Generalized Boosted Models: A guide to the gbm package

Greg Ridgeway

February 24, 2010

Boosting takes on various forms with different programs using different loss functions, different base models, and different optimization schemes. The gbm package takes the approach described in [2] and [3]. Some of the terminology differs, mostly due to an effort to cast boosting terms into more standard statistical terminology (e.g. deviance). In addition, the gbm package implements boosting for models commonly used in statistics but not commonly associated with boosting. The Cox proportional hazard model, for example, is an incredibly useful model and the boosting framework applies quite readily with only slight modification [5]. Also some algorithms implemented in the gbm package differ from the standard implementation. The AdaBoost algorithm [1] has a particular loss function and a particular optimization algorithm associated with it. The gbm implementation of AdaBoost adopts AdaBoost's exponential loss function (its bound on misclassification rate) but uses Friedman's gradient descent algorithm rather than the original one proposed. So the main purposes of this document is to spell out in detail what the gbm package implements.

1 Gradient boosting

This section essentially presents the derivation of boosting described in [2]. The gbm package also adopts the stochastic gradient boosting strategy, a small but important tweak on the basic algorithm, described in [3].

1.1 Friedman's gradient boosting machine

Friedman (2001) and the companion paper Friedman (2002) extended the work of Friedman, Hastie, and Tibshirani (2000) and laid the ground work for a new generation of boosting algorithms. Using the connection between boosting and optimization, this new work proposes the Gradient Boosting Machine.

In any function estimation problem we wish to find a regression function, $\hat{f}(\mathbf{x})$, that minimizes the expectation of some loss function, $\Psi(y, f)$, as shown in (4).

$$\hat{f}(\mathbf{x}) \quad = \quad \arg\min_{f(\mathbf{x})} \mathrm{E}_{y,\mathbf{x}} \Psi(y, f(\mathbf{x}))$$

Initialize $\hat{f}(\mathbf{x})$ to be a constant, $\hat{f}(\mathbf{x}) = \arg\min_\rho \sum_{i=1}^N \Psi(y_i, \rho)$.
For t in $1, \ldots, T$ do

1. Compute the negative gradient as the working response

$$z_i = -\frac{\partial}{\partial f(\mathbf{x}_i)} \Psi(y_i, f(\mathbf{x}_i)) \Big|_{f(\mathbf{x}_i)=\hat{f}(\mathbf{x}_i)} \tag{1}$$

2. Fit a regression model, $g(\mathbf{x})$, predicting z_i from the covariates $\mathbf{x}_i$.

3. Choose a gradient descent step size as

$$\rho = \arg\min_\rho \sum_{i=1}^N \Psi(y_i, \hat{f}(\mathbf{x}_i) + \rho g(\mathbf{x}_i)) \tag{2}$$

4. Update the estimate of $f(\mathbf{x})$ as

$$\hat{f}(\mathbf{x}) \leftarrow \hat{f}(\mathbf{x}) + \rho g(\mathbf{x}) \tag{3}$$

Figure 1: Friedman's Gradient Boost algorithm

$$= \quad \arg\min_{f(\mathbf{x})} \mathrm{E}_x \left[\mathrm{E}_{y|\mathbf{x}} \Psi(y, f(\mathbf{x})) \Big| \mathbf{x} \right] \tag{4}$$

We will focus on finding estimates of $f(\mathbf{x})$ such that

$$\hat{f}(\mathbf{x}) = \arg\min_{f(\mathbf{x})} \mathrm{E}_{y|\mathbf{x}} \left[\Psi(y, f(\mathbf{x})) | \mathbf{x} \right] \tag{5}$$

Parametric regression models assume that $f(\mathbf{x})$ is a function with a finite number of parameters, β, and estimates them by selecting those values that minimize a loss function (e.g. squared error loss) over a training sample of N observations on $(y, \mathbf{x})$ pairs as in (6).

$$\hat{\beta} = \arg\min_{\beta} \sum_{i=1}^{N} \Psi(y_i, f(\mathbf{x}_i; \beta)) \tag{6}$$

