9 PQHS 471 Notes Week 9

9.1 Week 9 Day 1

9.1.1 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.

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* $e_m = \sum w_i I(y_i \neq g_m(x_i)) / \sum w_i$, and *weight multiplier* $f_m = (1 - e_m)/e_m$;
  (c) update weight $w_i \leftarrow w_i$ if $y_i = g_m(x_i)$ and $w_i \leftarrow w_i f_m$ if $y_i \neq g_m(x_i)$.
(3) The final model is $g(x) = \text{sign}(\sum_m \alpha_m g_m(x))$, with *model weight* $\alpha_m = \log(f_m)$ for $g_m(x)$.

Notes:

- 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 $e_m > 0.5$, $f_m < 1$, $\alpha_m < 0$.
- The total weight $\sum_i w_i = 1$ at step 1, but increases for later steps. This is okay because the focus is on obtaining a prediction model, not on accuracy of parameter estimation nor hypothesis testing.

Algorithm 2 for squared error loss (ISLR Algorithm 8.2):

(1) Initialize model $f_0(x) = \bar{y}$.
(2) For $m = 1, \ldots, M$:
  (a) fit the residuals $r_i = y_i - f_{m-1}(x_i)$ to the features to obtain a model g_m;
  (b) update the model $f_m(x) = f_{m-1}(x) + \epsilon g_m(x)$.

Notes:

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

Algorithm 3: **Gradient boosting** generalizes Algorithm 2 to any loss function $L(y, \hat{y})$. **Pseudo-residuals** are defined as gradients $r_{im} = -\frac{\partial L(y_i, \gamma)}{\partial \gamma}|_{\gamma = f_{m-1}(x_i)}$. For squared error loss $L = \frac{1}{2}(y - \hat{y})^2$, its gradient, $-\frac{\partial L}{\partial \hat{y}} = y - \hat{y}$, is the traditional residual.

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

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 $f_m(x) = f_{m-1}(x) + \epsilon g(x; \gamma_m)$.

Notes:

- 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.

9.1.2 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.

When boosting trees, the interaction depth d controls the complexity (d-way interactions) of individual trees. When $d = 1$, we boost *stumps* (i.e., trees with a single split), and the final model is **an additive model fit in a fully adaptive way** in that it does variables 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.

Friedman's **gradient tree boosting**: 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) fit a tree with maximum interaction depth d to the targets r_{im} to obtain terminal nodes R_{jm} ($j = 1, \ldots, d+1$);
   (b) for terminal node j, compute $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma))$.
   (c) set $f_m(x) = f_{m-1}(x) + \sum_j \gamma_{jm} I(x \in R_{jm})$.

Notes:

- In (2b), 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 further improve the performance by using a subset of data in (2a).

Lasso post-processing: Boosting produces base models $\hat{g}_m$ ($m = 1, \ldots, M$), 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: $\text{minimize}_{\{\beta_m\}} \sum_i L\left[y_i, \sum_m \beta_m \hat{g}_m(x_i)\right] + \lambda \sum_m |\beta_m|$. A variant of this is to require all weights β_m to be nonnegative. Implemented in `glmnet`. (This is effectively regularized stacking.)

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. By default, the number of trees is `n.trees=100`, the interaction depth is `interaction.depth=1`, the minimum number of observations in ternimal node is `n.minobsinnode=10`, and the shrinkage is `shrinkage=0.001`. By default, the trees are grown on a subset of 50% randomly selected observations from the training data (`bag.fraction=0.5`). This speeds up computations and results in some variance reduction. But the results will vary a little from one run to another.

```
library(MASS)
help(Boston)
str(Boston)
hist(Boston$medv, breaks=seq(5,52,2))

## split data into 70% training data and 30% testing data
library(caret)
set.seed(2018)
split = 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

library(gbm)
bt.boston1 = gbm(medv ~ ., data=Bostontrain, distribution="gaussian", n.trees=5000)
bt.boston1
names(bt.boston1)
bt.boston1$interaction.depth  ## stumps
bt.boston1$cv.folds  ## no CV was done
```

