is that specifying five-way interactions allows each tree to have five splits. These splits are not

necessarily always used for an interaction (splitting on five different variables in the same tree). They could also be used for a nonlinearity in a single variable (five splits on different values for a single variable) or for combinations of nonlinearities and interactions.

- Shrinkage: Lower shrinkage values usually improve the test R^2 but they increase the running time dramatically. Shrinkage can be thought of as a step size. The smaller the step size, the more iterations and computing time are needed. In practice, I choose a small shrinkage value such that the command execution does not take too much time. Because I am impatient, I usually start out with `shrink(0.1)`. If time permits, I switch to 0.01 or lower in a final run. There are diminishing returns for very low shrinkage values.

- Bagging: Bagging may improve the R^2, and it also makes the estimate less variable. Typically, the exact value is not crucial. I typically use a value in the range of 0.4–0.8.

4 Boosted Gaussian regression

This section gives a simple example with four explanatory variables constructed to illustrate how to perform and evaluate boosted regressions. The results are also compared to linear regression. Linear regression is a good reference model because many scientists initially fit a linear model. I simulated data from the following model

$$y = 30(x_1 - 0.5)^2 + 2x_2^{-0.5} + x_3 + \epsilon$$

where ϵ~uniform$(0, 1)$ and $0 \leq x_i \leq 1$ for $i \in 1, 2, 3$. To keep things relatively simple, the model has been chosen to be additive without interactions. It is quadratic in `x1`, nonlinear in `x2`, and linear with a small slope in `x3`. The nonlinear contribution of `x2` is stronger than the linear contribution of `x3`, even though their slopes are similar. A fourth variable, `x4`, is unrelated to the response but is used in the analysis in an attempt to confuse the boosting algorithm. Scatter plots of `y` versus `x1` through `x4` are shown in figure 3.

(*Continued on next page*)

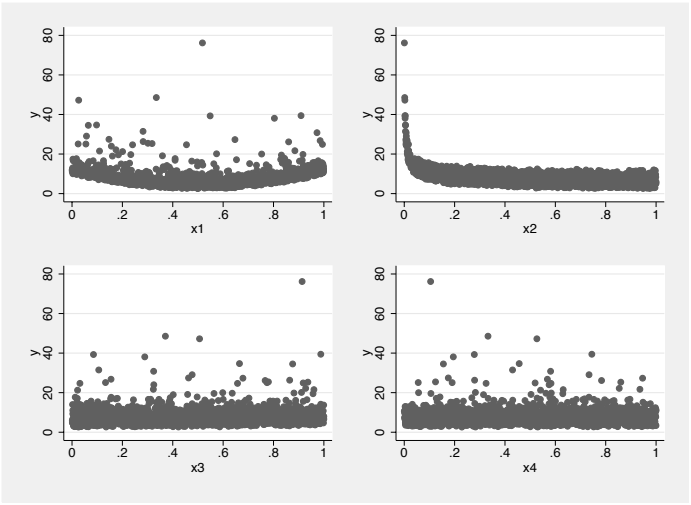

Figure 3: Scatter plots of `y` versus `x1` through `x4`

I chose the parameter values: `shrink(0.01)`, `bag(0.5)`, and `maxiter(4000)`. My rule of thumb in choosing the maximal number of iterations is that the shrinkage factor times the maximal number of iterations should be roughly between 10 and 100. In my experience, the cross-validated R^2 as a function of the number of iterations is unimodal; i.e., there is only one maximum. If *bestiter* is too close to `maxiter()`, the number of iterations that maximizes the likelihood may be greater than `maxiter()`. It is recommended that you rerun the command with a larger value for `maxiter()`. The command I am giving is

```
. boost y x1-x4, distribution(normal) trainfraction(0.5) maxiter(4000) seed(1)
> bag(0.5) interaction(‘inter’) shrink(0.01)
```

where the commands only differ by *inter* ranging from 1 through 5. One of these `boost` commands runs in 8.8 seconds on my laptop. Fixing the seed is only relevant for bagging. Figure 4 shows a plot of the test R^2 versus the number of interactions. The test R^2 is roughly the same, regardless of the number of interactions (note the scale of the plot). The fact that the test R^2 is high even for the main-effect model (`interaction(1)`) does not surprise us because our model did not contain any interactions. The actual number of iterations that maximizes the likelihood, *bestiter*, varies. Here the number of iterations is (number of interactions in parenthesis): 3769 (1), 2171 (2), 2401 (3), 1659 (4), 1156 (5).

