10 Boosting

Boosting is another ensemble technique. It sequentially builds a prediction model.

- Any method of model building can be boosted!
- It is best to boost simple models such as linear models or shallow trees.
- In every step we fit a model that is tailored to those observations that have not been well fit.

10.1 A few traditional boosting algorithms

Algorithm 1: **AdaBoost.M1** for binary outcomes coded as $\{-1, 1\}$.

```
(1) Initialize weights: $w_i = \frac{1}{n}$.
(2) For $m = 1, \ldots, M$:
    (a) fit a classifier $g_m(x)$ to data with weights $\{w_i\}$;
    (b) calculate
        • error rate $e_m = \sum w_i I(y_i \neq g_m(x_i))$
        • weight multiplier $f_m = (1 - e_m)/e_m$
    (c) update weight:
        • if $y_i = g_m(x_i)$, $w_i \leftarrow w_i$
        • if $y_i \neq g_m(x_i)$, $w_i \leftarrow w_i f_m$
        • Normalize the new weights so that their sum is 1
(3) The final model is $g(x) = \text{sign}(\sum_m \alpha_m g_m(x))$, where $\alpha_m = \log(f_m)$.
```

- The lower e_m, the higher f_m and α_m. When $e_m = 0.5$, $f_m = 1$, $\alpha_m = 0$; when $e_m = 0.3$, $f_m = 2.33$, $\alpha_m = 0.85$; when $e_m = 0.1$, $f_m = 9$, $\alpha_m = 2.20$. If $e_m > 0.5$, $f_m < 1$, $\alpha_m < 0$; but this situation is very rare.

Algorithm 2 for quantitative outcomes under squared error loss (ISLR Algorithm 8.2):

```
(1) Initialize model: $f_0(x) = \bar{y}$.
(2) For $m = 1, \ldots, M$:
    (a) compute the residuals $r_{im} = y_i - f_{m-1}(x_i)$;
    (b) fit a model $g_m(x)$ to data $\{(x_i, r_{im}) : i = 1, \ldots, n\}$, where $r_{im}$ is treated as the outcome;
    (c) update the model $f_m(x) = f_{m-1}(x) + \epsilon g_m(x)$.
```

The final model is $f_M(x)$.

- The **shrinkage parameter** (or **learning rate**) ϵ controls the rate boosting learns. It is a hyperparameter.
- One can also initialize model with $f_0(x) = 0$.

Algorithm 3: **Gradient boosting** generalizes Algorithm 2 to any loss function $L(y, \hat{y})$.

```
(1) Initialize model: $f_0(x_i) = \gamma_0 = \arg\min_\gamma \sum L(y_i, \gamma)$;
(2) For $m = 1$ to $M$:
    (a) compute the gradients (pseudo-residuals) $r_{im} = -\frac{\partial L(y_i, \gamma)}{\partial \gamma}|_{\gamma = f_{m-1}(x_i)}$.
    (b) fit a model $g_m(x)$ to data $\{(x_i, r_{im}) : i = 1, \ldots, n\}$, where $r_{im}$ is treated as the outcome;
    (c) set $f_m(x) = f_{m-1}(x) + \epsilon g_m(x)$.
```

The final model is $f_M(x)$.

- For squared error loss $L = \frac{1}{2}(y - \hat{y})^2$, its gradient is $-\frac{\partial L}{\partial \hat{y}} = y - \hat{y}$, which is the traditional residual.

All these are special cases of **Algorithm 4**: Forward stagewise fitting. Consider $\mathcal{M} = \{g(x; \gamma)\}$, a family of models indexed by γ, and a loss function L.

```
(1) Initialize model: $f_0(x) = 0$.
(2) For $m = 1, \ldots, M$:
    (a) compute $\gamma_m = \arg\min_\gamma \sum_i L(y_i, f_{m-1}(x_i) + g(x_i; \gamma))$, where $g(x; \gamma) \in \mathcal{M}$;
    (b) set $f_m(x) = f_{m-1}(x) + \epsilon g(x; \gamma_m)$.
```

The final model is $f_M(x)$.

- When L is the squared error loss, this becomes Algorithm 2.
- For binary outcomes coded as $\{-1, 1\}$ and exponential loss, $L(y, f(x)) = e^{-yf(x)}$, this becomes AdaBoost.M1.
- Algorithm 3 is an approximation of Algorithm 4. (2a) in Algorithm 3 is often more feasible than (2a) in Algorithm 4.

Boosting can overfit if M (number of models) is too large. Thus the hyperparameter M needs to be determined with cross-validation.

Boosting vs. GAM: The backfitting in GAM is similar to boosting in that we fit to partial residuals at every step. In GAM, variables are given an equal chance to be considered, and the function for x_i is refit anew to partial residuals. In boosting, the functions for x_i are accumulated. This accumulation can lead to overfitting.

10.2 Tree boosting

The individual models g_m in Algorithms 1–3 can be trees. One may set a maximum number d of splits (called "interaction depth") for each tree in (2a). In this case, there are **3 hyperparameters**: M, ϵ, d. We can overfit if M is too large. With a small ϵ, it takes many trees to fit, the fits are smoother, and they are less sensitive to M. The interaction depth d controls the complexity (d-way interactions) of individual trees.

