Code

# Boosting

**Ruoqing Zhu**

**Last Updated: October 24, 2024**

# Boosting a Linear Regression

In recent years `xgboost` becomes extremely popular. The main advantage is the computational performances. We will first demonstrate the idea of boosting by setting a connection with Lasso with linear weak learner. In this case, the model is similar to a stage-wise regression (https://teazrq.github.io/stat432/RNotes/Lasso/Lasso.html#Forward-stagewise_Regression_and_the_Solution_Path). After that, we demonstrate the tree boosting idea. Let's consider fitting a regression model by aggregating a collection of "small" models

$$F_T(x) = \sum_{t=1}^{T} \alpha_t f_t(x)$$

Here we fit $T$ models, where each small model $f_t(x)$ receive a weight $\alpha_t$. Here each $f_t(x)$ is constructed progressively, meaning that we first fit $f_1(x)$ and then figure out what would be the next most beneficial function $f_2(x)$ to add on top of $f_1(x)$, and iterate that process. This is different from random forests in which each $f_t(x)$ are built independently and parallelly, with weight $\alpha_t = 1/T$. For a linear boosting model, we are interested fitting a regression in the form of

$$Y \sim \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p.$$

However, we will not solve all parameters in just one step. Instead, we may view each step $f_t(x)$ as growing one $\beta_j$ parameter by a small magnitude $\alpha_t$, say $\alpha_t = 0.0001$. However, different $f_t(x)$ on different $t$ may involve different variable $j$.

Assuming the observed data as $\{y_i, x_i\}_{i=1}^n$, where $x_i$ is a $p$-dimensional covariate vector. At step $t = 1$, we initiate all $\beta_j$'s as 0. Hence the current residual is

$$\epsilon_i = y_i - 0 = y_i,$$

Then, let's find the variable with the highest absolute correlation with the current residual:

$$j_* = \arg\max_j \ |\text{corr}(\boldsymbol{\epsilon}, \mathbf{x}^{(j)})|$$

where $\mathbf{x}_j = (x_1^{(j)}, x_2^{(j)}, \dots, x_n^{(j)})^T$ is the $n$-vector of the $j$th variable, and $\boldsymbol{\epsilon} = (\epsilon_1, \epsilon_2, \dots, \epsilon_n)$ is the $n$-vector of residuals. We can then use

$$f_1(x) = \widehat{\beta}^{\text{ols}}_{j_*} x^{(j_*)}$$

where $\widehat{\beta}^{\text{ols}}_{j_*}$ is the OLS estimator that regresses the current residual $\boldsymbol{\epsilon}$ on $x^{(j_*)}$. But this is too aggressive. Hence, consider applying a step size $\alpha_t$ to it, which will effectively make the growth of the $j_*$'th parameter small. This completes the first step $t = 1$. Now, since we explained a bit of variation in the outcome, let's remove that from the outcome and define the residual of each observations $i$ in $t = 2$ as

$$\boldsymbol{\epsilon}_i = y_i - \alpha_1 f_1(x_i).$$

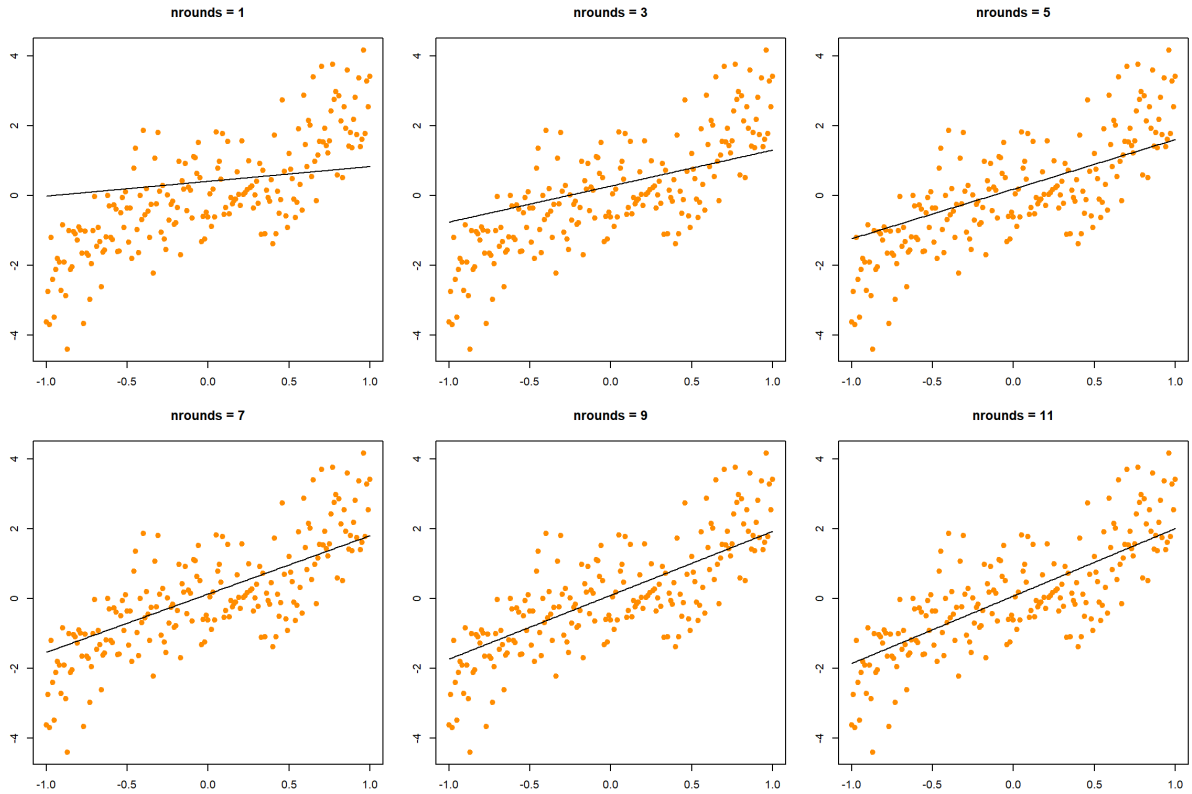
We then find the next best $j_*$ to boost $\beta_{j_*}$ by a small magnitude. In general, at the $(t+1)$th step, we will always calculate residuals as

