Boosting

AdaBoost, Gradient Boosting, XGboost

SYS 6018 | Spring 2024

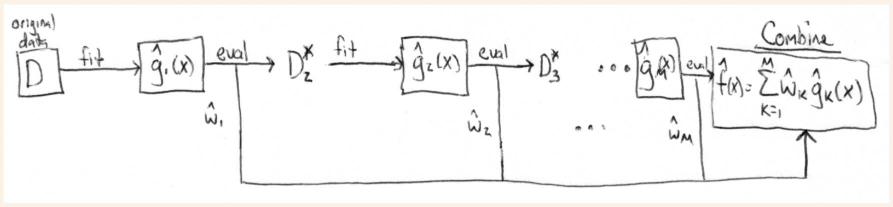
boosting.pdf

Contents

1 Boosting

Boosting is a *sequential* ensemble method.

Boosting Sketch



- A boosting model can be written as a generic ensemble
  - M is the number of base learners
  - $\hat{a}_k$ is the weight for the kth base learner ($\hat{a}_k \geq 0$).
  - $\hat{g}_k(x)$ is the prediction from the kth base learner

$$\hat{f}_M(x) = \sum_{k=1}^{M} \hat{a}_k \, \hat{g}_k(x)$$

- The key distinction of boosting models is that the base learners are **fitted sequentially**, and the best model at stage m is dependent on all models fit prior to stage m.

$$\hat{f}_{m+1}(x) = \underset{a,\, g(x)}{\arg\min} \sum_{i=1}^{n} L(y_i, \hat{f}_m(x_i) + a\, g(x_i))$$

- Boosting is primarily a *bias* reducer
  - The base models are often simple/weak (low variance, but high bias) models (like shallow trees)
- The complexity of the final model is based on i) the complexity of the base learners and ii) the number of iterations
  - Boosting models will overfit as the number of iterations increases
    - Early stopping is necessary
    - Less of a problem for hard classification problems with balanced data
  - Can apply *shrinkage* (making a_k smaller), to reduce complexity
- There are two main versions of boosting:
  - *Gradient Boosting*: fits the next model in the sequence $\hat{g}_k(x)$ to the (pseudo) residuals calculated from the predictions on the previous models
  - *AdaBoost*: fits the next model to sequentially *weighted* observations. The weights are proportional to the how poorly the current models predict the observation.

2 AdaBoost

AdaBoost was motivated by the idea that many *weak* leaners can be combined to produce a *strong* aggregate model.

- AdaBoost is for binary classification problems
- Trees are a popular base learner
  - *Weak* learners are usually used. For trees, this means shallow depth.
- At each iteration, the current model is evaluated.
  - The *ensemble weight* of model k is based on its performance (on all the training data)
  - The *observation weight* of observation i is increased if it is mis-classified and decreased if it is correctly classified.
  - Thus, at each iteration, those observations that are mis-classified are weighted higher and get extra attention in the next iteration.
- Because Adaboost uses hard-classifiers, it is sensitive to unbalanced data and unequal misclassification costs.
  - Because the thresholds are set at $p > .50$
  - There are, of course, ways to account for unbalance and unequal costs in the algorithm
  - An improvement to AdaBoost, *LogitBoost* explicitly attempts to estimate the class probability during each iteration which will allow easier post-fitting adjustments for unequal costs

Weighted Loss Functions (with observations weights)

Let $w_i \geq 0$ be a *weight* associated with observation i. The weighted loss for predictions $\hat{\mathbf{y}} = \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$ is

$$L(\mathbf{y}, \hat{\mathbf{y}}, \mathbf{w}) = \sum_{i=1}^{n} w_i L(y_i, \hat{y}_i)$$

2.1 Adaboost Algorithm

Algorithm: AdaBoost (Discrete)

Inputs:

- $D = \{(x_i, y_i\}_{i=1}^n$, where $y_i \in \{-1, 1\}$
- Tuning parameters for base model $\hat{g}$
- Maximum number of iterations, M or other stopping criteria

Algorithm:

1. Initialize *observation weights* $w_i = 1/n$ for all i
2. For $k = 1$ to M:
   a. Fit a *classifier* $\hat{g}_k(x)$ that maps (x_i, w_i) to $\{-1, 1\}$. In other words, the classifier must make a hard classification using weighted observations.

   b. Compute the weighted mis-classification rate

   $$e_k = \frac{\sum_{i=1}^n w_i \, \mathbb{1}(y_i \neq \hat{g}_k(x_i))}{\sum_{i=1}^n w_i}$$

   Note: $0 \leq e_k \leq .5$ since model fit and evaluated on same training data.

   c. Calculate the *coefficient* for model k (*ensemble weight*)

   $$\hat{a}_k = \log\left(\frac{1 - e_k}{e_k}\right)$$

   Note: $0 \leq a_k < \infty$.

   d. Update the *observations weights*. Increase weights for observations that are mis-classified by model $\hat{g}_k$ and decrease weights for the correctly classified observations.