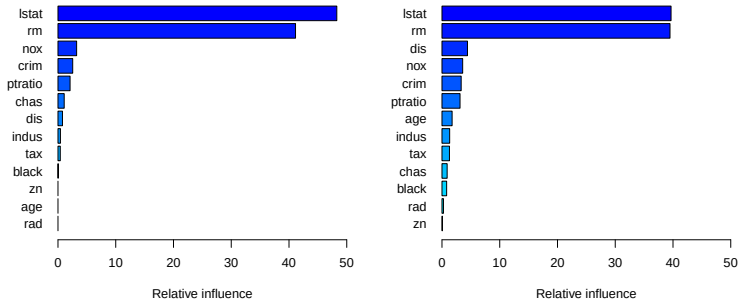
For this dataset, deeper trees give a better Prediction performance on the test set.

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

mse = function(a,b) mean((a-b)^2)
mse(Bostontest$medv, predict(bt.boston1, Bostontest, n.trees=5000))  ## MSE=16.8
mse(Bostontest$medv, predict(bt.boston4, Bostontest, n.trees=5000))  ## MSE=11.8
```

Individual trees can be extracted using `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.

```
summary(bt.boston1)  ## results and a plot
summary(bt.boston4)  ## with d=4, the influence of lstat is smaller
summary(bt.boston1, plotit=F)  ## without the plot

sum(summary(bt.boston1, plotit=F)$rel.inf)  ## 100
sum(summary(bt.boston4, plotit=F)$rel.inf)  ## 100
```

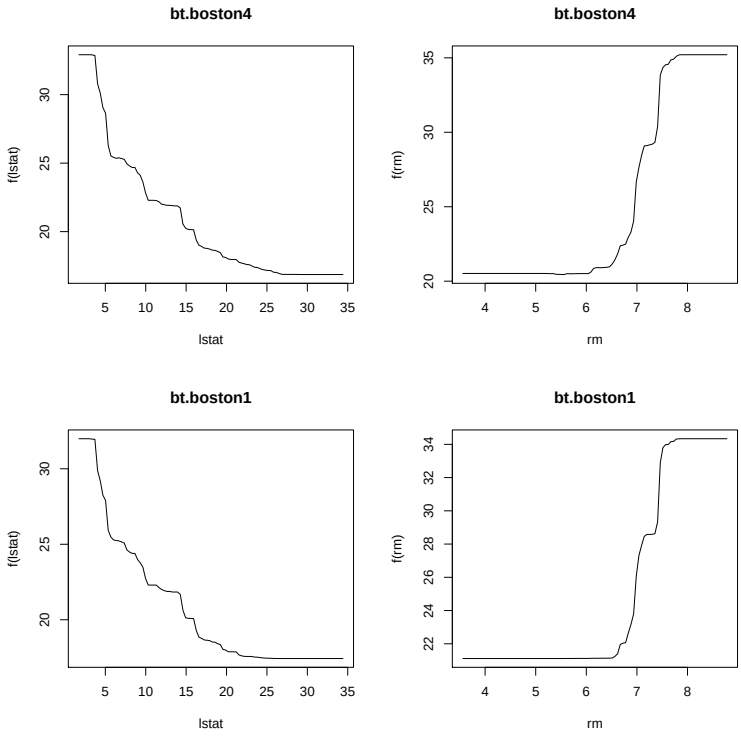


Now we check if setting `bag.fraction=1` would lead to deterministic results. (The answer is yes.)

```
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.

```
par(mfrow=c(2,2))
plot(bt.boston4, i="lstat", main='bt.boston4')
plot(bt.boston4, i="rm", main='bt.boston4')
plot(bt.boston1, i="lstat", main='bt.boston1')
plot(bt.boston1, i="rm", main='bt.boston1')
```



One can draw a 2-dimensional partial dependence plot.