When we wish to estimate $f(\mathbf{x})$ non-parametrically the task becomes more difficult. Again we can proceed similarly to [4] and modify our current estimate of $f(\mathbf{x})$ by adding a new function $f(\mathbf{x})$ in a greedy fashion. Letting $f_i = f(\mathbf{x}_i)$, we see that we want to decrease the N dimensional function

$$\begin{aligned} J(\mathbf{f}) &= \sum_{i=1}^{N} \Psi(y_i, f(\mathbf{x}_i)) \\ &= \sum_{i=1}^{N} \Psi(y_i, F_i). \end{aligned} \tag{7}$$

The negative gradient of $J(\mathbf{f})$ indicates the direction of the locally greatest decrease in $J(\mathbf{f})$. Gradient descent would then have us modify $\mathbf{f}$ as

$$\hat{\mathbf{f}} \leftarrow \hat{\mathbf{f}} - \rho \nabla J(\mathbf{f}) \tag{8}$$

where ρ is the size of the step along the direction of greatest descent. Clearly, this step alone is far from our desired goal. First, it only fits f at values of $\mathbf{x}$ for which we have observations. Second, it does not take into account that observations with similar $\mathbf{x}$ are likely to have similar values of $f(\mathbf{x})$. Both these problems would have disastrous effects on generalization error. However, Friedman suggests selecting a class of functions that use the covariate information to approximate the gradient, usually a regression tree. This line of reasoning produces his Gradient Boosting algorithm shown in Figure 1. At each iteration the algorithm determines the direction, the gradient, in which it needs to improve the fit to the data and selects a particular model from the allowable class of functions that is in most agreement with the direction. In the case of squared-error loss, $\Psi(y_i, f(\mathbf{x}_i)) = \sum_{i=1}^{N} (y_i - f(\mathbf{x}_i))^2$, this algorithm corresponds exactly to residual fitting.

There are various ways to extend and improve upon the basic framework suggested in Figure 1. For example, Friedman (2001) substituted several choices

in for Ψ to develop new boosting algorithms for robust regression with least absolute deviation and Huber loss functions. Friedman (2002) showed that a simple subsampling trick can greatly improve predictive performance while simultaneously reduce computation time. Section 2 discusses some of these modifications.

2 Improving boosting methods using control of the learning rate, sub-sampling, and a decomposition for interpretation

This section explores the variations of the previous algorithms that have the potential to improve their predictive performance and interpretability. In particular, by controlling the optimization speed or learning rate, introducing low-variance regression methods, and applying ideas from robust regression we can produce non-parametric regression procedures with many desirable properties. As a by-product some of these modifications lead directly into implementations for learning from massive datasets. All these methods take advantage of the general form of boosting

$$\hat{f}(\mathbf{x}) \leftarrow \hat{f}(\mathbf{x}) + \mathrm{E}(z(y, \hat{f}(\mathbf{x}))|\mathbf{x}). \tag{9}$$

So far we have taken advantage of this form only by substituting in our favorite regression procedure for $\mathrm{E}_w(z|\mathbf{x})$. I will discuss some modifications to estimating $\mathrm{E}_w(z|\mathbf{x})$ that have the potential to improve our algorithm.

2.1 Decreasing the learning rate

As several authors have phrased slightly differently, "...boosting, whatever flavor, seldom seems to overfit, no matter how many terms are included in the additive expansion". This is not true as the discussion to [4] points out.

In the update step of any boosting algorithm we can introduce a learning rate to dampen the proposed move.

$$\hat{f}(\mathbf{x}) \leftarrow \hat{f}(\mathbf{x}) + \lambda \mathrm{E}(z(y, \hat{f}(\mathbf{x}))|\mathbf{x}). \tag{10}$$

By multiplying the gradient step by λ as in equation 10 we have control on the rate at which the boosting algorithm descends the error surface (or ascends the likelihood surface). When $\lambda = 1$ we return to performing full gradient steps. Friedman (2001) relates the learning rate to regularization through shrinkage.