$$\epsilon_i = y_i - \sum_{k=1}^{t} \alpha_t f_k(x_i)$$

You can imaging that by doing this, we alternate the growth of $\beta_j$'s in many small steps. The following code demonstrate this in a one-dimensional problem. Since there is no other $j$ to alternate, we simply grow the slope in a steady way.

Hide

```r
library(xgboost)
set.seed(1)
p = 1
x = seq(-1, 1, 0.01)
fx <- function(x) 1*x + 2*x^3
y = fx(x) + rnorm(length(x))

# construct data for xgboost
traindata = xgb.DMatrix(data = data.matrix(x), label = y)

# we use linear booster with a step size 0.2
param = list(booster = "gblinear", eta = 0.2)

par(mfrow = c(2, 3))
par(mar = c(2, 2, 4, 2))

# try different number of T
for (nrounds in c(1, 3, 5, 7, 9, 11))
{
  xgb.fit = xgb.train(param, traindata, nrounds = nrounds)

  pred_y = predict(xgb.fit, traindata)
  plot(x, y, pch = 19, col="darkorange", main = paste("nrounds =", nrounds))
  points(x, pred_y, type = "l")
}
```



# Gradient Boosting View

In a more rigorous way, the boosting algorithm is trying to minimize the loss function by iteratively fitting models to the "residual". It should be easy to see that, if we use Mean Squared Error as the loss function in the regression case, then for a single instance, it is given by:

$$L(y, F(x)) = (y - F(x))^2$$

Let's look at the partial derivative of this loss function with respect to $F(x)$:

$$\frac{\partial L(y, F(x))}{\partial F(x)} = -2(y - F(x))$$

In the context of gradient boosting, the pseudo-residual $r$ for each instance $i$ is the negative gradient of the loss function evaluated at the current prediction $F(x)$:

$$r_i = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = 2(y_i - F(x_i))$$

Often, this is simplified further to:

$$r_i = y_i - F(x_i)$$

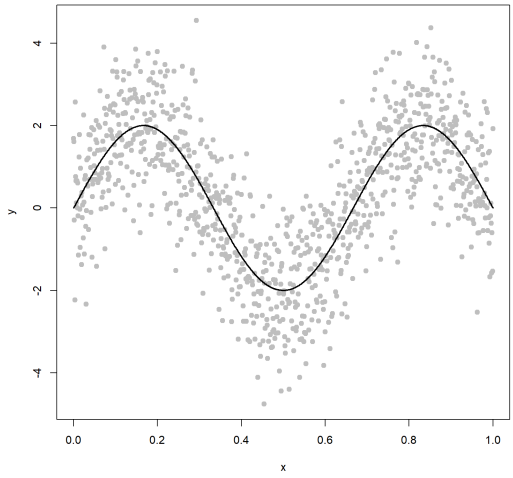
Here, $F(x_i)$ is the current prediction for instance $i$ before the new weak learner is added to the ensemble.

# Tree Boosting

Tree boosting is not really much different, expect that at each step, we can fit a regression tree model based on the current residual. We may multiply the tree model by a small magnitude $\eta$ to prevent growing the model too fast. We will use the `gbm` package as an example. You can further control the complexity of trees through tuning parameters.