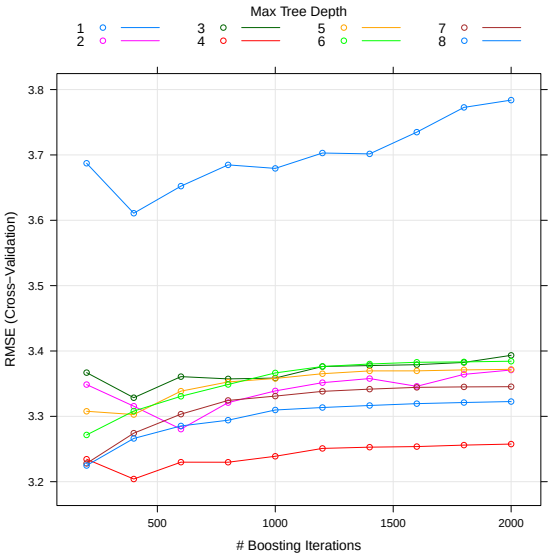
```
plot(bt.boston4, i=c("rm", "lstat"))
```

We can evaluate prediction performance by any number of steps in model building. In this example at depth 1, 5000 stumps seem not enough.

```
## look at model performance at the end of 1000, 2000, etc. trees.
yhat.bt1.nn = predict(bt.boston1, Bostontest, n.trees=seq(1000,5000,1000))
dim(yhat.bt1.nn)
apply(yhat.bt1.nn, 2, function(x) mse(Bostontest$medv, x))

bt.boston4b = gbm(medv ~ ., data=Bostontrain, distribution="gaussian", n.trees=5000, interaction.depth=4,
mse(Bostontest$medv, predict(bt.boston4b, Bostontest, n.trees=5000))
```

Cross-validation using `caret` to select the 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.

```
library(caret)
set.seed(2018)
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, trControl=ctr, method='gbm',
                     tuneGrid=mygrid,
                     preProc=c('center','scale'), verbose=F)

boost.caret
plot(boost.caret)
```



Using the optimal hyperparameters selected by `train()` improves the result!

```
boost.caret$bestTune
mse(Bostontest$medv, predict(boost.caret, Bostontest))  ## MSE=10.8

names(boost.caret)
boost.caret$results
boost.caret$finalModel
```

9.1.3 XGBoost

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 $m = 1, \ldots, M$:
  (a) Obtain 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 are selected with built-in regularization to mitigate overfitting (and to control what we regularize).

XGBoost: Although Algorithm 5 is more complicated than Algorithm 4, it is straightforward to calculate when using the Hessian optimization on trees. By the 2nd-order Taylor approximation, when t is close to zero, $L(u, v + t) \approx L(u, v) + \frac{\partial L(u,v)}{\partial v} t + \frac{1}{2} \frac{\partial^2 L(u,v)}{\partial v^2} t^2$. 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 step functions for a_m. A step function over a partition of the feature space with J subsets, $\mathcal{S} = \{S_1, \ldots, S_J\}$, has the form $a_m(x) = w_j$ if $x \in S_j$. For step functions, a good choice of Ω is

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

where $\gamma \geq 0$ and $\lambda \geq 0$ are tuning parameters. 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 $H_j = \sum_{x \in S_j} h_i$. Given a partition $\mathcal{S}$, the minimum of obj_m is $O(\mathcal{S}) = -\frac{1}{2}\sum_{j=1}^{J} \frac{G_j^2}{H_j+\lambda} + \gamma J$, which occurs at $w_j = -\frac{G_j}{H_j+\lambda}$ $(j = 1, \ldots, J)$. (This is effectively a regularized Hessian optimization.)

Now we need to identify the partition that minimizes $O(\mathcal{S})$. We only consider trees, and use $O(\mathcal{S})$ as the measure of impurity for tree growing so that every split achieves maximum reduction of $O(\mathcal{S})$. When splitting a leaf into two subsets, denoted as L and R, the reduction in $O(\mathcal{S})$ (also called the *gain*) 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 could stop growing the tree if no split can reduce $O(\mathcal{S})$, or grow a tree to a certain complexity and prune back.