The optimal number of iterations, T, and the learning rate, λ, depend on each other. In practice I set λ to be as small as possible and then select T by cross-validation. Performance is best when λ is as small as possible performance with decreasing marginal utility for smaller and smaller λ. Slower learning rates do not necessarily scale the number of optimal iterations. That is, if when $\lambda = 1.0$ and the optimal T is 100 iterations, does *not* necessarily imply that when $\lambda = 0.1$ the optimal T is 1000 iterations.

2.2 Variance reduction using subsampling

Friedman (2002) proposed the stochastic gradient boosting algorithm that simply samples uniformly without replacement from the dataset before estimating the next gradient step. He found that this additional step greatly improved performance. We estimate the regression $\mathrm{E}(z(y, \hat{f}(\mathbf{x}))|\mathbf{x})$ using a random subsample of the dataset.

2.3 ANOVA decomposition

Certain function approximation methods are decomposable in terms of a "functional ANOVA decomposition". That is a function is decomposable as

$$f(\mathbf{x}) = \sum_j f_j(x_j) + \sum_{jk} f_{jk}(x_j, x_k) + \sum_{jk\ell} f_{jk\ell}(x_j, x_k, x_\ell) + \cdots. \tag{11}$$

This applies to boosted trees. Regression stumps (one split decision trees) depend on only one variable and fall into the first term of 11. Trees with two splits fall into the second term of 11 and so on. By restricting the depth of the trees produced on each boosting iteration we can control the order of approximation. Often additive components are sufficient to approximate a multivariate function well, generalized additive models, the naïve Bayes classifier, and boosted stumps for example. When the approximation is restricted to a first order we can also produce plots of x_j versus $f_j(x_j)$ to demonstrate how changes in x_j might affect changes in the response variable.

2.4 Relative influence

Friedman (2001) also develops an extension of a variable's "relative influence" for boosted estimates. For tree based methods the approximate relative influence of a variable x_j is

$$\hat{J}_j^2 = \sum_{\text{splits on } x_j} I_t^2 \tag{12}$$

where I_t^2 is the empirical improvement by splitting on x_j at that point. Friedman's extension to boosted models is to average the relative influence of variable x_j across all the trees generated by the boosting algorithm.

3 Common user options

This section discusses the options to gbm that most users will need to change or tune.

3.1 Loss function

The first and foremost choice is `distribution`. This should be easily dictated by the application. For most classification problems either `bernoulli` or `adaboost` will be appropriate, the former being recommended. For continuous

Select

- a loss function (`distribution`)
- the number of iterations, T (`n.trees`)
- the depth of each tree, K (`interaction.depth`)
- the shrinkage (or learning rate) parameter, λ (`shrinkage`)
- the subsampling rate, p (`bag.fraction`)

Initialize $\hat{f}(\mathbf{x})$ to be a constant, $\hat{f}(\mathbf{x}) = \arg\min_\rho \sum_{i=1}^{N} \Psi(y_i, \rho)$
For t in $1, \ldots, T$ do

1. Compute the negative gradient as the working response

$$z_i = -\frac{\partial}{\partial f(\mathbf{x}_i)} \Psi(y_i, f(\mathbf{x}_i)) \Big|_{f(\mathbf{x}_i)=\hat{f}(\mathbf{x}_i)} \tag{13}$$

2. Randomly select $p \times N$ cases from the dataset
3. Fit a regression tree with K terminal nodes, $g(\mathbf{x}) = \mathrm{E}(z|\mathbf{x})$. This tree is fit using only those randomly selected observations
4. Compute the optimal terminal node predictions, $\rho_1, \ldots, \rho_K$, as

$$\rho_k = \arg\min_\rho \sum_{\mathbf{x}_i \in S_k} \Psi(y_i, \hat{f}(\mathbf{x}_i) + \rho) \tag{14}$$

   where S_k is the set of $\mathbf{x}$s that define terminal node k.