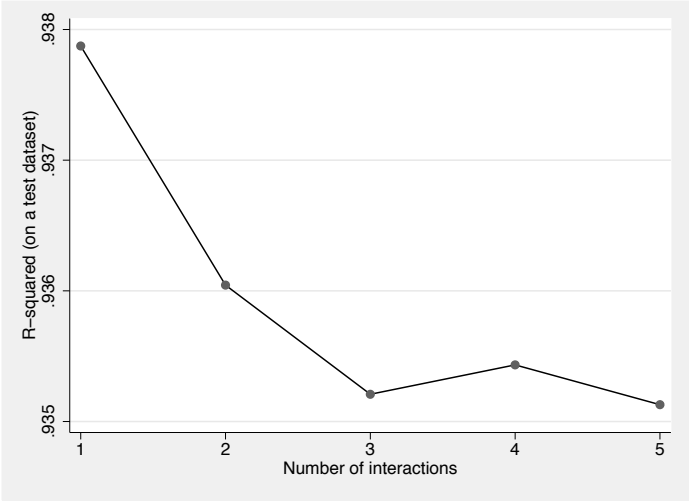

Figure 4: Scatter plot of the R^2 (computed on a test dataset) versus the number of interactions. Note the scale on the vertical axis.

I often want to confirm that this model indeed works better than linear regression. Directly comparing the R^2 value for the boosted regression and the linear regression is not a fair comparison. The boosted regression R^2 is computed on a test dataset, whereas the linear regression R^2 is computed on the training dataset. Using (1), it is possible to compute an R^2 on a test dataset for the linear regression. For the following set of Stata commands, I assume that the first `trainn` observations in the dataset constitute the training data, and the remainder are the test data. The predictions from the linear regression (or any other predictions) are denoted `regress_pred`.

```
global trainn=e(trainn)   /* using  e() from  boost */
regress y x1 x2 x3 x4 in 1/$trainn
predict regress_pred
* compute Rsquared on test data
generate regress_eps=y-regress_pred
generate regress_eps2= regress_eps*regress_eps
replace regress_eps2=0 if _n<=$trainn
generate regress_ss=sum(regress_eps2)
local mse=regress_ss[_N] / (_N-$trainn)
summarize y if _n>$trainn
local var=r(Var)
local regress_r2= (‘var’-‘mse’)/‘var’
display "mse=" ‘mse’ " var=" ‘var’  " regress r2="  ‘regress_r2’
```

The test R^2 will usually be lower than the R^2 in the output that Stata displays—but because of variability, it may be larger on occasion. From the Stata output of `regress`, the (training) R^2 value is $R^2 = 24.1\%$. The test R^2 computed from the above set of Stata commands is $R^2 = 21.3\%$. I compute the boosting predictions and the influences of the variables:

```
. boost y x1 x2 x3 x4, distribution(normal) trainfraction(0.5) bag(0.5)
> maxiter(4000) interaction(1) shrink(0.05) predict("bookst_pred") influence
```

If we substitute the boosting predictions for the linear regression predictions in the above set of Stata commands, the test boosting R^2 turns out to be $R^2 = 93.8\%$. This is the same value as the test R^2 displayed in figure 4. Figure 5 displays actual versus fitted y-values for both linear regression and boosting. The boosting model fits much better.