Hide

```
  library(gbm)
## Loaded gbm 2.2.2
## This version of gbm is no longer under development. Consider transitioning to gb
m3, https://github.com/gbm-developers/gbm3

  # a simple regression problem
  p = 1
  x = seq(0, 1, 0.001)
  fx <- function(x) 2*sin(3*pi*x)
  y = fx(x) + rnorm(length(x))

  plot(x, y, pch = 19, ylab = "y", col = "gray")
  lines(x, fx(x), lwd = 2) # plot the true regression line
```

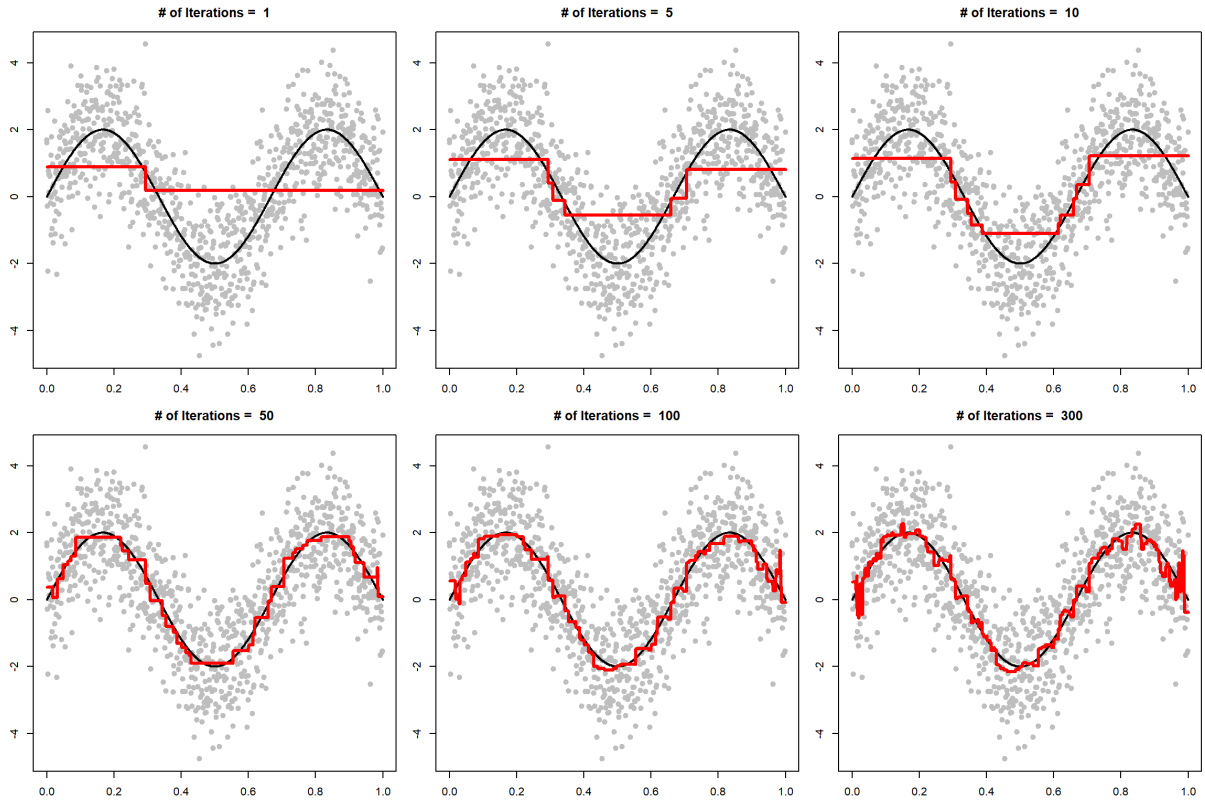



Hide

```
# fit regression boosting
# I use a very large shrinkage value for demonstrating the functions
# in practice you should use 0.1 or even smaller values for stability
gbm.fit = gbm(y~x, data = data.frame(x, y), distribution = "gaussian",
              n.trees=300, shrinkage=0.5, bag.fraction=0.8)

# somehow, cross-validation for 1 dimensional problem creates error
# gbm(y ~ ., data = data.frame(x, y), cv.folds = 3) # this produces an error
```

Hide

```
# plot the fitted regression function at several iterations
par(mfrow=c(2,3))
size=c(1,5,10,50,100,300)

for(i in 1:6)
{
  par(mar=c(2,2,3,1))
  plot(x, y, pch = 19, ylab = "y", col = "gray")
  lines(x, fx(x), lwd = 2)

   Fx = predict(gbm.fit, n.trees=size[i]) # this returns the fitted function, but
not class
  lines(x, Fx, lwd = 3, col = "red")
  title(paste("# of Iterations = ", size[i]))
}
```




# Gradient Boosting for Classification

For classification problems, we do not directly have the residual. However, we could think of the residual as coming from the contribution to the likelihood function, so the goal becomes minimizing the negative log-likelihood (analog to $\ell_2$ loss). So let's use the logistic link function the predicted probability $p$ is often defined as:

$$p = \frac{1}{1 + e^{-F(x)}}$$

The negative log-likelihood for the Bernoulli distribution for a single instance is given by:

$$L(y, p) = -[y \log(p) + (1 - y) \log(1 - p)]$$

Let's first find the partial derivative of this loss function with respect to $p$:

$$\frac{\partial L(y, p)}{\partial p} = -\left[y \frac{1}{p} - (1 - y) \frac{1}{1 - p}\right]$$