5. Update $\hat{f}(\mathbf{x})$ as

$$\hat{f}(\mathbf{x}) \leftarrow \hat{f}(\mathbf{x}) + \lambda \rho_{k(\mathbf{x})} \tag{15}$$

   where $k(\mathbf{x})$ indicates the index of the terminal node into which an observation with features $\mathbf{x}$ would fall. Again this step uses only the randomly selected observations

Figure 2: Boosting as implemented in `gbm()`

outcomes the choices are `gaussian` (for minimizing squared error), `laplace` (for minimizing absolute error), and quantile regression (for estimating percentiles of the conditional distribution of the outcome). Censored survival outcomes should require `coxph`. Count outcomes may use `poisson` although one might also consider `gaussian` or `laplace` depending on the analytical goals.

3.2 The relationship between shrinkage and number of iterations

The issues that most new users of gbm struggle with are the choice of `n.trees` and `shrinkage`. It is important to know that smaller values of `shrinkage` (almost) always give improved predictive performance. That is, setting `shrinkage=0.001` will almost certainly result in a model with better out-of-sample predictive performance than setting `shrinkage=0.01`. However, there are computational costs, both storage and CPU time, associated with setting `shrinkage` to be low. The model with `shrinkage=0.001` will likely require ten times as many iterations as the model with `shrinkage=0.01`, increasing storage and computation time by a factor of 10. Figure 3 shows the relationship between predictive performance, the number of iterations, and the shrinkage parameter. Note that the increase in the optimal number of iterations between two choices for shrinkage is roughly equal to the ratio of the shrinkage parameters. It is generally the case that for small shrinkage parameters, 0.001 for example, there is a fairly long plateau in which predictive performance is at its best. My rule of thumb is to set `shrinkage` as small as possible while still being able to fit the model in a reasonable amount of time and storage. I usually aim for 3,000 to 10,000 iterations with shrinkage rates between 0.01 and 0.001.

3.3 Estimating the optimal number of iterations

gbm offers three methods for estimating the optimal number of iterations after the gbm model has been fit, an independent test set (`test`), out-of-bag estimation (`OOB`), and v-fold cross validation (`cv`). The function `gbm.perf` computes the iteration estimate.

Like Friedman's MART software, the independent test set method uses a single holdout test set to select the optimal number of iterations. If `train.fraction` is set to be less than 1, then only the *first* `train.fraction`×`nrow(data)` will be used to fit the model. Note that if the data are sorted in a systematic way (such as cases for which $y = 1$ come first), then the data should be shuffled before running gbm. Those observations not used in the model fit can be used to get an unbiased estimate of the optimal number of iterations. The downside of this method is that a considerable number of observations are used to estimate the single regularization parameter (number of iterations) leaving a reduced dataset for estimating the entire multivariate model structure. Use `gbm.perf(...,method="test")` to obtain an estimate of the optimal number of iterations using the held out test set.