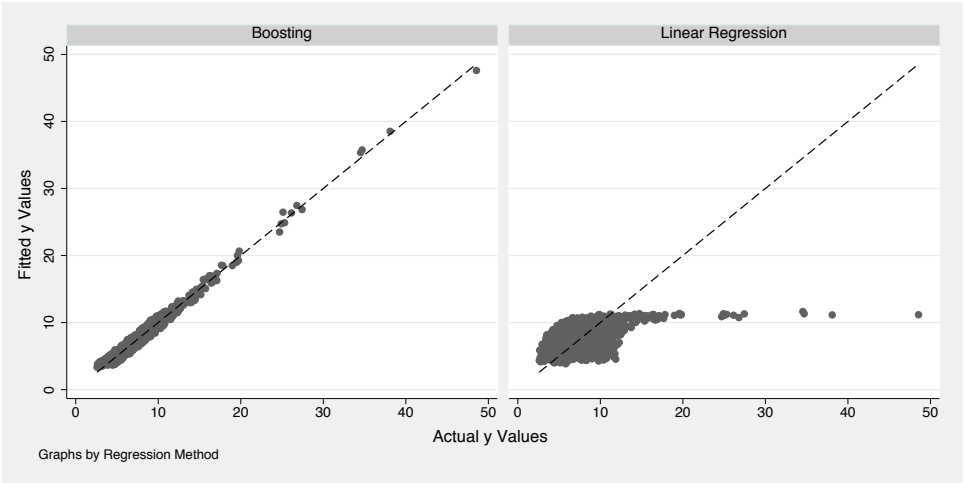

Figure 5: A calibration plot for the linear regression example: fitted versus actual values for both linear regression and boosting. The boosting model is much better calibrated.

When working with linear regressions, it is common to look at coefficients to assess how a variable affects the outcome. In boosted regression, one looks at the influence of all variables. The influences are given in percentages. Specifying influence in the Stata command listed above gives the following output:

Table 1: Influence of each variable (percent)

x1	30.9%
x2	68.3%
x3	0.67%
x4	0.08%

Variables x2 and x1 are most influential. The other variables have almost no influence. Given that there is a weak relationship between y and x3 and no relationship between y and x4, it is nice to see that the influence of x3 is larger than that of x4.

The influence shows one can learn how large the effect of individual variables is but not the functional form. To help you visualize the conditional effect of a variable, `x1`, all variables except `x1` are set to a fixed value (here 0.5). For x_i, values that cover its range are chosen. The new observations are fed to the model for predicting the response, and the predicted response is plotted against `x1`. This can be accomplished as follows:

```
drop if _n>1000
set obs 1400
replace x1=0.5 if _n>1000
replace x2=0.5 if _n>1000
replace x3=0.5 if _n>1000
replace x4=0.5 if _n>1000
replace x1= (_n-1000)/100 if _n>1000 & _n<=1100
replace x2= (_n-1100)/100 if _n>1100 & _n<=1200
replace x3= (_n-1200)/100 if _n>1200 & _n<=1300
replace x4= (_n-1300)/100 if _n>1300 & _n<=1400
boost y x1 x2 x3 x4 in 1/1000 , distribution(normal) /*
*/ maxiter(4000) bag(0.5)  interaction(1) shrink(0.01) /*
*/ pred("pred")
line pred x1 if _n>1000 & _n<=1100
line pred x2 if _n>1100 & _n<=1200
line pred x3 if _n>1200 & _n<=1300
line pred x4 if _n>1300 & _n<=1400
```

The `boost` command uses the first 1,000 observations to fit the model but uses all observations for prediction. Figure 6 displays all four conditional effects. All effects are step functions because the base learners, regression trees, can only produce step functions. The features of the smooth curves are well reproduced.