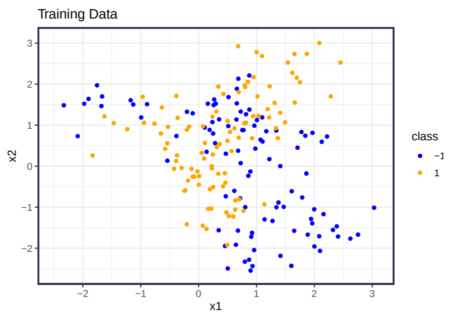
   $$\begin{aligned}\tilde{w}_i &= w_i \cdot \exp\left(\hat{a}_k \cdot \mathbb{1}(y_i \neq \hat{g}_k(x_i))\right) \\ &= \begin{cases} w_i \frac{1-e_k}{e_k} & \text{if obs } i \text{ is misclassified} \\ w_i & \text{if obs } i \text{ is correctly classified} \end{cases} \\ w_i &= \frac{\tilde{w}_i}{\sum_{j=1}^n \tilde{w}_j} \qquad (\textit{re-normalize weights})\end{aligned}$$

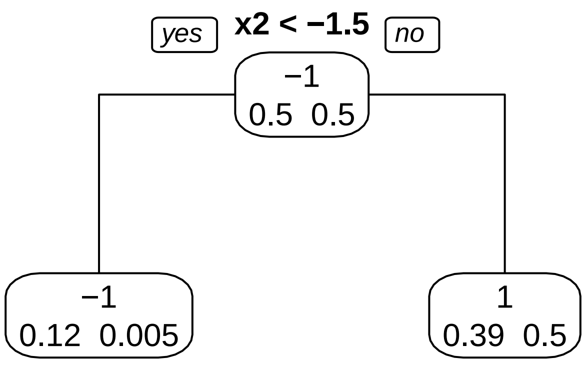
3. Output final ensemble $\hat{f}_M(x)$

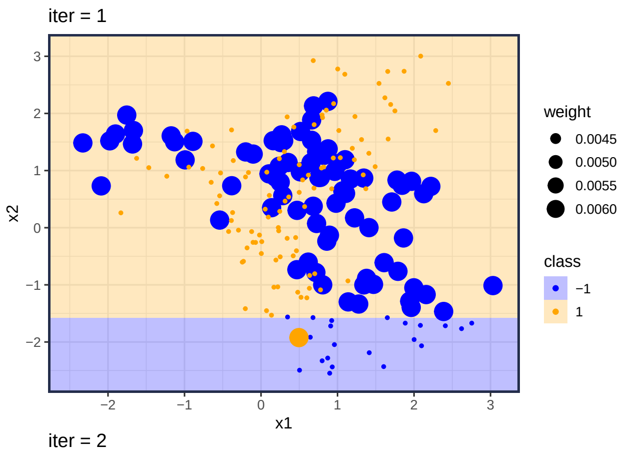
$$\hat{f}_M(x) = \sum_{k=1}^{M} \hat{a}_k \, \hat{g}_k(x)$$

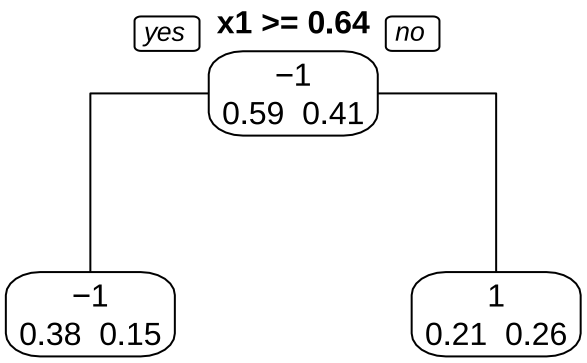
- Where $\hat{f}_k(x) = \hat{a}_k \, \hat{g}_k(x)$
- Hard classification: $\hat{f}_M(x) > 0$
- Or remap to a probability $\hat{p}(x) = \frac{e^{2\hat{f}(x)}}{1+e^{2f\hat{(x)}}}$ for thresholding

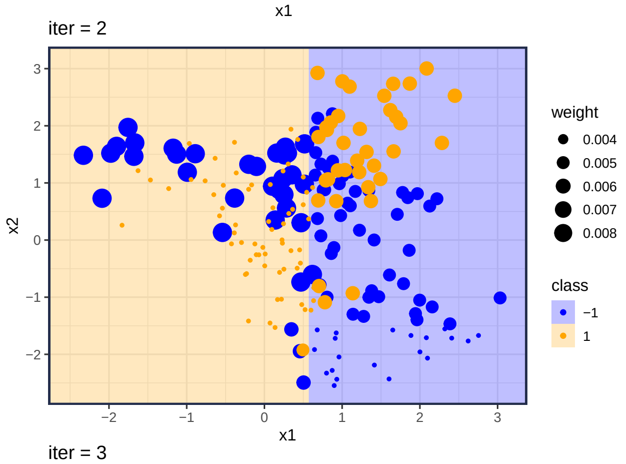
2.1.1 Illustration with Stumps (depth = 1, n.nodes=2)