The derivative of $p$ with respect to $F(x)$ is:

$$\frac{\partial p}{\partial F(x)} = \frac{e^{-F(x)}}{(1 + e^{-F(x)})^2} = p(1 - p)$$

Hence, the partial derivative of the loss function with respect to $F$ is:

$$
\begin{aligned}
\frac{\partial L(y,p)}{\partial F(x)} &= \frac{\partial L(y,p)}{\partial p} \cdot \frac{\partial p}{\partial F(x)} \\
&= -\left[ y\frac{1}{p} - (1-y)\frac{1}{1-p} \right] \cdot p(1-p) \\
&= -(y - yp + p - yp) \\
&= -(y-p)
\end{aligned}
$$

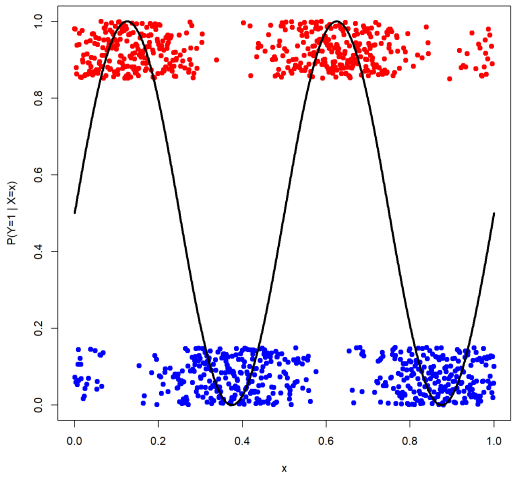
The $y_i - p_i$ is the pseudo-residual that we use in the boosting algorithm to fit the next tree / linear booster. The sign change is because we move to the negative gradient in a minimization problem.

# AdaBoost for Binary Classification

AdaBoost is a boosting model for classification. The idea is slightly different from regression since we cannot directly calculate the residuals. Instead, Adaboost defines weights on each observation, and progressively update weights. Note that observations with larger weights is effectively similar as observations with large residuals, such that the $f_t(x)$ in the next iteration will try to address them (fit them bitter). The following code demonstrate AdaBoost.

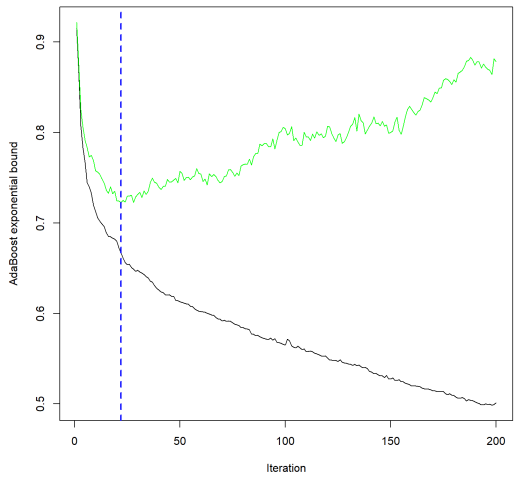
Hide

```
  n = 1000; set.seed(1)
  x = cbind(seq(0, 1, length.out = n), runif(n))
  py = (sin(4*pi*x[, 1]) + 1)/2
  y = rbinom(n, 1, py)

  plot(x[, 1], y + ifelse(y == 1, runif(n, -0.15, 0), runif(n, 0, 0.15)),
       pch = 19, col = ifelse(y==1, "red", "blue"), xlab = "x", ylab = "P(Y=1 | X=
x)")
  points(x[, 1], py, type = "l", lwd = 3)
```