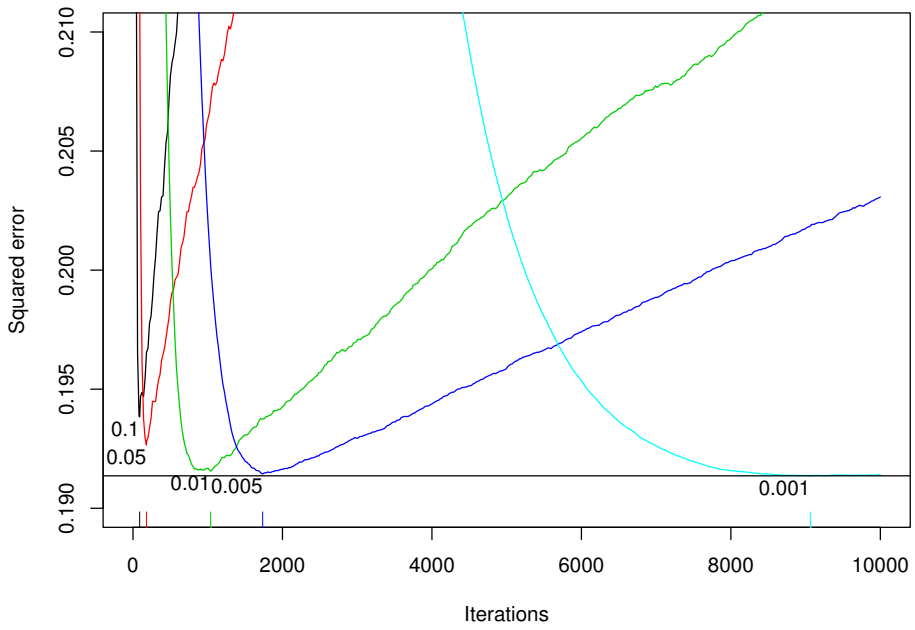

Figure 3: Out-of-sample predictive performance by number of iterations and shrinkage. Smaller values of the shrinkage parameter offer improved predictive performance, but with decreasing marginal improvement.

If `bag.fraction` is set to be greater than 0 (0.5 is recommended), gbm computes an out-of-bag estimate of the improvement in predictive performance. It evaluates the reduction in deviance on those observations not used in selecting the next regression tree. The out-of-bag estimator underestimates the reduction in deviance. As a result, it almost always is too conservative in its selection for the optimal number of iterations. The motivation behind this method was to avoid having to set aside a large independent dataset, which reduces the information available for learning the model structure. Use `gbm.perf(...,method="OOB")` to obtain the OOB estimate.

Lastly, gbm offers v-fold cross validation for estimating the optimal number of iterations. If when fitting the gbm model, `cv.folds=5` then gbm will do 5-fold cross validation. gbm will fit five gbm models in order to compute the cross validation error estimate and then will fit a sixth and final gbm model with `n.trees`iterations using all of the data. The returned model object will have a component labeled `cv.error`. Note that `gbm.more` will do additional gbm iterations but will not add to the `cv.error` component. Use

`gbm.perf(...,method="cv")` to obtain the cross validation estimate.

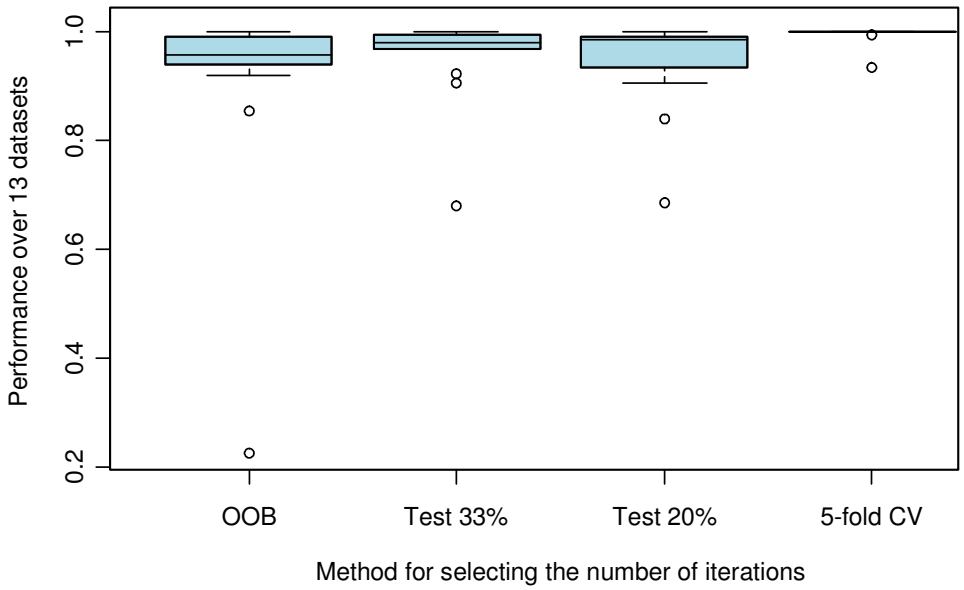

Figure 4: Out-of-sample predictive performance of four methods of selecting the optimal number of iterations. The vertical axis plots performance relative the best. The boxplots indicate relative performance across thirteen real datasets from the UCI repository. See `demo(OOB-reps)`.