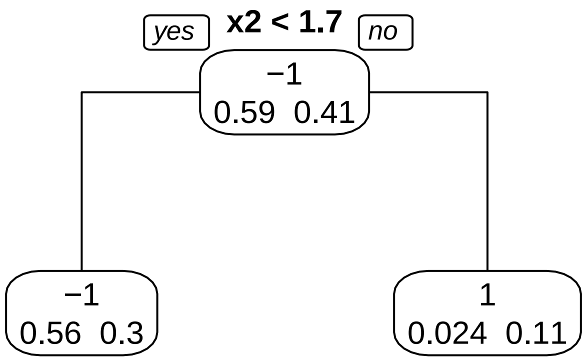


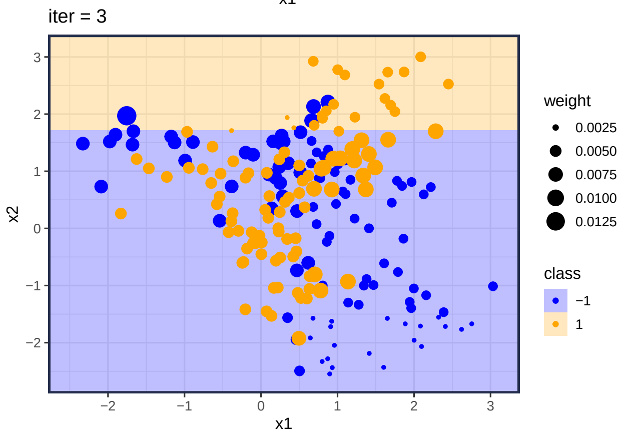


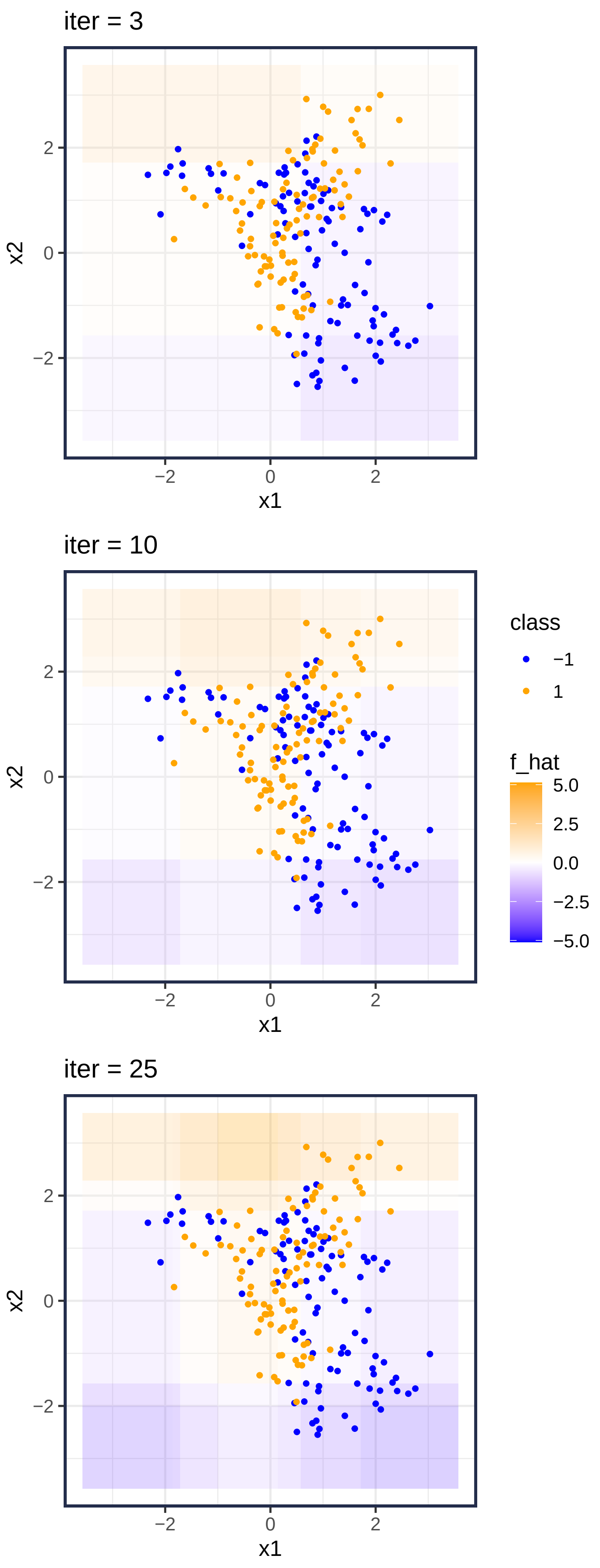




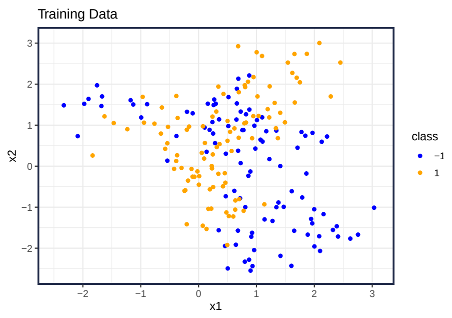


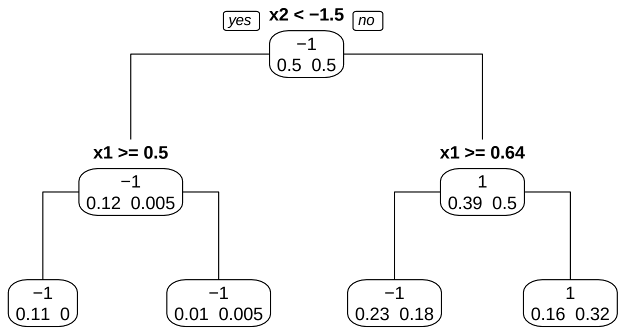


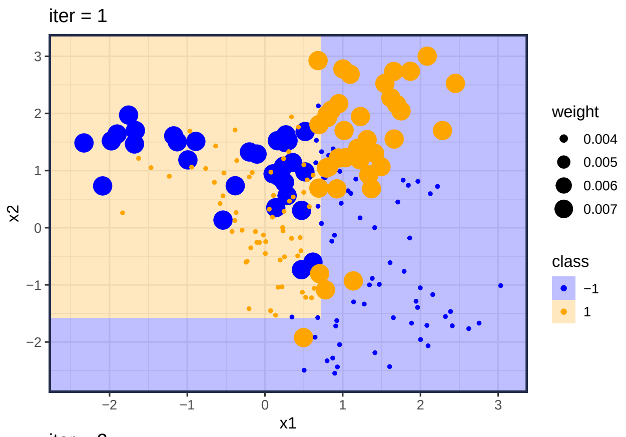


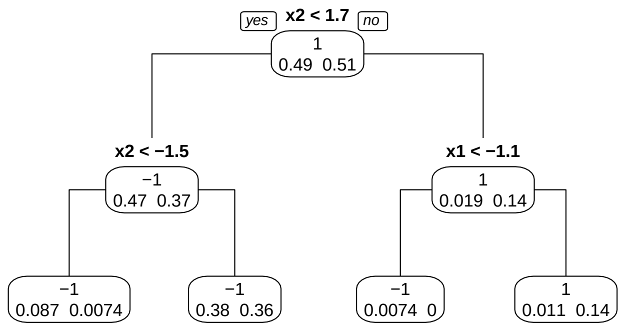

iter = 3
iter = 10
iter = 25
x1
x2
−2
0
2
class
−1
1
f_hat
5.0
2.5
0.0
−2.5
−5.0

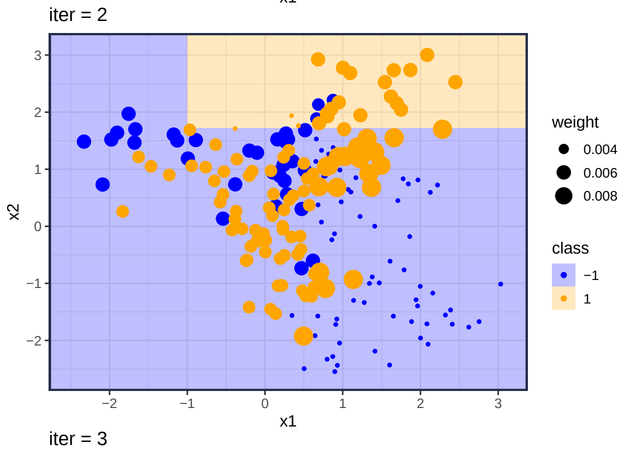
2.1.2 Illustration with depth = 2, n.nodes=4