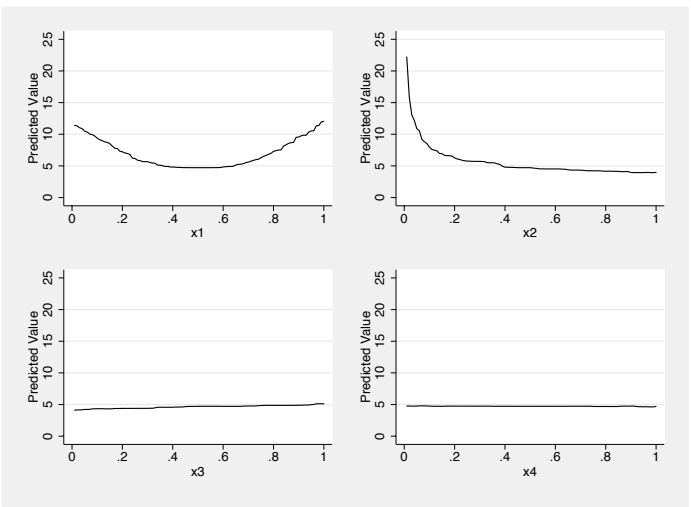

Figure 6: Conditional plots: predictions for `x1` through `x4` while other variables are held constant at $x_i = 0.5$.

5 Example: Boosted logistic regression

I compare boosted logistic regression with a regular logistic regression model. I simulate data from the following model

$$\log\left(\frac{p}{1-p}\right) = -35 + \sum_{j=1}^{10} \frac{1}{0.1 + \sum_{i=1}^{3}(x_i - j/10)^2} - 100\mathbf{I}(x_4 > 0.95) + \epsilon \qquad (2)$$

where the indicator function $\mathbf{I}(arg)$ equals one if its argument is true and zero otherwise, ϵ~uniform$(0, 22.985)$, and $0 \leq x_i \leq 1$ for $i = 1, 2, 3$. The value 22.985 corresponds roughly to one standard deviation of $\log\{p/(1-p)\}$; i.e., the signal to noise ratio on the logit scale is 1. This model has a nonlinear 3-level interaction between `x1`, `x2`, and `x3` and a nonlinearity in the form of a step function for `x4`. I simulate 46 additional variables, `x5` through `x50`, uniformly distributed across their support $0 \leq x_i \leq 1$ for $i = 4, 5, \ldots, 50$. All x-variables are uncorrelated to one another. The response `y` is a function of only the first four of the 50 variables. Of course, boosting still performs well when the variables are correlated. Boosting can also be used when there are more covariates than observations.

The data consist of the response `y`, 50 x-variables, and 4,000 observations. Half of these observations are used as training data, and half are test data. I fit a regular, linear logistic regression model to the data. The odds ratios of the first 4 variables are shown in table 2. The odds ratios were obtained by running the command `logistic y x1-x50`. Except for `x4`, none of the variables is significant. As expected, by chance, a few of the remaining 46 variables were also significant at the 5% level.

Table 2: Odds ratios of the first 4 variables from the logistic regression

Variable	Odds Ratio	p
`x1`	1.07	0.69
`x2`	1.26	0.15
`x3`	1.07	0.68
`x4`	0.54	$< .001$

For the boosting model, figure 7 shows the R^2 value on a test dataset as a function of the number of interactions. Clearly, the main-effect model is not sufficiently complex. The slight dip in the curve for `interaction(6)` is just a reminder that these values are estimates and that they are variable. The maximal R^2 is reached for `interaction(8)`, but any number of interactions greater or equal to 4 would probably perform similarly. The large number of interactions is somewhat surprising because the model in (2) contains only a 3-level interaction. I speculate that the nonlinearities are more easily accommodated with additional nodes.

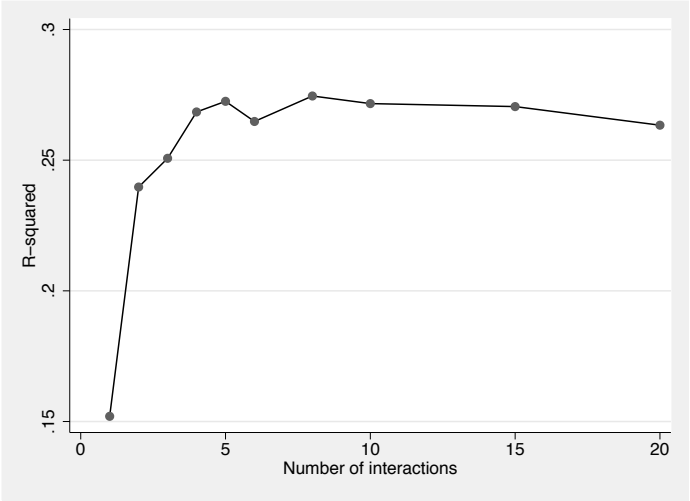

Figure 7: Scatterplot of the pseudo-R^2 computed on a test dataset versus the number of interactions