Notes:

- M, γ, λ, and ϵ are hyperparameters.
- In `xgboost`, many other tuning options are available. So, many other hyperparameters may need to be carefully selected.
- Because trees are grown in a greedy way, this procedure does not guarantee to yield the minimal $O(\mathcal{S})$ even among all trees, not to mention all partitions, but the result is often close.
- w_j is not the average for terminal node j. This is similar to gradient tree boosting.
- The penalty $\Omega(a_m)$ can have a third term: $\Omega(a_m) = \gamma J + \frac{1}{2}\lambda\sum_{j=1}^{J} w_j^2 + \alpha\sum_{j=1}^{J}|w_j|$. It penalizes both the partition size J and the coefficients w_j. It favors small trees that have small fitted values. In `xgboost`, by default `gamma=0`, `lambda=1`, and `alpha=0`. Note that this is different from the parameterization commonly used for elastic nets: $\lambda\sum_{j=1}^{J}(\alpha|w_j| + (1-\alpha)w_j^2)$. Details in https://towardsdatascience.com/boosting-algorithm-xgboost-4d9ec0207d
- The **sum of weight** for a terminal node with a fitted value $\hat{y}$ is the Hessian of the node, $\sum_i \frac{\partial^2}{\partial \hat{y}^2} L(y_i, \hat{y})$, where 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 algorithm can also be motivated in the following way: We seek to build an ensemble of models $\{a_j\}$ to

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

where $\hat{y}_i = \sum_{m=1}^{M} a_m(x_i)$. At step m, given $f_{m-1} = \sum_{j=1}^{m-1} a_j$, we seek to identify 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 all previous individual models a_j are fixed, 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) can be used to regularize any ensemble methods, including random forests.

`xgboost` version 0.7 has more features than the current version (0.6.4.1) on CRAN. To install version 0.7, use the following steps:

```
git clone --recursive https://github.com/dmlc/xgboost
cd xgboost
git submodule init
git submodule update
cd R-package
R CMD INSTALL .
```

Some demonstrations for `xgboost` are at https://github.com/dmlc/xgboost/tree/master/demo

In `xgboost` version 0.7, there are 3 boosters: `gbtree`, `dart`, and `gblinear`. For `gbtree` (the default), we can specify the following parameters (their defaults in parentheses):

- learning rate (`eta=0.3`);
- minimum loss reduction (`gamma=0`);
- maximum depth (`max_depth=6`);
- minimum sum of weight for a child (`min_child_weight=1`);
- subsample ratio (`subsample=1`; setting it to 0.5 means randomly select half of the data to grow a tree);
- subsample ratio of features (`colsample_bytree=1`; setting it to 0.3 means randomly select 30% of the features to grow a tree);

- subsample ratio of features per split (`colsample_bylevel=1`);
- regularization term (`lambda=1`);
- regularization term (`alpha=0`; l_2 only);
- maximum delta step in weight estimation (`max_delta_step=0`; used for severe class imbalance);
- balance of positive and negative weights (`scale_pos_weight=1`; used for severe class imbalance);
- `tree_method` can take values `auto`, `exact`, `approx`, `hist`, `gpu_exact`, `gpu_hist` (default `auto`).
  - For non-exact methods, `sketch_eps` is applicable.
  - 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 loss change);
    - `max_leaves=0`, only relevant for the 'lossguide' grow policy;
    - `max_bin=256`.
- steps to run (`updater='grow_colmaker,prune'` grows a tree to maximum depth and then prune back to gamma?)
- Others: `updater='grow_colmaker,prune'`, `refresh_leaf=1`, `process_type='default'`, `predictor='cpu_predictor'`.