When $d = 1$, we boost *stumps* (a stump is a tree with a single split), and the final model is **an additive model fit in a fully adaptive way** in that it does variable selection and allows for different amount of smoothing for different variables. This adaptive nature can lead to overfitting for a large M. The final function form of each variable can be plotted.

Gradient tree boosting by Friedman: A tree model has two components: a partition of the feature space, and the predicted values for the subsets in the partition. Here, we only take the partition component (in 2a below) but refit a model to obtain the predicted values (in 2b below).

```
(1) Initialize model: $f_0(x_i) = \gamma_0 = \arg\min_\gamma \sum L(y_i, \gamma)$.
(2) For $m = 1$ to $M$:
    (a) compute the gradients $r_{im} = -\frac{\partial L(y_i,\gamma)}{\partial\gamma}|_{\gamma=f_{m-1}(x_i)}$.
    (b) fit a tree with maximum interaction depth $d$ to the targets $r_{im}$ to obtain terminal nodes $R_{jm}$ ($j = 1, \dots, d+1$);
    (c) for terminal node $j$, compute $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$.
    (d) set $f_m(x) = f_{m-1}(x) + \sum_j \gamma_{jm} I(x \in R_{jm})$.
```

Notes:

- In (2c), we can fit a GLM with dummy variables for the terminal nodes and offset $f_{m-1}(x_i)$ for observation i.
- This algorithm is half way between Algorithms 3 and 4.
- Friedman proposed **stochastic gradient boosting** to use a subset of data in (2b). This speeds up computations. But the results will vary a little from one run to another.

We use the `Boston` dataset to build models to predict `medv`, median value of owner-occupied homes in $1000s. We split the data so that 70% are in the training set and 30% in the test set.

```r
library(MASS)
names(Boston); dim(Boston); str(Boston)  ## 506 x 14
apply(is.na(Boston), 2, sum)
hist(Boston$medv, breaks=seq(5,52,2))

set.seed(2018)
split = caret::createDataPartition(y=Boston$medv, p=0.7, list=FALSE)
Bostontrain = Boston[split,]
Bostontest = Boston[-split,]
dim(Bostontrain); dim(Bostontest)  ## 356 x 14, 150 x 14
```

The R `gbm` package has `gbm()` for boosted regression models. The argument `distribution` specifies the base model to boost. The default is `distribution="bernoulli"` for logistic regression. Specify `distribution="gaussian"` to use squared error.

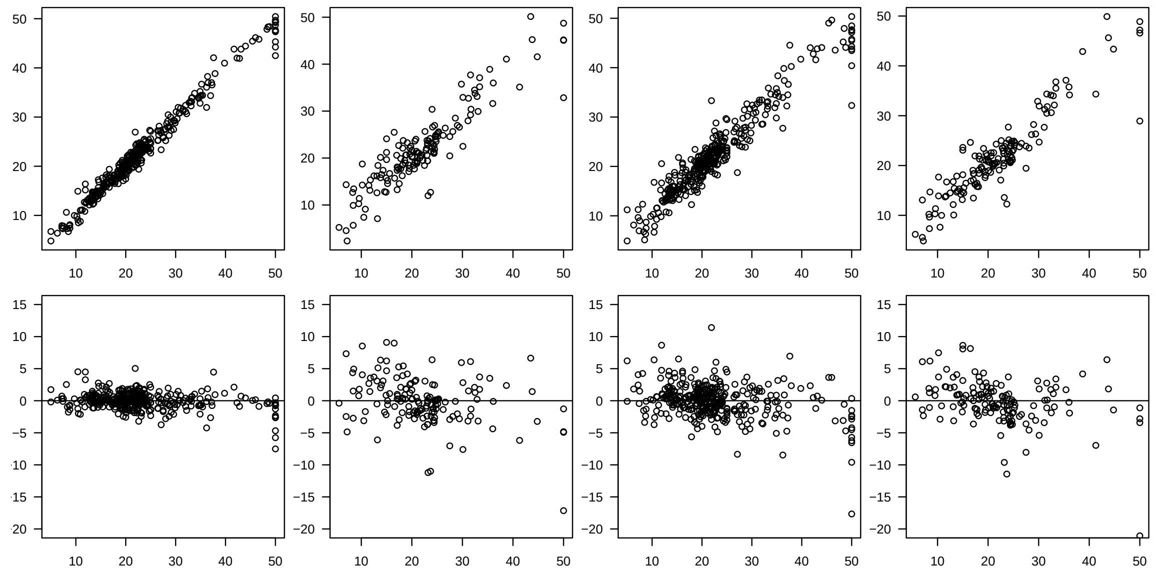
```
library(gbm)
set.seed(2019)
bt.boston1 = gbm(medv ~ ., data=Bostontrain, distribution="gaussian", n.trees=5000)
bt.boston1
names(bt.boston1)
```

By default,

- `n.trees=100` (number of trees; that is, number of steps);
- `interaction.depth=1` (stumps; every tree relies on a single feature);
- `n.minobsinnode=10` (minimum number of observations in a ternimal node);
- `shrinkage=0.1` (shrinkage parameter, learning rate);
- `bag.fraction=0.5` (every tree is grown on 50% randomly selected observations);
- `cv.folds=0` (cross-validation is not performed).