I compare classification rates on the test data. Roughly 49% of the test data are classified as zero (y = 0) and 51% as one (y = 1). Using a coin flip to classify observations, one would have been right about half the time. Using logistic regression, 52.0% of observations in the test dataset are classified correctly. This is just barely better than the rate one could have obtained by a coin flip. Because there were a lot of unrelated x-variables, I also tried a backward regression using $p > 0.15$ as criterion to remove a variable. Using the backward regression, 54.1% of the observations were classified correctly. The boosting model with 8 interactions, `shrinkage(0.5)`, and `bag(0.5)` classifies 76.0% of the test data observations correctly.

The Stata output displays the pseudo-R^2 values for logistic regression (pseudo-R^2 = 0.02) and backward logistic regression (pseudo-R^2 = 0.01). Because the training data were used to compute the pseudo-R^2 values, the backward logistic regression necessarily has a lower value. Both values are much lower than the value obtained by boosted logistic regression (test $R^2 = 0.27$).

Because there are only two response values (0 and 1), I use a different plot for calibration than the scatterplot shown in figure 5. If the predicted values are accurate, one would expect that the predicted values are roughly the same as the fraction of response values classified as "1" that give rise to a given predicted value. The fraction of response values classified as "1" can be estimated by averaging or smoothing over response values with similar predictions. In Stata, I use a lowess smoother to compare the predictions from the boosted logistic regression and the linear logistic regression:

```
. twoway (lowess y logit_pred, bwidth(0.2)) (lowess y boost_pred, bwidth(0.2))
> (lfit straight y), xtitle("Actual Values")
> legend(label(1 "Logistic Regression") label(2 "Boosting")
> label(3 "Fitted Values=Actual Values"))
```

Calibration plots for the test data are shown in figure 8. The near horizontal line for logistic regression in the test calibration plot implies that logistic regression classifies 50% of the observations correctly, regardless of the actual predicted value. The logistic regression model does not generalize well.

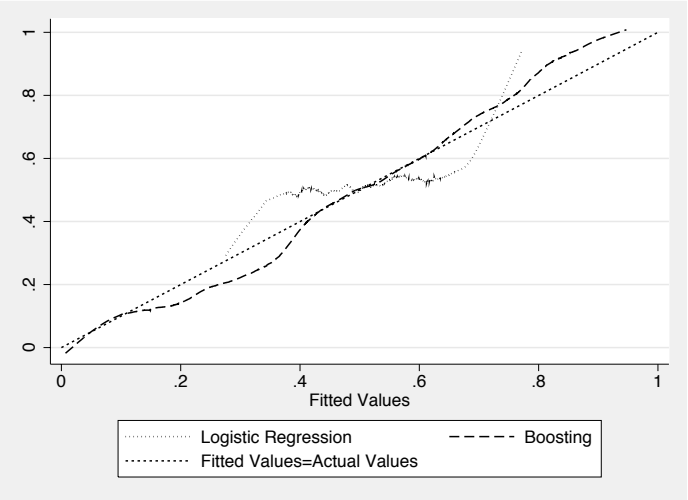

Figure 8: Calibration plot for the logistic regression example: Fitted versus actual values for logistic regression and boosting on the test dataset

A bar chart of the influence of each of the 50 variables is shown in figure 9. The first bar corresponds to `x1`, the second to `x2`, and so forth. Boosting clearly discriminates between the important variables (the first 4 variables) and the remainder. The remaining 46 variables only explain a small percentage of the variation each. The concept of significance does not yet exist in boosting, and there is no formal test that declares these variables "unimportant". Interestingly, the influences of the noise variables, `x5` through `x50`, depend on the number of interactions specified in the model. If the number of interactions is too large, the influence of the noise variables increases.