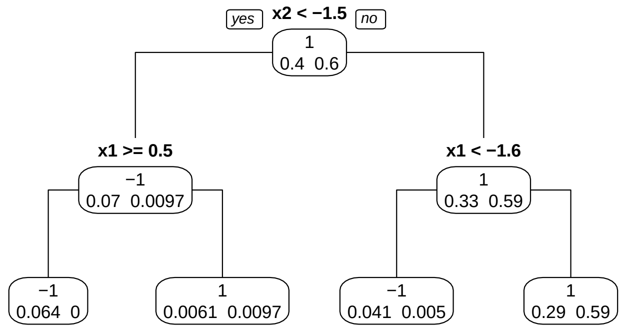


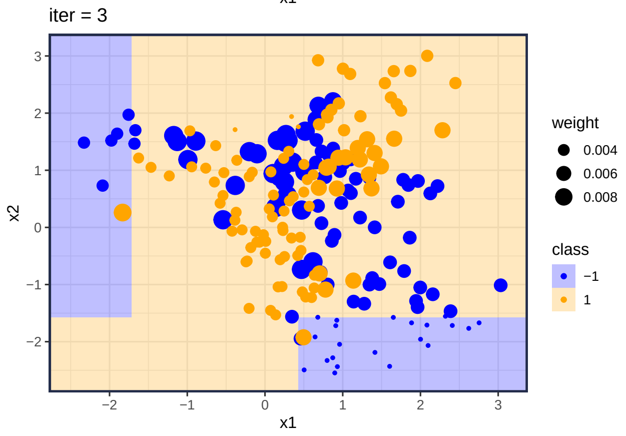


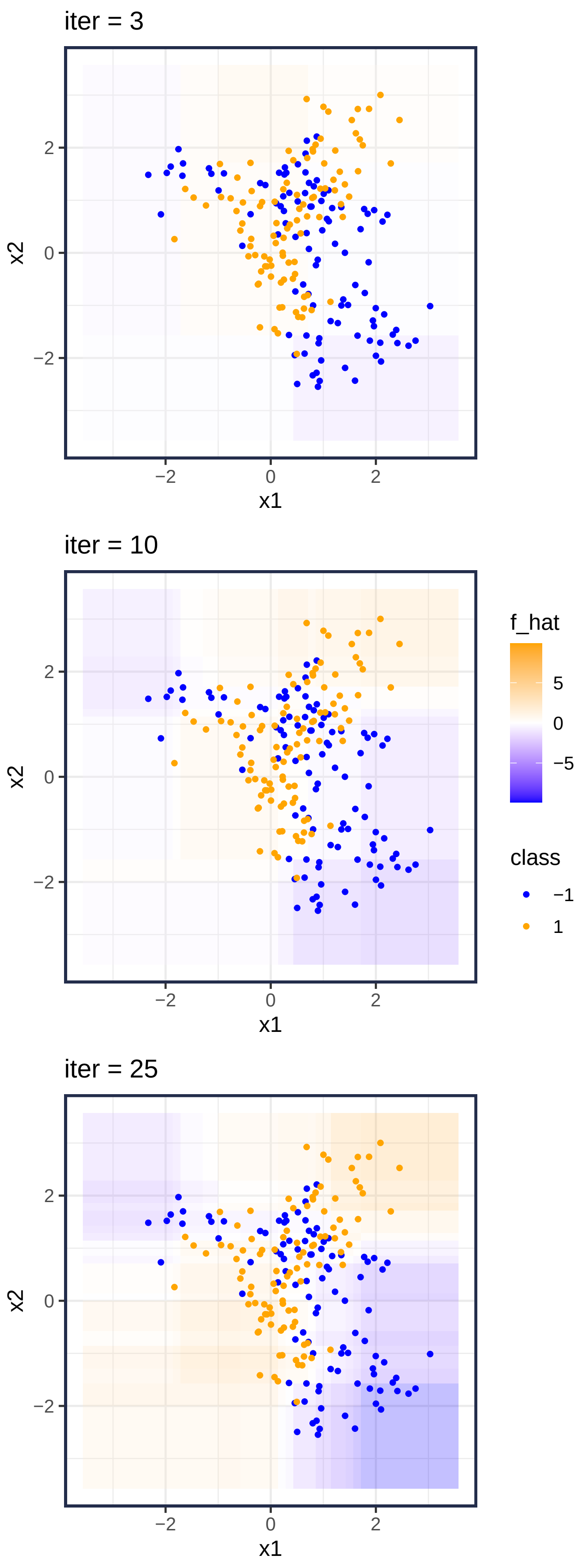


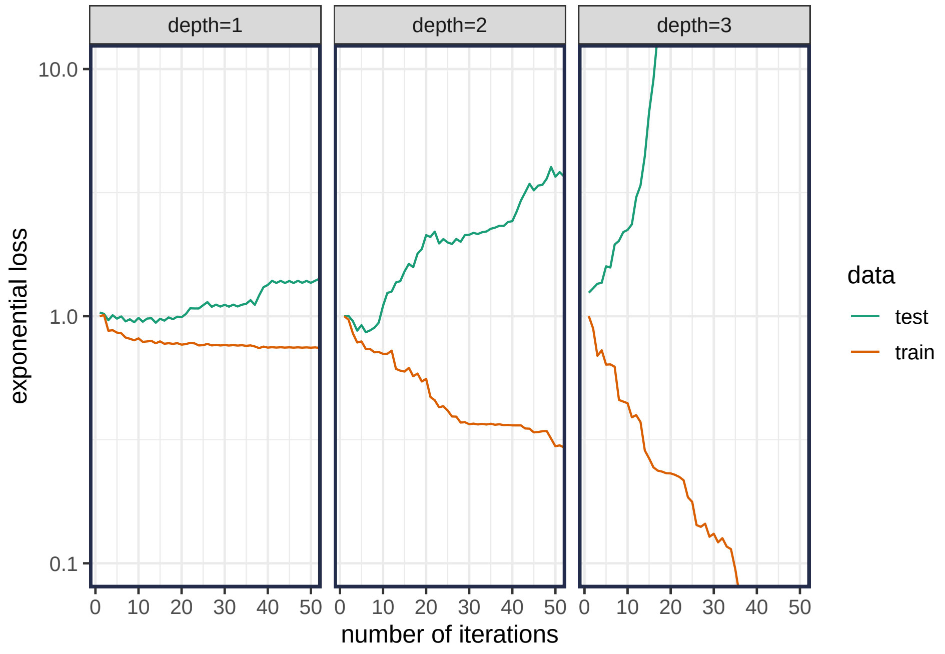


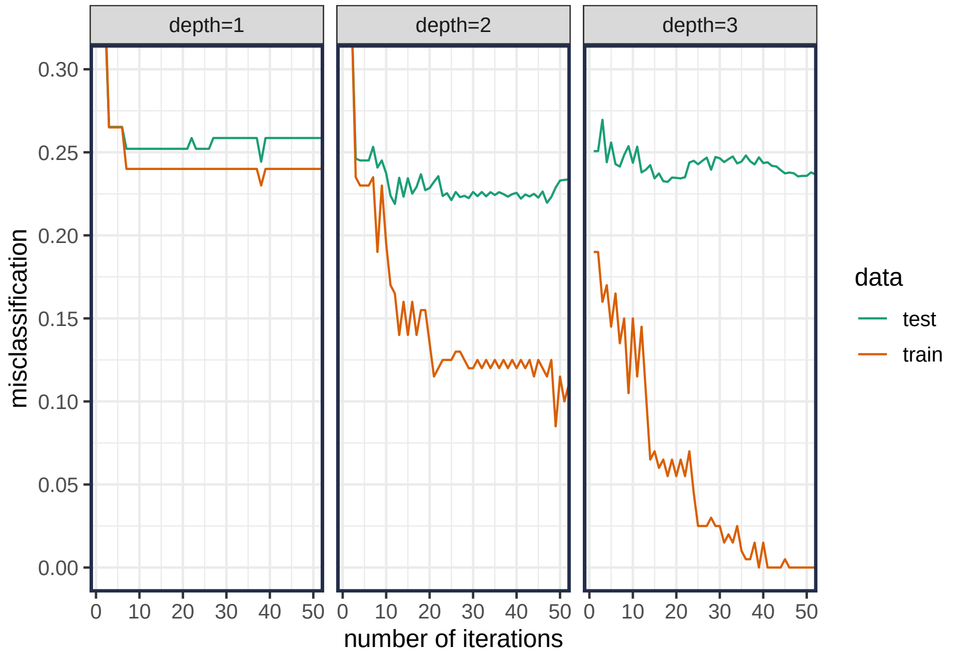







iter = 3
iter = 10
iter = 25
x1
x2
f_hat
5
0
−5
class
−1
1


depth=1
depth=2
depth=3
exponential loss
10.0
1.0
0.1
0
10
20
30
40
50
number of iterations
data
test
train


depth=1
depth=2
depth=3
misclassification
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
10
20
30
40
50
number of iterations
data
test
train