Hide

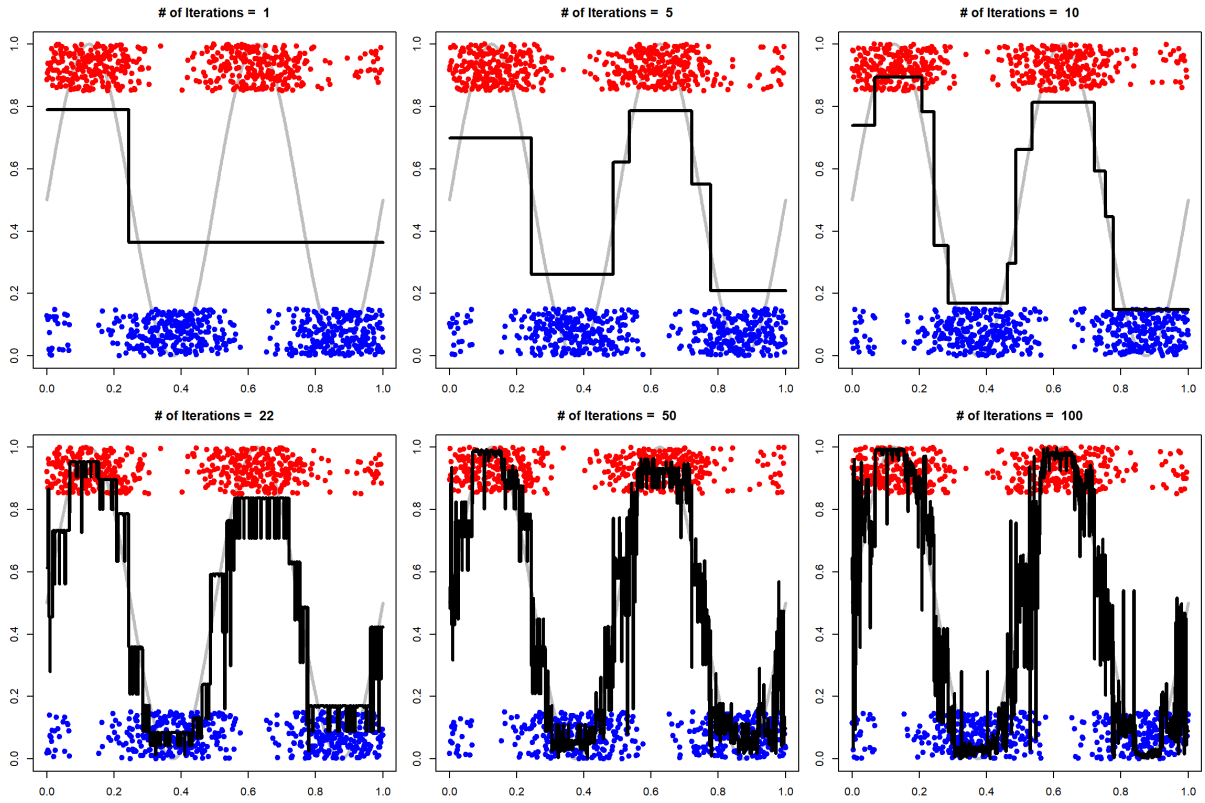
```
  # fit AdaBoost with bootstrapping, again I am using a large shrinkage factor

  gbm.fit = gbm(y~., data.frame(x, y), distribution="adaboost", n.minobsinnode = 2,
                n.trees=200, shrinkage = 1, bag.fraction=0.8, cv.folds = 10)

  # get the best number of trees from cross-validation (or oob if no cv is used)
  gbm.perf(gbm.fit)
```




```
## [1] 22
```

Hide

```
  # plot the decision function (Fx, not sign(Fx))
  par(mfrow=c(2,3))
  size=c(1, 5, 10, 22, 50, 100)

  for(i in 1:6)
  {
    par(mar=c(2,2,3,1))
    plot(x[, 1], py, type = "l", lwd = 3, ylab = "P(Y=1 | X=x)", col = "gray")
    points(x[, 1], y + ifelse(y == 1, runif(n, -0.15, 0), runif(n, 0, 0.15)),
           pch = 19, col = ifelse(y==1, "red", "blue"))
     Fx = predict(gbm.fit, n.trees=size[i]) # this returns the fitted function, but
not class
    lines(x[, 1], 1/(1+exp(-2*Fx)), lwd = 3)
    title(paste("# of Iterations = ", size[i]))
  }
```




Hide

```
# You can peek into a single tree
pretty.gbm.tree(gbm.fit, i.tree = 1)
```

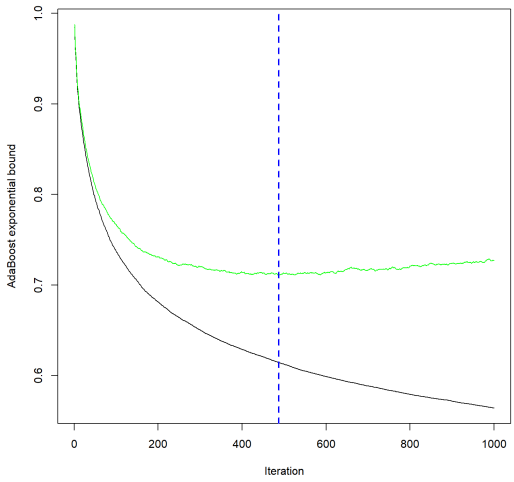
| | SplitVar <int> | SplitCodePred <dbl> | LeftNode <int> | RightNode <int> | MissingNode <int> | ErrorReduction <dbl> | Wei... <dbl> |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0.24374374 | 1 | 2 | 3 | 129.2909 | 800 |
| 1 | -1 | 0.68000925 | -1 | -1 | -1 | 0.0000 | 188 |
| 2 | -1 | -0.26108319 | -1 | -1 | -1 | 0.0000 | 612 |
| 3 | -1 | -0.03992647 | -1 | -1 | -1 | 0.0000 | 800 |