The following fitted-vs-observed and residual-vs-observed plots show that `n.trees=5000` clearly overfit the data because the performance on test set is clearly worse than that on the training set. The results for `n.trees=500` appear to be comparable between the two sets.

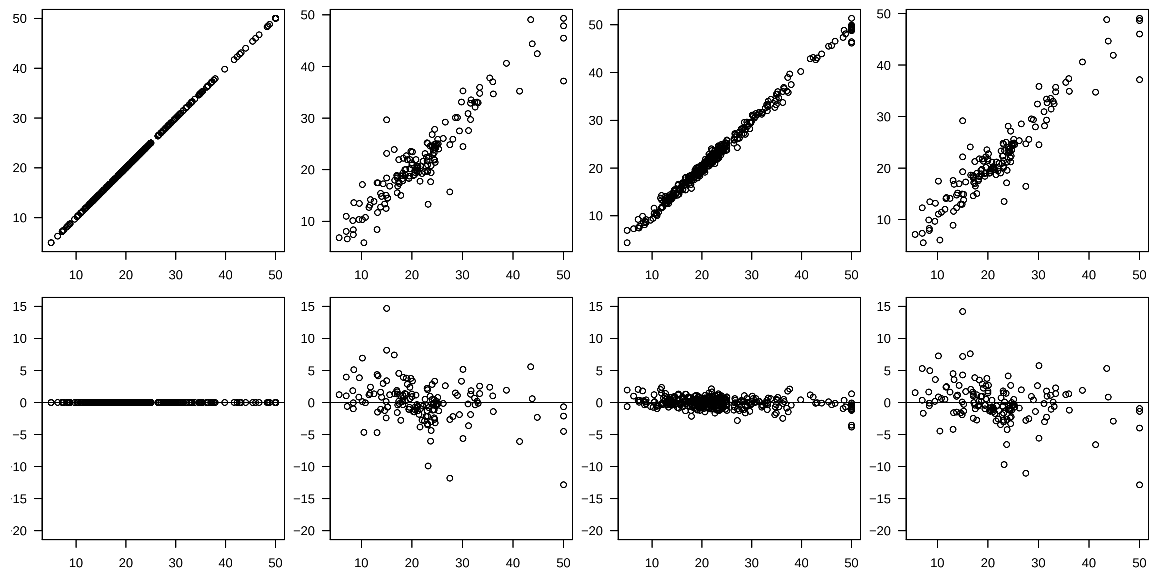
```
par(mfcol=c(2,4), mar=c(2,2,1,1), las=1)
plot(Bostontrain$medv, predict(bt.boston1, n.trees=5000))
plot(Bostontrain$medv, predict(bt.boston1, n.trees=5000)-Bostontrain$medv, ylim=c(-20,15)); abline(h=0)
plot(Bostontest$medv, predict(bt.boston1, Bostontest, n.trees=5000))
plot(Bostontest$medv, predict(bt.boston1, Bostontest, n.trees=5000)-Bostontest$medv, ylim=c(-20,15)); abli
plot(Bostontrain$medv, predict(bt.boston1, n.trees=500))
plot(Bostontrain$medv, predict(bt.boston1, n.trees=500)-Bostontrain$medv, ylim=c(-20,15)); abline(h=0)
plot(Bostontest$medv, predict(bt.boston1, Bostontest, n.trees=500))
plot(Bostontest$medv, predict(bt.boston1, Bostontest, n.trees=500)-Bostontest$medv, ylim=c(-20,15)); ablin
```



For this dataset, deeper trees give a faster model building. In the following model, even `n.trees=500` already overfit.

```
set.seed(2019)
bt.boston4 = gbm(medv ~ ., data=Bostontrain, distribution="gaussian", n.trees=5000, interaction.depth=4)

par(mfcol=c(2,4), mar=c(2,2,1,1), las=1)
plot(Bostontrain$medv, predict(bt.boston4, n.trees=5000))
plot(Bostontrain$medv, predict(bt.boston4, n.trees=5000)-Bostontrain$medv, ylim=c(-20,15)); abline(h=0)
plot(Bostontest$medv, predict(bt.boston4, Bostontest, n.trees=5000))
plot(Bostontest$medv, predict(bt.boston4, Bostontest, n.trees=5000)-Bostontest$medv, ylim=c(-20,15)); abli
plot(Bostontrain$medv, predict(bt.boston4, n.trees=500))
plot(Bostontrain$medv, predict(bt.boston4, n.trees=500)-Bostontrain$medv, ylim=c(-20,15)); abline(h=0)
plot(Bostontest$medv, predict(bt.boston4, Bostontest, n.trees=500))
plot(Bostontest$medv, predict(bt.boston4, Bostontest, n.trees=500)-Bostontest$medv, ylim=c(-20,15)); ablin
```