2.2 AdaBoost Details

- Adaboost uses an outcome variable of $y \in \{-1, 1\}$
- AdaBoost implicitly uses the loss function:

$$\begin{aligned} L(y, f) &= e^{-yf} \\ &= \begin{cases} e^{-f} & y = +1 \\ e^{f} & y = -1 \end{cases} \end{aligned}$$

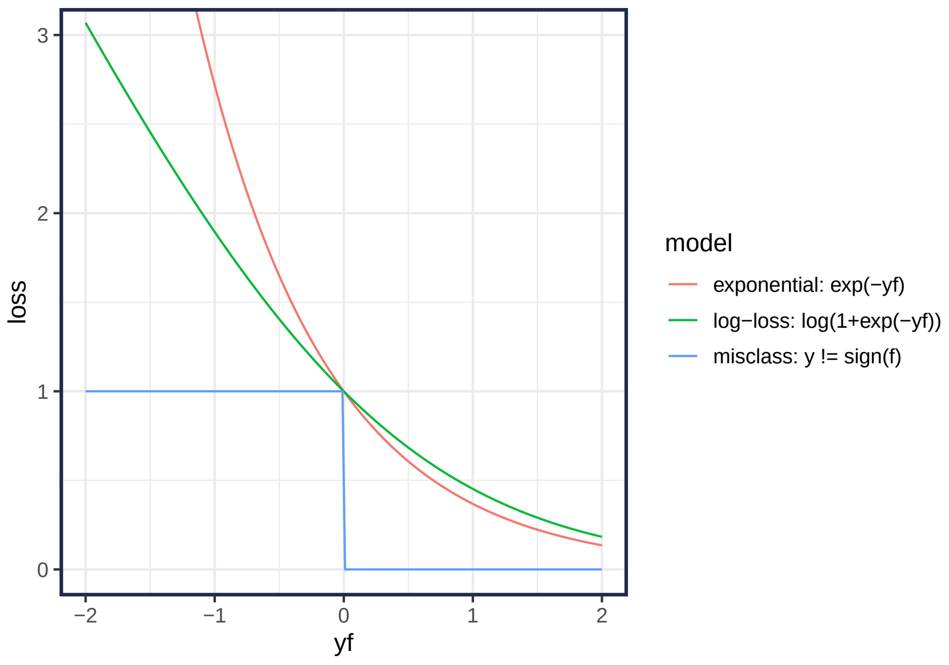
- Adaboost estimates the probability that $Y = +1$ as

$$\begin{aligned} \hat{p}(x) &= \frac{e^{\hat{f}_M(x)}}{e^{-\hat{f}_M(x)} + e^{\hat{f}_M(x)}} \\ &= \frac{e^{2\hat{f}_M(x)}}{1 + e^{2\hat{f}_M(x)}} \end{aligned}$$

where $p(x) = \Pr(Y = +1 \mid X = x)$

- And $\hat{f}(x)$ is an estimate of

$$\begin{aligned} \hat{f}_M(x) &= \frac{1}{2} \log \frac{\hat{p}(x)}{1 - \hat{p}(x)} \\ &= \frac{1}{2} \text{logit}\, \hat{p}(x) \end{aligned}$$

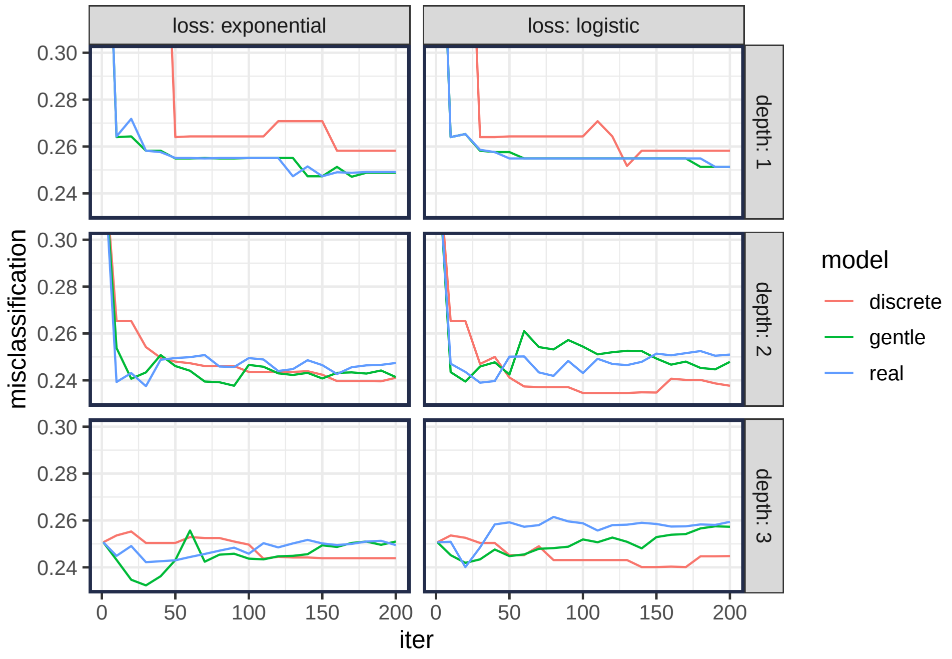


- Comparison with logistic regression (using log-loss / negative binomial log-likelihood)
  - $\hat{f}(x) = \text{logit}\, \hat{p}(x)$
  - $\hat{p}(x) = \frac{e^{\hat{f}_M(x)}}{1+e^{\hat{f}_M(x)}}$
  - Log-loss: $\log(1 + e^{-yf})$ (using $y \in \{-1, +1\}$)

2.3 R package `ada`

The R package `ada` provides an implementation of AdaBoost (and related methods).