Figure 4 compares the three methods for estimating the optimal number of iterations across 13 datasets. The boxplots show the methods performance relative to the best method on that dataset. For most datasets the method perform similarly, however, 5-fold cross validation is consistently the best of them. OOB, using a 33% test set, and using a 20% test set all have datasets for which the perform considerably worse than the best method. My recommendation is to use 5- or 10-fold cross validation if you can afford the computing time. Otherwise you may choose among the other options, knowing that OOB is conservative.

4 Available distributions

This section gives some of the mathematical detail for each of the distribution options that gbm offers. The gbm engine written in C++ has access to a C++ class for each of these distributions. Each class contains methods for computing the associated deviance, initial value, the gradient, and the constants to predict in each terminal node.

In the equations shown below, for non-zero offset terms, replace $f(\mathbf{x}_i)$ with $o_i + f(\mathbf{x}_i)$.

4.1 Gaussian

Deviance	$\frac{1}{\sum w_i}\sum w_i(y_i - f(\mathbf{x}_i))^2$
Initial value	$f(\mathbf{x}) = \frac{\sum w_i(y_i - o_i)}{\sum w_i}$
Gradient	$z_i = y_i - f(\mathbf{x}_i)$
Terminal node estimates	$\frac{\sum w_i(y_i - f(\mathbf{x}_i))}{\sum w_i}$

4.2 AdaBoost

Deviance	$\frac{1}{\sum w_i}\sum w_i \exp(-(2y_i - 1)f(\mathbf{x}_i))$
Initial value	$\frac{1}{2}\log\frac{\sum y_i w_i e^{-o_i}}{\sum(1-y_i)w_i e^{o_i}}$
Gradient	$z_i = -(2y_i - 1)\exp(-(2y_i - 1)f(\mathbf{x}_i))$
Terminal node estimates	$\frac{\sum(2y_i - 1)w_i \exp(-(2y_i - 1)f(\mathbf{x}_i))}{\sum w_i \exp(-(2y_i - 1)f(\mathbf{x}_i))}$

4.3 Bernoulli

Deviance	$-2\frac{1}{\sum w_i}\sum w_i(y_i f(\mathbf{x}_i) - \log(1 + \exp(f(\mathbf{x}_i))))$
Initial value	$\log\frac{\sum w_i y_i}{\sum w_i(1-y_i)}$
Gradient	$z_i = y_i - \frac{1}{1+\exp(-f(\mathbf{x}_i))}$
Terminal node estimates	$\frac{\sum w_i(y_i - p_i)}{\sum w_i p_i(1-p_i)}$ where $p_i = \frac{1}{1+\exp(-f(\mathbf{x}_i))}$

Notes:

- For non-zero offset terms, the computation of the initial value requires Newton-Raphson. Initialize $f_0 = 0$ and iterate $f_0 \leftarrow f_0 + \frac{\sum w_i(y_i - p_i)}{\sum w_i p_i(1-p_i)}$ where $p_i = \frac{1}{1+\exp(-(o_i + f_0))}$.

4.4 Laplace

Deviance	$\frac{1}{\sum w_i}\sum w_i\lvert y_i - f(\mathbf{x}_i)\rvert$
Initial value	$\text{median}_w(y)$
Gradient	$z_i = \text{sign}(y_i - f(\mathbf{x}_i))$
Terminal node estimates	$\text{median}_w(z)$

Notes:

- $\text{median}_w(y)$ denotes the weighted median, defined as the solution to the equation $\frac{\sum w_i I(y_i \le m)}{\sum w_i} = \frac{1}{2}$
- `gbm()` currently does not implement the weighted median and issues a warning when the user uses weighted data with `distribution="laplace"`.

4.5 Quantile regression

Contributed by Brian Kriegler.

Deviance	$\frac{1}{\sum w_i}\left(\alpha \sum_{y_i > f(\mathbf{x}_i)} w_i(y_i - f(\mathbf{x}_i)) + (1-\alpha)\sum_{y_i \le f(\mathbf{x}_i)} w_i(f(\mathbf{x}_i) - y_i)\right)$
Initial value	$\text{quantile}_w^{(\alpha)}(y)$
Gradient	$z_i = \alpha I(y_i > f(\mathbf{x}_i)) - (1-\alpha) I(y_i \le f(\mathbf{x}_i))$
Terminal node estimates	$\text{quantile}_w^{(\alpha)}(z)$

Notes:

- $\text{quantile}_w^{(\alpha)}(y)$ denotes the weighted quantile, defined as the solution to the equation $\frac{\sum w_i I(y_i \le q)}{\sum w_i} = \alpha$
- `gbm()` currently does not implement the weighted median and issues a warning when the user uses weighted data with `distribution=list(name="quantile")`.