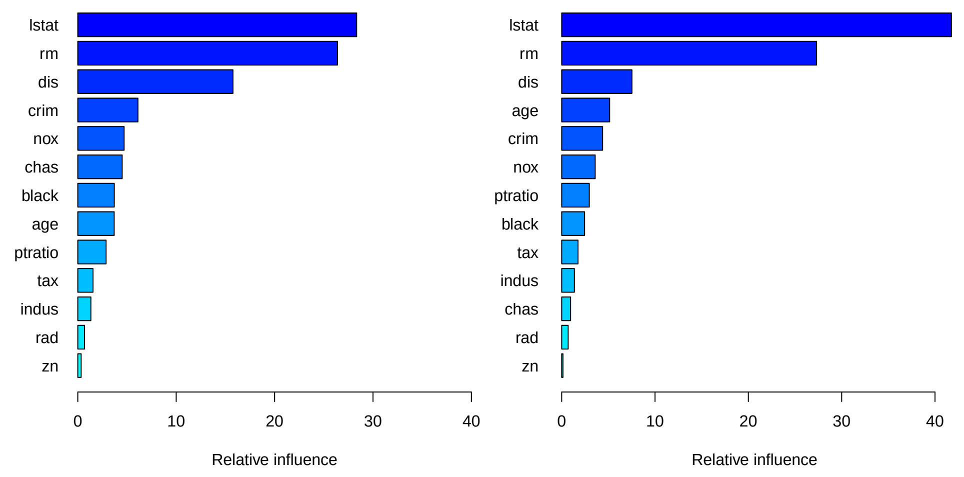
Individual trees can be extracted with `pretty.gbm.tree()`, although they are not informative by themselves.

Relative influence can be calculated for all the features. It is a percentage, and the total relative influence over all features is always 100. By default, `summary()` generates a barplot.

```
summary(bt.boston1)
summary(bt.boston1, plotit=F)$rel.inf       ## plotit=F turns off plotting
sum(summary(bt.boston1, plotit=F)$rel.inf)  ## 100
```

With `d=4`, the influence of `lstat` becomes larger than `d=1`

```
par(mfrow=c(1,2), las=1, mar=c(4,4,.5,.5))
summary(bt.boston1, xlim=c(0,40))
summary(bt.boston4)
```



Because a random subset of data is used in every step, the results vary a little from one run to another. Repeat the following to see.

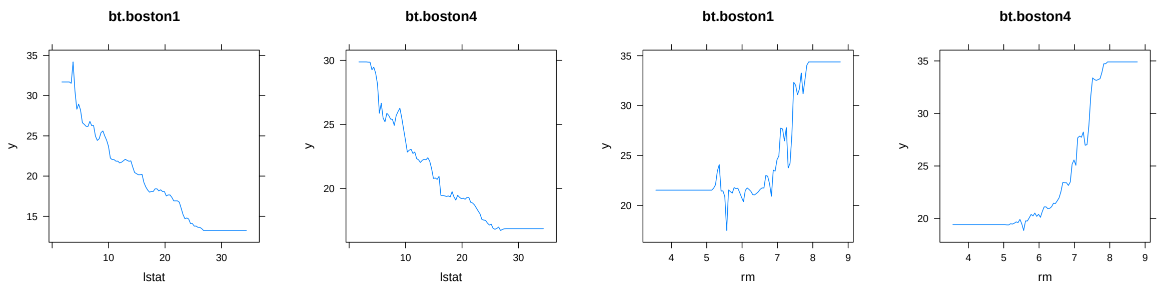
```
bt.try = gbm(medv ~ ., data=Bostontrain, distribution="gaussian", n.trees=5000)
summary(bt.try, plotit=F)$rel.inf
```

If we set `bag.fraction=1`, the process will deterministic. Repeat the following to see.

```
bt.try = gbm(medv ~ ., data=Bostontrain, distribution="gaussian", n.trees=5000, bag.fraction=1)
summary(bt.try, plotit=F)$rel.inf
```

Partial dependence plots display the marginal effect of a predictor, similar to GAM.

```
plot(bt.boston1, i="lstat", main='bt.boston1')
plot(bt.boston4, i="lstat", main='bt.boston4')
plot(bt.boston1, i="rm", main='bt.boston1')
plot(bt.boston4, i="rm", main='bt.boston4')
```



One can draw a 2-dimensional partial dependence plot, which does not offer much information for this dataset.

```
plot(bt.boston4, i=c("rm", "lstat"))
```

We can evaluate prediction performance by any number of steps in model building. In this example at depth 1, 5000 stumps seem not enough. For exampe, we look at model performance at 500, 1000, 1500, etc. trees when applied to the test set. The performance is the best at 500 trees.

```
yhat.bt1.nn = predict(bt.boston1, Bostontest, n.trees=seq(500,5000,500))
dim(yhat.bt1.nn)  ## 150 x 10
apply(yhat.bt1.nn, 2, function(x) mean((Bostontest$medv - x)^2))
```