- See Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive Logistic Regression: A statistical view of boosting. Annals of Statistics, 28(2), 337-374. for the details of model variations
  - {Discrete, Real, Gentle} AdaBoost
  - Logitboost



Algorithm: Real AdaBoost

Inputs:

- $D = \{(x_i, y_i\}_{i=1}^n$, where $y_i \in \{-1, 1\}$
- Tuning parameters for base model $\hat{g}$
- Maximum number of iterations, M

Algorithm:

1. Initialize *observation weights* $w_i = 1/n$ for all i
2. For $k = 1$ to M:
   a. Fit a model $\hat{g}_k(x)$ that uses weighted inputs (x_i, w_i) to estimate a probability $\hat{p}_k(x) = \widehat{\Pr}(Y = 1 \mid X = x)$. In other words, the classifier must make a soft classification using weighted observations.
   b. Set $f_m(x) = \frac{1}{2}\text{logit}\ \hat{p}_k(x)$
   c. Update the *observations weights*. Increase weights for observations that are mis-classified by model $\hat{g}_k$ and decrease weights for the correctly classified observations.

$$\tilde{w}_i = w_i \cdot \exp\left(-y_i \hat{f}_m(x_i)\right)$$
$$w_i = \frac{\tilde{w}_i}{\sum_{j=1}^n \tilde{w}_j} \qquad \textit{(re-normalize weights)}$$

3. Output final ensemble $\hat{f}_M(x)$

$$\hat{f}_M(x) = \sum_{k=1}^M \hat{f}_k(x)$$

- Hard classification: $\hat{f}_M(x) > 0$
- Or remap to a probability $\hat{p}(x) = \frac{e^{2f}}{1+e^{2f}}$ for thresholding

Algorithm: Gentle AdaBoost

Inputs:
- $D = \{(x_i, y_i\}_{i=1}^n$, where $y_i \in \{-1, 1\}$
- Tuning parameters for base model $\hat{g}$
- Maximum number of iterations, M

Algorithm:
1. Initialize *observation weights* $w_i = 1/n$ for all i and $f_0(x) = 0$
2. For $k = 1$ to M:
   a. Fit a model $\hat{g}_k(x)$ with weighted least squares that estimates y_i using features x_i and weights w_i.
   b. Update the *observations weights*. Increase weights for observations that are mis-classified by model $\hat{g}_k$ and decrease weights for the correctly classified observations.

$$\tilde{w}_i = w_i \cdot \exp\left(-y_i \hat{g}_m(x_i)\right)$$
$$w_i = \frac{\tilde{w}_i}{\sum_{j=1}^n \tilde{w}_j} \qquad \textit{(re-normalize weights)}$$

3. Output final ensemble $\hat{f}_M(x)$

$$\hat{f}_M(x) = \sum_{k=1}^M \hat{g}_k(x)$$

- Hard classification: $\hat{f}_M(x) > 0$

Algorithm: LogitBoost

Inputs:
- $D = \{(x_i, y_i\}_{i=1}^n$, where $y_i \in \{-1, 1\}$
- Tuning parameters for base model $\hat{g}$
- Maximum number of iterations, M
- Let $y_i^* = (y+1)/2 \in \{0, 1\}$

Algorithm:

1. Initialize *observation weights* $w_i = 1/n$ for all i and $f_0(x) = 0$
2. For $k = 1$ to M:
   a. Like in newton-raphson for logistic regression, calculate the working response and weights for all observations

$$z_i = \frac{y_i^* - p_i}{p_i(1 - p_i)}$$
$$w_i = p_i(1 - p_i)$$

   b. Fit a model $\hat{g}_k(x)$ with weighted least squares that estimates z_i using features x_i and weights w_i.
   c. Update $\hat{f}_k(x) = \hat{f}_{k-1}(x) + \hat{g}_k(x)/2$ and $p_i = e^{\hat{f}_k(x)}/(e^{\hat{f}_k(x)} + e^{-\hat{f}_k(x)})$
3. Output final ensemble $\hat{f}_M(x) \in \mathbb{R}$

$$\hat{f}_M(x) = \sum_{k=1}^{M} \frac{1}{2}\,\hat{g}_k(x)$$

- Where $\hat{f}_k(x) = \frac{1}{2}\,\hat{g}_k(x)$.
- Hard classification: $\hat{f}_M(x) > 0$
- Or remap to a probability $\hat{p}(x) = \frac{e^{2f}}{1+e^{2f}}$ for thresholding

3 Gradient Boosting

The boosting model:

$$\hat{f}_M(x) = \sum_{k=1}^{M} \hat{a}_k \, \hat{g}_k(x)$$

Sequential Fitting:

$$\hat{f}_{k+1}(x) = \mathop{\arg\min}_{a,\, g(x)} \sum_{i=1}^{n} L(y_i, \hat{f}_k(x_i) + a\, g(x_i))$$

The concept of gradient boosting is sequentially re-fit to the negative (functional) gradients of the loss function (or *pseudo* residuals).