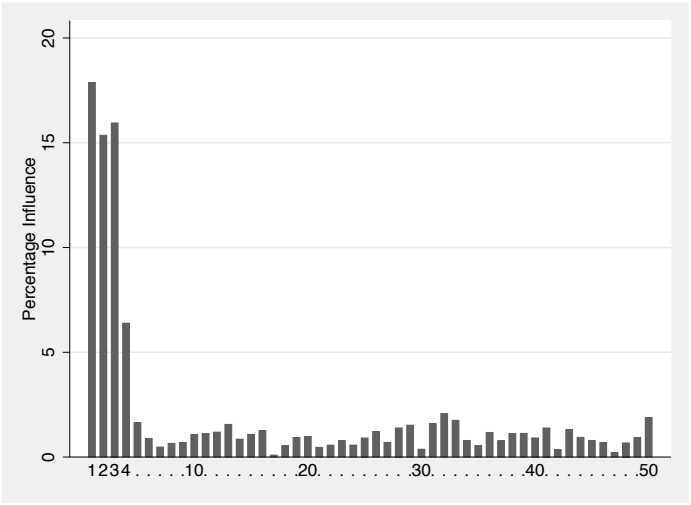

Figure 9: Percentage of influence of variables `x1` through `x50` in the logistic regression example

To visualize the effect of `x4` on the response, I group the predictions (on the probability scale) from the training dataset into 20 groups, according to their corresponding `x4` value. The first group contains predictions where $0 <$ `x4` $< .05$, the second where $0.05 <$ `x4` < 0.1, and so forth. Figure 10 gives a box plot for each of the 20 groups. Consistent with the model in (2), the last group with values $0.95 <$ `x4` < 1.0 has much lower predictions than the other 19 groups. This approach to visualizing data is different from that in the previous example. In the previous example, all other covariates were fixed at one value. All predictions in the training dataset are used. This second method displays much more variation. Visualizing the nonlinear interaction between `x1`, `x2`, and `x3` is, of course, much harder.

(*Continued on next page*)

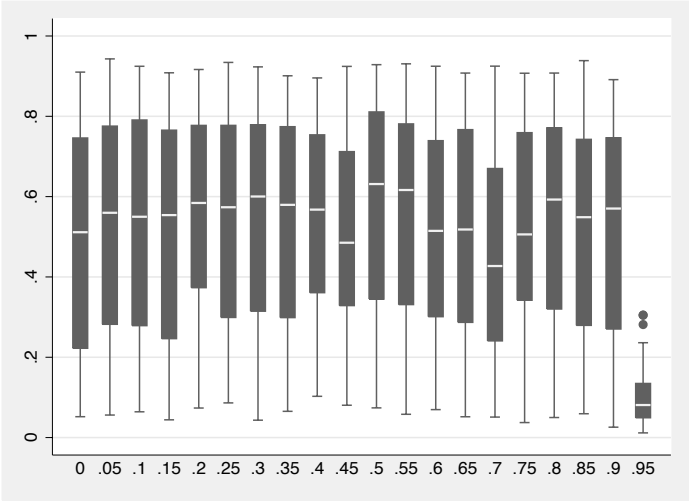

Figure 10: Twenty box plots of predictions for twenty nonoverlapping subsets of `x4`. All predictions stem from the training data.