The available Learning Task Parameters:

- The objective function (`objective=reg:linear`)
- Initial prediction score (`base_score=0.5`)
- `eval_metric` (various default according to the objective function)
- `seed=0`

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*). The demonstration below involves data of class `dgCMatrix`, a format for storing sparse matrices. We take a look at the dgCMatrix class first.

```
library(Matrix)
data(agaricus.train, package='xgboost')
str(agaricus.train)  ## a list containing "data" and "label"
Xtrain = agaricus.train$data   ## a dgCMatrix
Ytrain = agaricus.train$label  ## a numeric vector
class?dgCMatrix

colnames(Xtrain)
dim(Xtrain)  ## 6513 x 126
colSums(Xtrain)

aa = as.matrix(Xtrain)  ## convert to a regular/dense matrix
class(aa); dim(aa)
table(aa, exclude=NULL)  ## 677352 0s (83%), 143286 1s (17%)
aa[1:5, 1:4]
Xtrain[1:5, 1:4]
Xtrain[1:5, 1]
object.size(Xtrain); object.size(aa)  ## 1.7 MB vs 6.6 MB
```

Now run `xgboost()` to build a model. The mushroom data is described here: https://archive.ics.uci.edu/ml/datasets/mushroom

```
library(xgboost)
help(agaricus.train)
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
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)
```

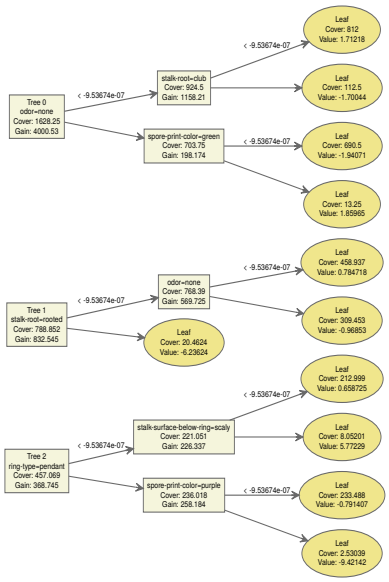
```
bst = xgboost(data=Xtrain, label=Ytrain, max_depth=4, eta=1, nround=5,
              objective="binary:logistic")
pred = predict(bst, Xtest)  ## predicted probabilities
all.equal(as.numeric(pred>0.5), Ytest)  ## perfect prediction on test set

## verbose level can be 0, 1, 2 (default is 1)
bst = xgboost(data=Xtrain, label=Ytrain, max_depth=2, eta=1, nround=5,
              objective="binary:logistic", verbose=2)
```

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

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

The model can be printed in plain text. The individual trees can be plotted.

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

library(DiagrammeR)
xgb.plot.tree(model=bst)
xgb.plot.tree(model=bst, trees=0:2)  ## first 3 trees
```



To save this to a pdf file:

```
library(DiagrammeRsvg)
aa = xgb.plot.tree(model=bst, trees=0:2, render=F)
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 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()`; e.g., `getinfo(dtest, "label")` extracts the `label` component from `info`.

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

```
dtrain = xgb.DMatrix(Xtrain, label=Ytrain)
dtest = xgb.DMatrix(Xtest, label=Ytest)

param = list(max_depth=2, eta=1, objective='binary:logistic')
watchlist = list(train=dtrain, test=dtest)
```

```
bst2 = xgb.train(param, dtrain, nround=5, watchlist)
bst2 = xgb.train(param, dtrain, nround=5, watchlist, eval_metric="error", eval_metric="logloss")

names(bst2)
bst2$params
pred = predict(bst2, dtest)

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

Linear model boosting by setting `booster="gblinear"` (its default is `gbtree`, for tree boosting).

```
bst = xgb.train(param, dtrain, booster="gblinear", nrounds=5, watchlist)
```

Notes on other R packages for boosting:

- `caret` (document) provides a unified interface (and a wrapper) for using `gbm`, `xgboost` and many other R packages. `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 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).

9.1.4 HOML 6

9.1.5 HOML 7

9.1.6 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 random forest and boosting to the training set, and tuning the hyperparameters to improve the models. Report your main steps and final results.
2. Reading for next lecture: ISLR 9; HOML Chapter 5
3. ISLR Chapter 9 R Labs