10.3 Cross-validation using `caret`

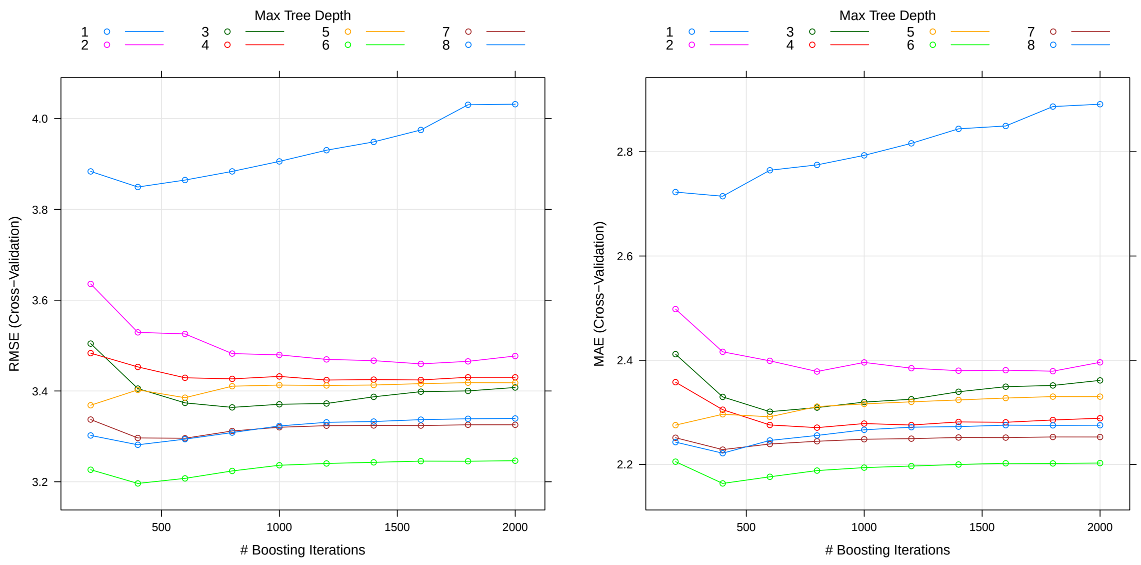
There are hyperparameters `n.trees`, `interaction.depth`, `shrinkage`, `n.minobsinnode`. By default, a grid over the hyperparameter is selected by the `train()` function. To change that, use `tuneGrid=` to specify the values in a data frame as in the following example, where we evaluate 10 tree numbers and 8 interaction depths.

```
library(caret)
set.seed(2019)
ctr = trainControl(method="cv", number=10)  ## 10-fold CV
mygrid = expand.grid(n.trees=seq(200, 2000, 200), interaction.depth=1:8,
                     shrinkage=0.1, n.minobsinnode=10)
boost.caret <- train(medv ~ ., Bostontrain, method='gbm',
                     trControl=ctr, tuneGrid=mygrid,
                     preProc=c('center','scale'), verbose=F)
```

```
boost.caret
names(boost.caret)
boost.caret$results   ## results for all combinations of hyperparameters
boost.caret$bestTune  ## best combination
boost.caret$metric    ## default metric to use for plot
```

The `caret` package can generate nice plots. The default metric is `boost.caret$metric[1]`, which is “RMSE”.

```
par(mfrow=c(1,2))
plot(boost.caret)
plot(boost.caret, metric="MAE")
```



10.4 Lasso post-processing

Boosting produces base models $\hat{g}_m$ $(m = 1, \dots, M)$, which are then shrinked and added up to be $\sum_m \epsilon \hat{g}_m$. Many of the shrinked models are similar. Alternatively, we can estimate the weights by solving a lasso problem:

$$\text{minimize}_{\{\beta_m\}} \sum_i L\left(y_i, f_0(x_i) + \sum_m \beta_m \hat{g}_m(x_i)\right) + \lambda \sum_m |\beta_m|.$$

This is implemented in the `glmnet` package. This approach is effectively regularized stacking. A variant of this is to require all weights β_m to be nonnegative.

10.5 The XGBoost method

XGBoost stands for "extreme gradient boosting". The method is actually driven more by Hessians than by gradients.

Algorithm 5: Consider a loss function $L(y, \hat{y})$ and a penalty function $\Omega(a)$ for model a.

```
(1) Initialize model: $f_0(x) = 0$.
(2) For $m = 1, \dots, M$:
    (a) Obtain model $a_m$ that minimizes $\sum_i L(y_i, f_{m-1}(x_i) + a_m(x_i)) + \Omega(a_m)$;
    (b) set $f_m(x) = f_{m-1}(x) + \epsilon a_m(x)$.
```

In this algorithm, the individual models a_m are selected with built-in regularization to mitigate overfitting. Although Algorithm 5 looks more complicated than Algorithm 4, it turns out that it is relatively straightforward when using *Hessian optimization on trees.*

XGBoost (motivation and derivation for step 2(a)): By the 2nd-order Taylor approximation, when $t \approx 0$, $L(u, v+t) \approx L(u,v) + \frac{\partial L(u,v)}{\partial v} t + \frac{1}{2}\frac{\partial^2 L(u,v)}{\partial v^2} t^2$. Then step (2a) is approximately to minimize

$$\text{obj}_m = \sum_i (g_i a_m(x_i) + \frac{1}{2} h_i a_m(x_i)^2) + \Omega(a_m),$$

where $g_i = \frac{\partial L(y_i,\gamma)}{\partial \gamma}|_{\gamma = f_{m-1}(x_i)}$ and $h_i = \frac{\partial^2 L(y_i,\gamma)}{\partial \gamma^2}|_{\gamma = f_{m-1}(x_i)}$. These values can be calculated at the end of previous iteration.