6 Runtime benchmarks

To generate runtime benchmarks, I ran boosting models on randomly generated data. A single observation is generated by generating random x-values, x_{ij}~Uniform$(0,1)$, $j = 1, \ldots, p$, where p is the number of x-variables, $i = 1, \ldots, n$ denotes the observations, and the output y_i is computed as follows:

$$y_i = \sum_{j=1}^{p} x_{ij}$$

A number of datasets were generated with varying numbers of observations $N \in \{100, 1000, 10000\}$, varying number of x-variables $p \in \{10, 30, 100\}$, varying numbers of iterations `maxiter` $\in \{1000, 5000, 10000\}$, and varying numbers of interactions `interaction` $\in \{2, 4, 6\}$. I fitted a boosting model to each of the datasets specifying `distribution(normal)`, `trainfraction(0.5)`, `bagging(0.5)`, `shrink(0.01)`, and `predict(`*varname*`)`. Table 3 displays runtimes for all 81 combinations of these four factors. Because the runtimes range over several orders of magnitude, table 3 displays the runtime in both seconds (top) and hours (bottom) for each combination. The benchmark calculations were computed on a Dell D600 laptop with a 1.6 GHz processor and 0.5 GB of RAM.

Obs	Variables	Iterations								
		1000			5000			10000		
		Interactions			Interactions			Interactions		
		2	4	6	2	4	6	2	4	6
100										
	10	0.3	0.5	0.6	1.3	2.2	3	2.6	4.4	6.2
		0	0	0	0	0	0	0	0	0
	30	0.6	1.1	1.6	3	5.5	7.8	6.2	11.1	15.6
		0	0	0	0	0	0	0	0	0
	100	2	3.5	5	9.5	17.9	25	19.1	35.3	50
		0	0	0	0	0	0	0	0	0
1000										
	10	3.9	5.6	7.9	15.8	22.1	31.1	27.6	43	62.9
		0	0	0	0	0	0	0	0	0
	30	7	11.4	17.2	32.3	57.6	82.6	63.8	113.9	158.7
		0	0	0	0	0	0	0	0	0
	100	39.7	75.9	111.1	197	380.9	557.6	394.5	758.4	1118
		0	0	0	0.1	0.1	0.2	0.1	0.2	0.3
10000										
	10	29.8	64.5	91.1	156.5	328	422.1	377	599.5	862.5
		0	0	0	0	0.1	0.1	0.1	0.2	0.2
	30	97	178.6	249.6	479.2	906.3	1311.7	964.3	1683.4	2780.7
		0	0	0.1	0.1	0.3	0.4	0.3	0.5	0.8
	100	487.8	935.1	1382.6	2443.7	4668.2	6840.8	4878.4	9360.5	13615.2
		0.1	0.3	0.4	0.7	1.3	1.9	1.4	2.6	3.8

Table 3: Benchmark runtimes for boosting: various combinations of the number of observations (50% used for training, 50% for testing), number of variables, boosting iterations, and number of interactions chosen. The time is given both in seconds (top number) and in hours (bottom number).

The runtime ranges from 0.3 seconds (100 observations, 10 variables, main effects only, 1000 iterations) to 3.8 hours (10,000 observations, 100 variables, six-way interactions, 10,000 iterations). The time increases roughly linearly with the number of iterations, the number of interactions, and the number of variables. The time increases more than linearly with the number of observations. Because the observations are sorted, the runtime is $O\{n\log(n)\}$, where n is the number of observations; i.e., the runtime is bounded by a constant times $n\log(n)$; for linear increases, the runtime would be bounded by a constant times n.

Shrinking does not affect runtime, except in the sense that a smaller shrinkage value will tend to require a larger number of iterations. Bagging improves runtime. Typically, the runtime with bagging with 50% of the observations is roughly 30% faster

than the runtime without bagging. Specifying a logistic distribution instead of a normal distribution increases the runtime only a little (usually less than 10%).

Table 3 displays runtimes for up to 10,000 observations, but 10,000 is not an upper limit. I have used this implementation of boosting with 100,000 observations.

7 Discussion

Boosting is a powerful regression tool. Unlike linear regression, boosting will work when there are more variables than observations. I successfully performed a boosted logistic regression with 200 observations and 500 x-variables. Linear logistic regression will cease to run normally with more than about 50 x-variables and assuming 200 observations because individual observations are uniquely identified.