4.6 Cox Proportional Hazard

Deviance	$-2\sum w_i(\delta_i(f(\mathbf{x}_i) - \log(R_i/w_i)))$
Gradient	$z_i = \delta_i - \sum_j \delta_j \frac{w_j I(t_i \ge t_j) e^{f(\mathbf{x}_i)}}{\sum_k w_k I(t_k \ge t_j) e^{f(\mathbf{x}_k)}}$
Initial value	0
Terminal node estimates	Newton-Raphson algorithm

1. Initialize the terminal node predictions to 0, $\boldsymbol{\rho} = 0$
2. Let $p_i^{(k)} = \frac{\sum_j I(k(j)=k) I(t_j \ge t_i) e^{f(\mathbf{x}_i)+\rho_k}}{\sum_j I(t_j \ge t_i) e^{f(\mathbf{x}_i)+\rho_k}}$
3. Let $g_k = \sum w_i \delta_i \left(I(k(i)=k) - p_i^{(k)}\right)$
4. Let $\mathbf{H}$ be a $k \times k$ matrix with diagonal elements
   (a) Set diagonal elements $H_{mm} = \sum w_i \delta_i p_i^{(m)} \left(1 - p_i^{(m)}\right)$
   (b) Set off diagonal elements $H_{mn} = -\sum w_i \delta_i p_i^{(m)} p_i^{(n)}$
5. Newton-Raphson update $\boldsymbol{\rho} \leftarrow \boldsymbol{\rho} - \mathbf{H}^{-1}\mathbf{g}$
6. Return to step 2 until convergence

Notes:

- t_i is the survival time and δ_i is the death indicator.
- R_i denotes the hazard for the risk set, $R_i = \sum_{j=1}^{N} w_j I(t_j \geq t_i) e^{f(\mathbf{x}_i)}$
- $k(i)$ indexes the terminal node of observation i
- For speed, `gbm()` does only one step of the Newton-Raphson algorithm rather than iterating to convergence. No appreciable loss of accuracy since the next boosting iteration will simply correct for the prior iterations inadequacy.
- `gbm()` initially sorts the data by survival time. Doing this reduces the computation of the risk set from $O(n^2)$ to $O(n)$ at the cost of a single up front sort on survival time. After the model is fit, the data are then put back in their original order.

4.7 Poisson

Deviance	$-2\frac{1}{\sum w_i}\sum w_i(y_i f(\mathbf{x}_i) - \exp(f(\mathbf{x}_i)))$
Initial value	$f(\mathbf{x}) = \log\left(\frac{\sum w_i y_i}{\sum w_i e^{o_i}}\right)$
Gradient	$z_i = y_i - \exp(f(\mathbf{x}_i))$
Terminal node estimates	$\log\frac{\sum w_i y_i}{\sum w_i \exp(f(\mathbf{x}_i))}$

The Poisson class includes special safeguards so that the most extreme predicted values are e^{-19} and e^{+19}. This behavior is consistent with `glm()`.

References

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

[2] J.H. Friedman (2001). "Greedy Function Approximation: A Gradient Boosting Machine," *Annals of Statistics* 29(5):1189-1232.

[3] J.H. Friedman (2002). "Stochastic Gradient Boosting," *Computational Statistics and Data Analysis* 38(4):367-378.

[4] J.H. Friedman, T. Hastie, R. Tibshirani (2000). "Additive Logistic Regression: a Statistical View of Boosting," *Annals of Statistics* 28(2):337-374.

[5] G. Ridgeway (1999). "The state of boosting," *Computing Science and Statistics* 31:172-181.