Let us only consider piecewise constant functions for a_m. A piecewise constant function over a partition of the feature space, $\mathcal{S} = \{S_1, \dots, S_J\}$, has the form $a_m(x) = w_j$ if $x \in S_j$. For piecewise constant functions, a choice of the penalty function Ω is to regularize both the partition size, J, and the l_2-norm of the coefficients:

$$\Omega(a_m) = \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2,$$

where $\gamma \geq 0$ and $\lambda \geq 0$ are tuning parameters. By penalizing both the partition size and the coefficients, we favor small trees that have small fitted values. With these, obj_m becomes

$$\text{obj}_m = \sum_{j=1}^{J} (G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2) + \gamma J,$$

where $G_j = \sum_{x \in S_j} g_i$ and $H_j = \sum_{x \in S_j} h_i$ $(j = 1, \ldots, J)$. The problem now becomes regularized Hessian optimization. Given any partition $\mathcal{S}$ of size J, the minimum of obj_m is achieved when $w_j = -\frac{G_j}{H_j+\lambda}$ $(j = 1, \ldots, J)$. This minimum is a function of $\mathcal{S}$:

$$O_m(\mathcal{S}) = -\frac{1}{2} \sum_{j=1}^{J} \frac{G_j^2}{H_j + \lambda} + \gamma J.$$

Hence given any partition, we have a piecewise constant model based on the partition that minimizes obj_m. All we need is to identify a partition that minimizes $O_m(\mathcal{S})$.

We now only consider trees. We can use $O_m(\mathcal{S})$ as the measure of impurity so that every split achieves maximum reduction of $O_m(\mathcal{S})$! When splitting a node into two subsets, denoted as L and R, the reduction in $O_m(\mathcal{S})$ is $\frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$, which is fast to calculate. We may stop growing the tree if no split can reduce $O_m(\mathcal{S})$, or grow a tree to a certain complexity and prune it.

The penalty $\Omega(a_m)$ can have a third term: $\Omega(a_m) = \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2 + \alpha \sum_{j=1}^{J} |w_j|$. In `xgboost`, by default `gamma=0`, `lambda=1`, and `alpha=0`; that is, the default penalty function is $\Omega(a_m) = \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2$. Note that the parameterization here is different from that commonly used for elastic nets: $\lambda \sum_{j=1}^{J} (\alpha |w_j| + (1-\alpha) w_j^2)$.

Notes:

- M, γ, λ, α, and ϵ are hyperparameters.
- In `xgboost`, there are other tuning options (hyperparameters). They need to be carefully selected. (see below)
- The tree depth is adaptively determined and it varies as m changes.
- The measure of impurity $O_m(\mathcal{S})$ changes as m changes.
- $w_j = -\frac{G_j}{H_j+\lambda}$ is not the average outcome for terminal node j. This is similar to gradient tree boosting.
- Because trees are grown in a greedy way, this procedure does not guarantee to yield the minimal $O_m(\mathcal{S})$ even among all trees, not to mention all partitions. But like in random forests, we do not care much about the performance of individual trees.
- **Sum of weight** for a terminal node is the Hessian of the node, $\sum_i \frac{\partial^2}{\partial \hat{y}^2} L(y_i, \hat{y})$, where $\hat{y}$ is the fitted value for the node and the summation is over the observations in the node. When $L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$, the sum of weight for a node is the number of observations in the node.

The criterion in step (2a) can be motivated as the following: We seek to build an ensemble model, $f_M(x) = \sum_{j=1}^{M} a_j(x)$, from base models $\{a_j\}$. We penalize the complexity of the final model through $\sum_{j=1}^{M} \Omega(a_j)$. That is,

$$\text{minimize} \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \sum_{j=1}^{M} \Omega(a_j), \tag{9A}$$

where $\hat{y}_i = f_M(x_i) = \sum_{j=1}^{M} a_j(x_i)$. We identify the individual base models sequentially. At step m, given $f_{m-1} = \sum_{j=1}^{m-1} a_j$, we identify model a_m so that $\sum_{i=1}^{n} L(y_i, \hat{y}_i^{(m)}) + \sum_{j=1}^{m} \Omega(a_j)$ is minimized, where $\hat{y}_i^{(m)} = f_{m-1}(x_i) + a_m(x_i)$. Since the base models $\{a_1, \cdots, a_{m-1}\}$ have already been determined, this is equivalent to minimizing $\sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + a_m(x_i)) + \Omega(a_m)$, the criterion in step (2a). Note that the criterion (9A) could be used to regularize any ensemble methods, including random forests.

10.6 The xgboost package

The current version of `xgboost` is 0.81. To install the python version, you may use `pip install xgboost`. Some demonstrations for `xgboost` are at https://github.com/dmlc/xgboost/tree/master/demo

In `xgboost` there are 3 "boosters": `gbtree` (the default), `dart`, and `gblinear`. For the first two options, trees are the base models; for `gblinear`, linear models are base models.