The question "When should I use boosting?" is not easy to answer, but some general indicators are outlined in table 4. A strength of the boosting algorithm is that interactions and nonlinearities need not be explicitly specified. Unless the functional relationship is highly nonlinear, there is probably little point in using boosted regression on a small dataset with, for example, 50 observations and a handful of variables. In small datasets, linearity is usually an adequate approximation. Large numbers of continuous variables make nonlinearities more likely. Indicator variables have only two levels. Nonlinearities cannot arise from indicator variables. Ordered categorical x-variables are awkward to deal with in regular regression. Because boosting uses trees as a base learner, it is highly suited for the use of ordered variables. The separation of training data and test data guards against overfitting that may arise in the context of correlated data.

Indicator	Indicator favors the use of boosting	Indicator against the use of boosting	Why?
small dataset		x	linear approximation usually adequate
large dataset	x		nonlinearities and interactions likely
more variables than observations (or close)	x		linear (Gaussian and logistic) regression methods fail
suspected nonlinearities	x		nonlinearities need not be explicitly modeled
suspected interactions	x		interactions need not be explicitly specified
ordered categorical x-variables	x		awkward in parametric regression
correlated data	x		potential for overfitting
x-variables consist of indicator variables only		x	nonlinearities cannot arise from indicator variables; interactions still might

Table 4: Some indicators in favor and against the use of boosting

I would like to point out a few issues that the reader may run into when experimenting with the boosting plugin. The "tree fully fit" error means that more tree splits

are required than are possible. This arises, for example, if the data contain only 10 observations but `interaction(11)` (or if `bag(0.5)` and `interaction(6)`) is specified. It will also tend to arise when the number of iterations (`maxiter()`) and the number of interactions (`interaction()`) are accidentally switched. Reducing the number of interactions always solves this problem. A second issue is missing values, which are not supported in the current version of the boosting plugin. I suggest imputing variables ahead of time; for example, using a stratified hotdeck imputation (my implementation of hotdeck, `hotdeckvar`, is available from *www.schonlau.net*, or by typing `net search hotdeckvar` within Stata). Another option that is popular in the social sciences is to create variables that flag missing data and add them to the list of covariates. Discarding observations with missing values is a less-desirable alternative. I am working on a future release that allows saving the boosting model in Stata.

8 Acknowledgments

I am most grateful to Gregory Ridgeway, developer of the GBM boosting package in *R*, for many discussions on boosting, advice on C++ programming, and for comments on earlier versions of this paper. I am equally grateful to Nelson Lim at the RAND Corporation for his support and interest in this methodology and for comments on earlier versions of this paper.

9 References

Bauer, E. and R. Kohavi. 1999. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning* 36: 105–139.

Breiman, L., J. Friedman, R. Olshen, and C. Stone. 1984. *Classification and Regression Trees*. Belmont, CA: Wadsworth.

Freund, Y. and R. E. Schapire. 1997. A decision-theoretic generalization of online learning and an application to boosting. *Journal of Computer and System Sciences* 55(1): 119–139.

Friedman, J. 2001. Greedy function approximation: a gradient boosting machine. *Annals of Statistics* 29: 1189–1232.

Friedman, J., T. Hastie, and R. Tibshirani. 2000. Additive logistic regression: a statistical view of boosting. *Annals of Statistics* 28: 337–407.

Hastie, T., R. Tibshirani, and J. Friedman. 2001. *The Elements of Statistical Learning*. New York: Springer.

Long, J. S. and J. Freese. 2003. *Regression Models for Categorical Dependent Variables Using Stata*. rev. ed. College Station, TX: Stata Press.

McCullagh, P. and J. A. Nelder. 1989. *Generalized Linear Models*. 2nd ed. London: Chapman & Hall.

Ridgeway, G. 1999. The state of boosting. *Computing Science and Statistics* 31: 172–181. Also available at *http://www.i-pensieri.com/gregr/papers.shtml*.

About the Author

Matthias Schonlau is the Head of the Statistical Consulting Service at the RAND Corporation. He is always interested in writing reusable statistical programs and documenting them—he wishes he had more time for that. Matt is also interested in data visualization and causal inference.