4 rows | 1-8 of 9 columns

# Tuning parameter: `shrinkage`

Hide

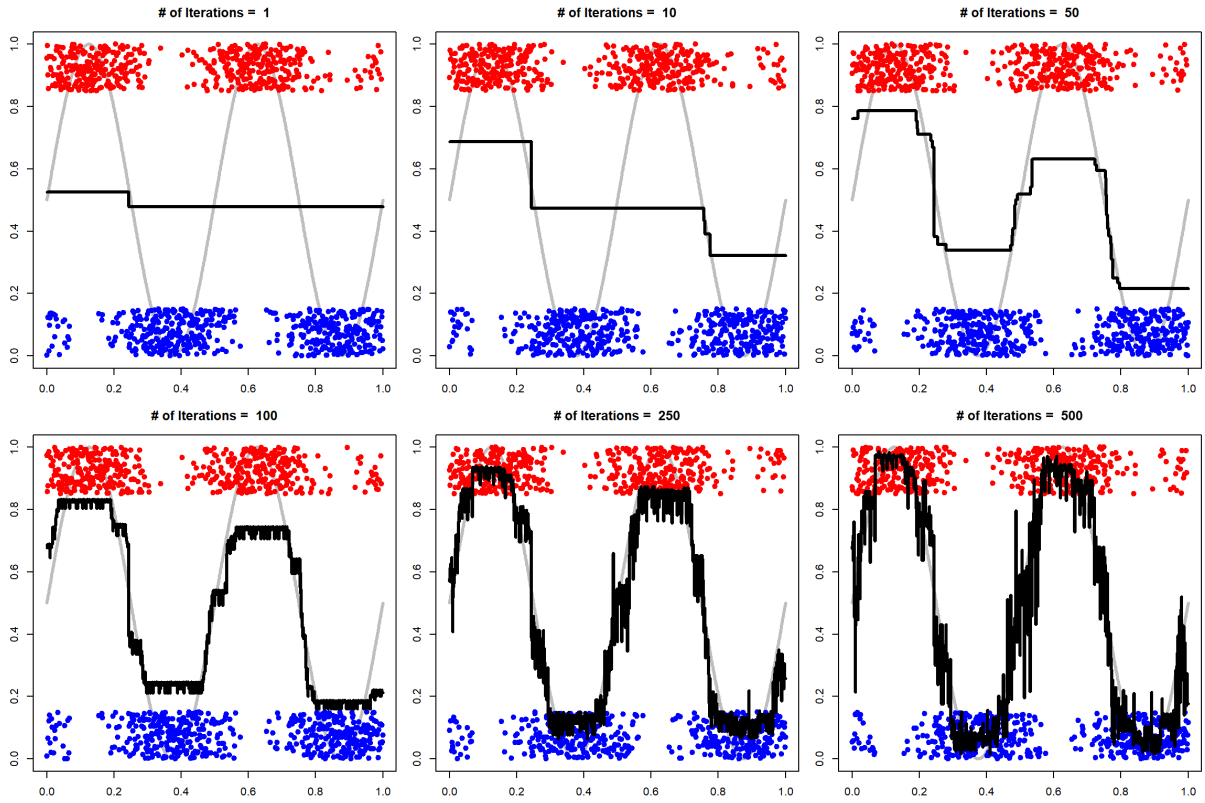
```
# you may use a very small shrinkage factor to improve stability
# however, this means you will need to use more trees

gbm.fit = gbm(y~., data.frame(x, y), distribution="adaboost", n.minobsinnode = 2,
              n.trees=1000, shrinkage = 0.1, bag.fraction=0.8, cv.folds = 10)
gbm.perf(gbm.fit)
```




```
## [1] 487
```

Hide

```
  par(mfrow=c(2,3))
  size=c(1,10,50,100,250,500) # 1000 trees is clearly over-fitting

  for(i in 1:6)
  {
    par(mar=c(2,2,3,1))
    plot(x[, 1], py, type = "l", lwd = 3, ylab = "P(Y=1 | X=x)", col = "gray")
    points(x[, 1], y + ifelse(y == 1, runif(n, -0.15, 0), runif(n, 0, 0.15)),
           pch = 19, col = ifelse(y==1, "red", "blue"))
     Fx = predict(gbm.fit, n.trees=size[i]) # this returns the fitted function, but
not class
    lines(x[, 1], 1/(1+exp(-2*Fx)), lwd = 3)
    title(paste("# of Iterations = ", size[i]))
  }
```