For `gbtree` , we can specify the following parameters (and their default values): (for a complete list and explanations, see https://xgboost.readthedocs.io/en/latest/parameter.html)

- Learning rate (`eta=0.3`);
- Regularization penalty terms (`gamma=0`, `lambda=1`, `alpha=0`);
- Stopping criteria: maximum depth (`max_depth=6`); minimum sum of weight for a child (`min_child_weight=1`);
- Fractions: fraction of data used to grow a tree (`subsample=1`); fraction of features used to to grow a tree (not to split a node as in RF) (`colsample_bytree=1`);
- `tree_method` to speed up tree growing process for big data:
  - 5 methods: `exact`, `approx`, `hist`, `gpu_exact`, `gpu_hist`;
  - `auto` is the default (`exact` or `approx` depending on dataset size);
  - When `tree_method='approx'`: `sketch_eps=0.03` contronls the number of bins;
  - When `tree_method='hist'`:
    * `grow_policy='depthwise'` controls the way new nodes are added to the tree ('depthwise': split at nodes closest to the root; 'lossguide': split at nodes with highest change of loss);
    * `max_bin=256` controls the maximum number of bins;
- For severe **class imbalance**: `max_delta_step=0`; `scale_pos_weight=1`;
- For **boosted random forest**: `num_parallel_tree=1` controls the number of parallel trees in each iteration.

"Learning Task Parameters":

- The "objective function" determines what loss function to use (`objective=reg:linear`, which means the loss function is squared error)
- The "evaluation metric" for performance evaluation (`eval_metric`, default depending on the objective function)
- `seed=0` (I do not think this is good.)

We now run `xgboost()` to build a model. The mushroom data is described here: https://archive.ics.uci.edu/ml/datasets/mushroom . The outcome is binary: poisonous or edible.

```
library(xgboost)
help(agaricus.train)
data(agaricus.train, agaricus.test)  ## load data
Xtrain = agaricus.train$data; Ytrain = agaricus.train$label
Xtest = agaricus.test$data; Ytest = agaricus.test$label
dim(Xtrain)  ## 6513 x 126
dim(Xtest)   ## 1611 x 126
table(Ytrain); table(Ytest)  ## outcome distributions
```

A **sparse matrix** is a matrix in which most of the elements are zero. Note that zeros are not NAs. A matrix that is not sparse is called a *dense matrix*. In R, the class `dgCMatrix` is a format for storing sparse matrices. A `dgCMatrix` object behaves mostly like a matrix but takes much less storage space.

```
str(agaricus.train)  ## a list containing "data" and "label"
class(Xtrain)   ## a dgCMatrix
dim(Xtrain)     ## 6513 x 126
class?dgCMatrix
colnames(Xtrain)
Matrix::colSums(Xtrain)  ## this requires the Matrix package

aa = as.matrix(Xtrain)  ## convert to a regular/dense matrix
class(aa); dim(aa)
colSums(aa)
```

```
aa[1:5, 1:4]
Xtrain[1:5, 1:4]
Xtrain[1:5, 1]
table(aa, useNA="ifany")  ## 677352 0s (83%), 143286 1s (17%)
object.size(Xtrain); object.size(aa)  ## 1.7 MB vs 6.6 MB
```

The features are coded with **one-hot encoding** (i.e., dummy variables). One-hot encoded data can be very sparse. There are actually 22 variables. We can gather their values below.

```
bb = strsplit(colnames(Xtrain), "=")
varnames = unique(unlist(purrr::map(bb, 1)))
varnames

varlevels = list()
for(ii in varnames)
  varlevels[[ii]] = unlist(purrr::map(bb,2)[ purrr::map(bb,1) == ii ])
varlevels
```

The `magrittr` package offers a very intuitive pipe syntax with the `%>%` operator. Pipes are very useful for a series of operations. The following two ways lead to the same assignment.

```
library(magrittr)
colnames(Xtrain) %>% strsplit("=") %>% purrr::map(1) %>% unlist %>% unique -> varnames
varnames = unique(unlist(purrr::map(strsplit(colnames(Xtrain), "="), 1)))
```

Now we fit an xgboost model and make predictions on the test set. The dataset is too easy for this task. The first model gives perfect prediction.

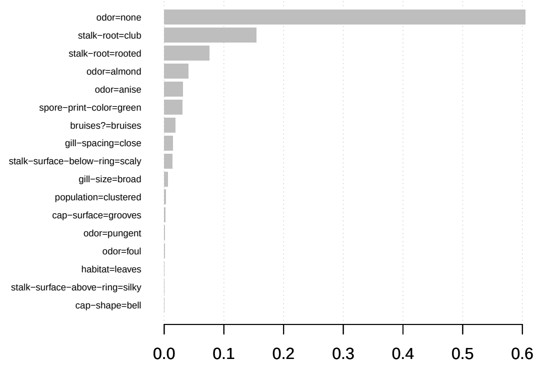
```
bst = xgboost(data=Xtrain, label=Ytrain, max_depth=4, eta=1, nround=5,
              objective="binary:logistic")
pred = predict(bst, Xtest)  ## predicted probabilities
all.equal(as.numeric(pred>0.5), Ytest)  ## TRUE
```

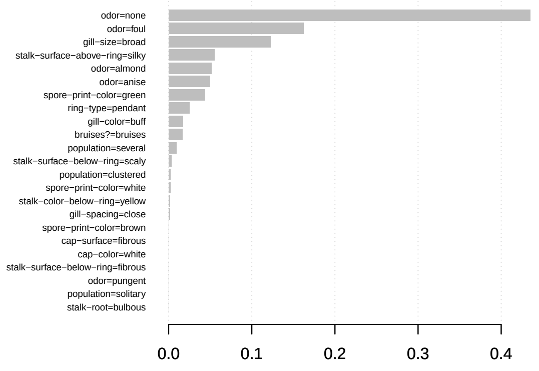
Let us slow down the learning, just to see what would happen. Try `nround=5`, `nround=50`, `nround=500`.

```
bst2 = xgboost(data=Xtrain, label=Ytrain, max_depth=1, eta=0.1, nround=500,
              objective="binary:logistic")
table(predict(bst2, Xtest) > 0.5, Ytest)
bst2 %>% predict(Xtest) %>% `>`(0.5) %>% table(Ytest)  ## the pipe way
```