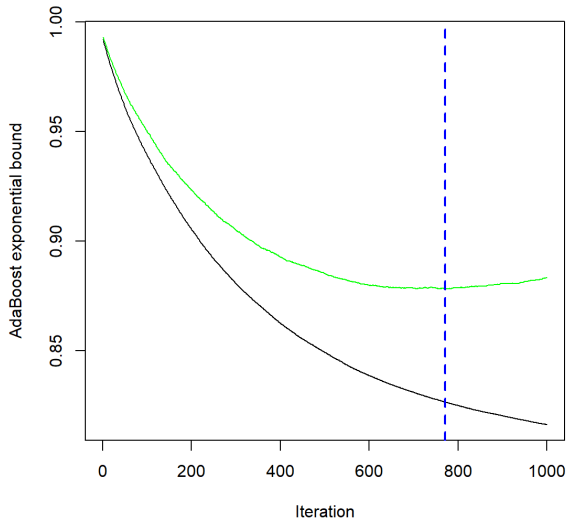

# Tune the weak learner

Hide

```
# our previous example with a circle
set.seed(1)
n = 500
x1 = runif(n, -1, 1)
x2 = runif(n, -1, 1)
y = rbinom(n, size = 1, prob = ifelse(x1^2 + x2^2 < 0.6, 0.8, 0.2))
xgrid = expand.grid(x1 = seq(-1, 1, 0.01), x2 = seq(-1, 1, 0.01))
```

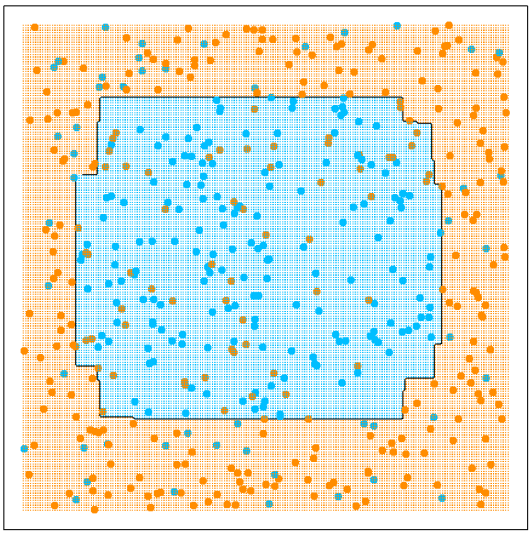
Hide

```
# if we only use one variable in each tree
gbm.fit = gbm(y~., data = data.frame(x1, x2, y), distribution="adaboost",
              n.trees=1000, shrinkage=0.01, bag.fraction=0.8,
              interaction.depth = 1, cv.folds=10)

usetree = gbm.perf(gbm.fit, method="cv")
```




Hide

```
  Fx = predict(gbm.fit, xgrid, n.trees=usetree)

  pred = matrix(1/(1+exp(-2*Fx)) > 0.5, 201, 201)
  par(mar=rep(2,4))
  contour(seq(-1, 1, 0.01), seq(-1, 1, 0.01), pred, levels=0.5, labels="",axes=FALS
E)
  points(x1, x2, col = ifelse(y == 1, "deepskyblue", "darkorange"),
         pch = 19, yaxt="n", xaxt = "n")
  points(xgrid, pch=".", cex=1.2, col=ifelse(pred, "deepskyblue", "darkorange"))
  box()
```



Hide

```
  pretty.gbm.tree(gbm.fit, i.tree = 1)
```

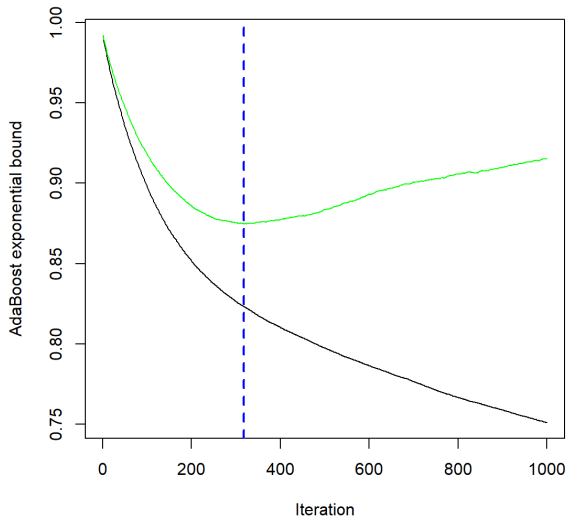
| | SplitVar <int> | SplitCodePred <dbl> | LeftNode <int> | RightNode <int> | MissingNode <int> | ErrorReduction <dbl> | Wei... <dbl> |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0.7279700371 | 1 | 2 | 3 | 24.10443 | 400 |
| 1 | -1 | 0.0006567649 | -1 | -1 | -1 | 0.00000 | 347 |
| 2 | -1 | -0.0072046589 | -1 | -1 | -1 | 0.00000 | 53 |
| 3 | -1 | -0.0003848737 | -1 | -1 | -1 | 0.00000 | 400 |

4 rows | 1-8 of 9 columns

We could also allow interactions in each tree. This will lead to deeper tree and it will take less number of iterations to achieve similar performances.