Variable importance can be obtained using `xgb.importance()` and then plotted with `xgb.plot.importance()`.

```
importance_matrix = xgb.importance(model = bst)
importance_matrix
xgb.plot.importance(importance_matrix = importance_matrix)
xgb.plot.importance(importance_matrix = xgb.importance(model = bst2))
```





The model can be printed in plain text with `xgb.dump()`. The individual trees can be plotted if the `DiagrammeR` package has been installed.

```r
xgb.dump(bst, with_stats=T)  ## output/save as plain text

xgb.plot.tree(model=bst)     ## plot all trees
xgb.plot.tree(model=bst, trees=0:1)   ## first 2 trees of model bst
xgb.plot.tree(model=bst2, trees=0:3)  ## first 4 trees of model bst2

xgb.plot.multi.trees(model=bst)  ## a summary plot of multiple trees
xgb.plot.deepness(bst)  ## distribution of the depth of the leaves
```

To save this to a pdf file, one needs the `DiagrammeRsvg` and `rsvg` packages.

```r
library(DiagrammeRsvg)
aa = xgb.plot.tree(model=bst, trees=0:2, render=F)
DiagrammeR::export_graph(aa, "aa.pdf", width=288, height=432)  ## 4in x 6in because of 72 ppi
```

`xgboost` provides the `xgb.DMatrix` format as a way to hold a feature matrix and related meta data. All meta data are stored in a list named `info`. The component of the meta data named `label` is the outcome variable. To extract a component, use `getinfo()`.

```r
dtrain = xgb.DMatrix(Xtrain, label=Ytrain)
dtest = xgb.DMatrix(Xtest, label=Ytest)
getinfo(dtest, "label")  ## extracts the `label` component from `info`
```

With the `xgb.DMatrix` format, we can now use `xgb.train()` instead of `xgboost()`.

```r
watchlist = list(train=dtrain, test=dtest)
param = list(max_depth=2, eta=1, objective='binary:logistic')
bst3 = xgb.train(param, dtrain, nround=5, watchlist)
bst3 = xgb.train(param, dtrain, nround=5, watchlist, eval_metric="error", eval_metric="logloss")

names(bst3)
bst3$params
pred3 = predict(bst3, dtest)
table(Ytest, pred3>0.5)
```

We also can use `xgb.cv()` to evaluate at what iteration we should stop boosting process.

```r
bst.cv = xgb.cv(param, dtrain, nround=5, nfold=10)
names(bst.cv)

bst.cv = xgb.cv(data = dtrain, nrounds = 500, nthread = 2, nfold = 10, max_depth = 1, eta = 0.1,
                metrics = list("rmse","auc"), objective = "binary:logistic")
print(bst.cv, verbose=T)
names(bst.cv$evaluation_log); dim(bst.cv$evaluation_log)
matplot(bst.cv$evaluation_log[,c(2,6)], type='l')
```

10.7 Notes

Other R packages for boosting:

- `caret` (document) provides an interface for using `gbm`, `xgboost` and a few other boosting packages. Details are here. `names(getModelInfo())` provides a list of all trainable model types.
- `gbm3` was started because the `gbm` package seemed to stop development (it is still maintained). `gbm3` is not backward compatible with `gbm`. It can be installed with `devtools::install_github("gbm-developers/gbm3")`.
- `LightGBM` is an alternative to `xgboost`. It is developed by Microsoft. It can be installed with `devtools::install_github("Microsoft/LightGBM", subdir = "R-package")`. The competition between

xgboost and LightGBM has improved both. Their performance is similar now (see https://github.com/dmlc/xgboost/issues/1950 and https://github.com/Microsoft/LightGBM/blob/master/docs/Experiments.rst)

Milestones of Boosting:

- Schapire (1990, Mach Learning) on feasibility of boosting;
- Freund and Schapire (1996; 1997, J Comp System Sci) on AdaBoost (2003 ACM SIGACT Gödel Prize; 2004 ACM Paris Kanellakis Award);
- Breiman (1996; 1997, Ann Stat) on "arcing";
- Friedman (1999; 2001, Ann Stat) on gradient boosting machine;
- Chen and Guestrin (2014; 2016 Proc KDD16) on XGBoost (2016 ASA John Chambers Award to He and Chen).
- Some KDD and Kaggle.com competitions were won by using various versions of boosting and XGBoost.
- Viola–Jones detector (2001) for real-time face detection used AdaBoost (2011 IEEE Computer Society Longuet–Higgins Prize).

10.8 Assignment

1. **Homework**: ISLR has a dataset `Khan`. It contains gene expression data for 4 types of small round blue cell tumors. Use `help(Khan)` to see details. Apply both random forests and tree boosting to the training set, and tuning the hyperparameters to improve the models. Report your main steps and final results.