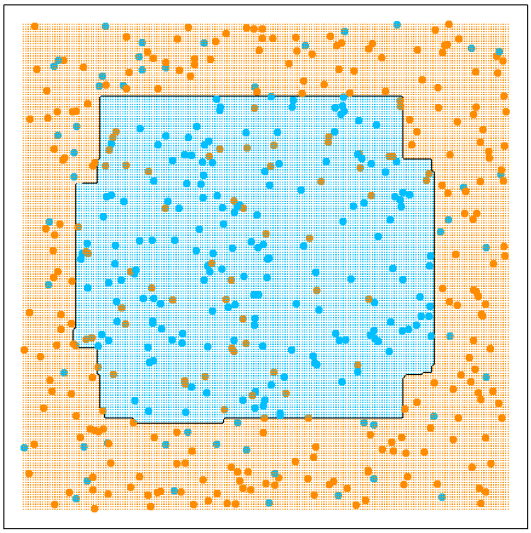
Hide

```
  gbm.fit = gbm(y~., data = data.frame(x1, x2, y), distribution="adaboost",
                n.trees=1000, shrinkage=0.01, bag.fraction=0.8,
                interaction.depth = 2, cv.folds=10)

  usetree = gbm.perf(gbm.fit, method="cv")
```




Hide

```
  Fx = predict(gbm.fit, xgrid, n.trees=usetree)

  pred = matrix(1/(1+exp(-2*Fx)) > 0.5, 201, 201)
  par(mar=rep(2,4))
  contour(seq(-1, 1, 0.01), seq(-1, 1, 0.01), pred, levels=0.5, labels="",axes=FALS
E)
  points(x1, x2, col = ifelse(y == 1, "deepskyblue", "darkorange"),
         pch = 19, yaxt="n", xaxt = "n")
  points(xgrid, pch=".", cex=1.2, col=ifelse(pred, "deepskyblue", "darkorange"))
  box()
```



Hide

```
pretty.gbm.tree(gbm.fit, i.tree = 1)
```

| | SplitVar <int> | SplitCodePred <dbl> | LeftNode <int> | RightNode <int> | MissingNode <int> | ErrorReduction <dbl> | Wei... <dbl> |
|---|---|---|---|---|---|---|---|
| 0 | 1 | -0.658632189 | 1 | 2 | 6 | 30.69190 | 400 |
| 1 | -1 | -0.005887393 | -1 | -1 | -1 | 0.00000 | 71 |
| 2 | 0 | 0.694127902 | 3 | 4 | 5 | 31.73123 | 329 |
| 3 | -1 | 0.003043478 | -1 | -1 | -1 | 0.00000 | 277 |
| 4 | -1 | -0.005792988 | -1 | -1 | -1 | 0.00000 | 52 |
| 5 | -1 | 0.001646833 | -1 | -1 | -1 | 0.00000 | 329 |
| 6 | -1 | 0.000309508 | -1 | -1 | -1 | 0.00000 | 400 |

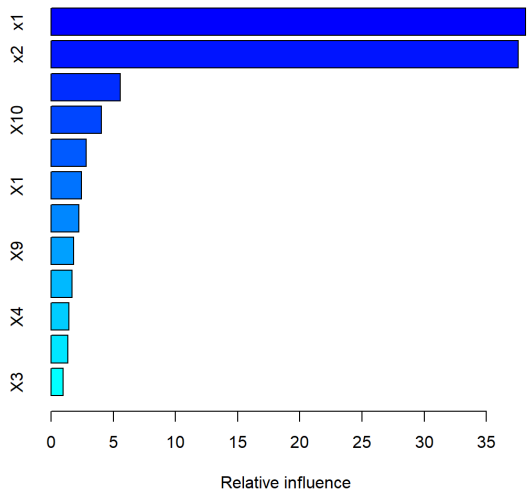
7 rows | 1-8 of 9 columns

# Variable importance

You can estimate the Variable importance with similar fashion as a random forest. Since we can calculate the reduced accuracy by randomly permuting the values of out-of-bag data, we may then aggregate these measures over all trees.

Hide

```
gbm.fit = gbm(y~., data = data.frame(x1, x2, matrix(runif(n*10), n, 10), y),
              distribution="adaboost", n.trees=1000, shrinkage=0.01,
              bag.fraction=0.8, interaction.depth = 2, cv.folds=10)

summary(gbm.fit, method = permutation.test.gbm)
```




| var<br><chr> | rel.inf<br><dbl> |
|---|---|
| x1 | 38.1405889 |
| x2 | 37.5402623 |
| X2 | 5.5556527 |
| X10 | 4.0639256 |
| X7 | 2.8284456 |
| X1 | 2.4353977 |
| X8 | 2.2193764 |
| X9 | 1.8153845 |
| X6 | 1.6683819 |
| X4 | 1.4292909 |