- The same structure can be used for many different loss functions
  - it works the same for regression and classification
  - survival analysis, ranking, etc.

3.1 Gradient Descent

- Our objective is to find the model (or model parameters) that minimize the loss function
- From any starting point, we can move toward the optimum using *gradient descent*:

$$f_{k+1} = f_k - \nu_k \, L'(f_k)$$

  - $\nu_k > 0$ is the step-size
  - $L'(f_k)$ is the functional derivative of the loss with respect to the model f_k

- Boosting fits models sequentially:

$$\hat{f}_{k+1}(x) = \hat{f}_k(x) + \hat{a}_k \, \hat{g}_k(x)$$

- So we see a parallel; each boosting model $\hat{g}_k(x)$ can be viewed as estimating the *negative derivative* of the loss function.

3.2 L_2 Boosting

L_2 boosting is based on the the squared error loss function

$$L(y_i, \hat{f}(x_i)) = \frac{1}{2}(y_i - \hat{f}(x_i))^2$$

- The *negative gradients* are

$$\begin{aligned} r_i &= \left[-\frac{\partial L(y_i, f_i)}{\partial f_i} \right]_{f_i = \hat{f}(x_i)} \\ &= y_i - \hat{f}(x_i) \end{aligned}$$

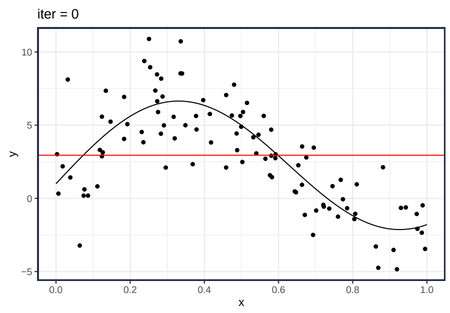
- L2 Boosting is simply re-fitting to the residuals.

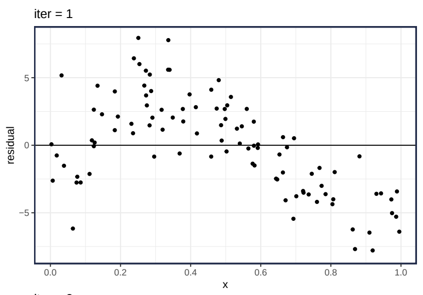
Algorithm: L_2 Boosting

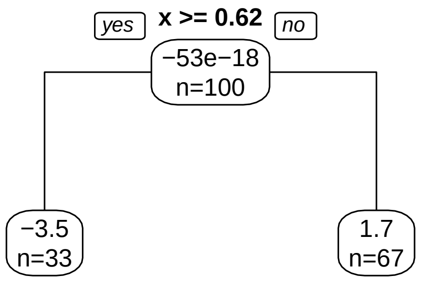
```
1. Initialize $\hat{f}_0(x) = \bar{y}$
2. For $k = 1$ to $M$:
    a. Calculate residuals $r_i = y_i - \hat{f}_{k-1}(x_i)$ for all $i$
    b. Fit a base learner (e.g., regression trees) to the residuals $\{(x_i, r_i)\}_{i=1}^n$ to get the model $\hat{g}_k(x)$
    c. Update the overall model $\hat{f}_k(x) = f_{k-1}(x) + \nu \hat{g}_k(x)$
        • $0 \le \nu \le 1$ is the step-size (*shrinkage*)
3. Final model is $\hat{f}_M(x) = \bar{y} + \sum_{k=1}^M \nu \hat{g}_k(x)$
```

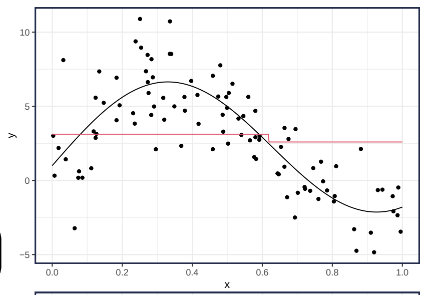
- Like AdaBoost, emphasis is given to observations that are predicted poorly (large residuals)

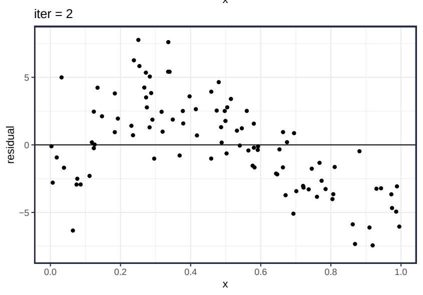
3.2.1 Illustration using stumps (depth=1, n.nodes=2, $\nu = .1$)

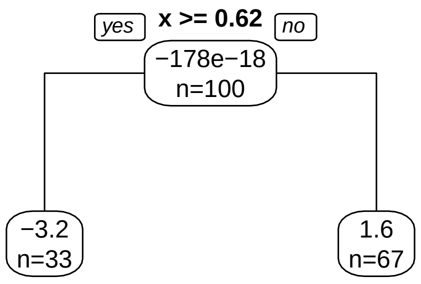


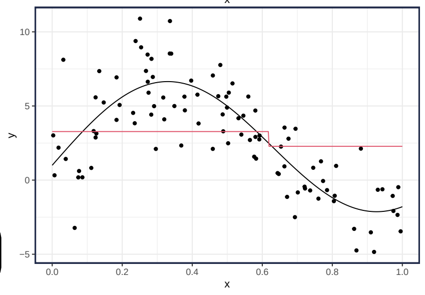


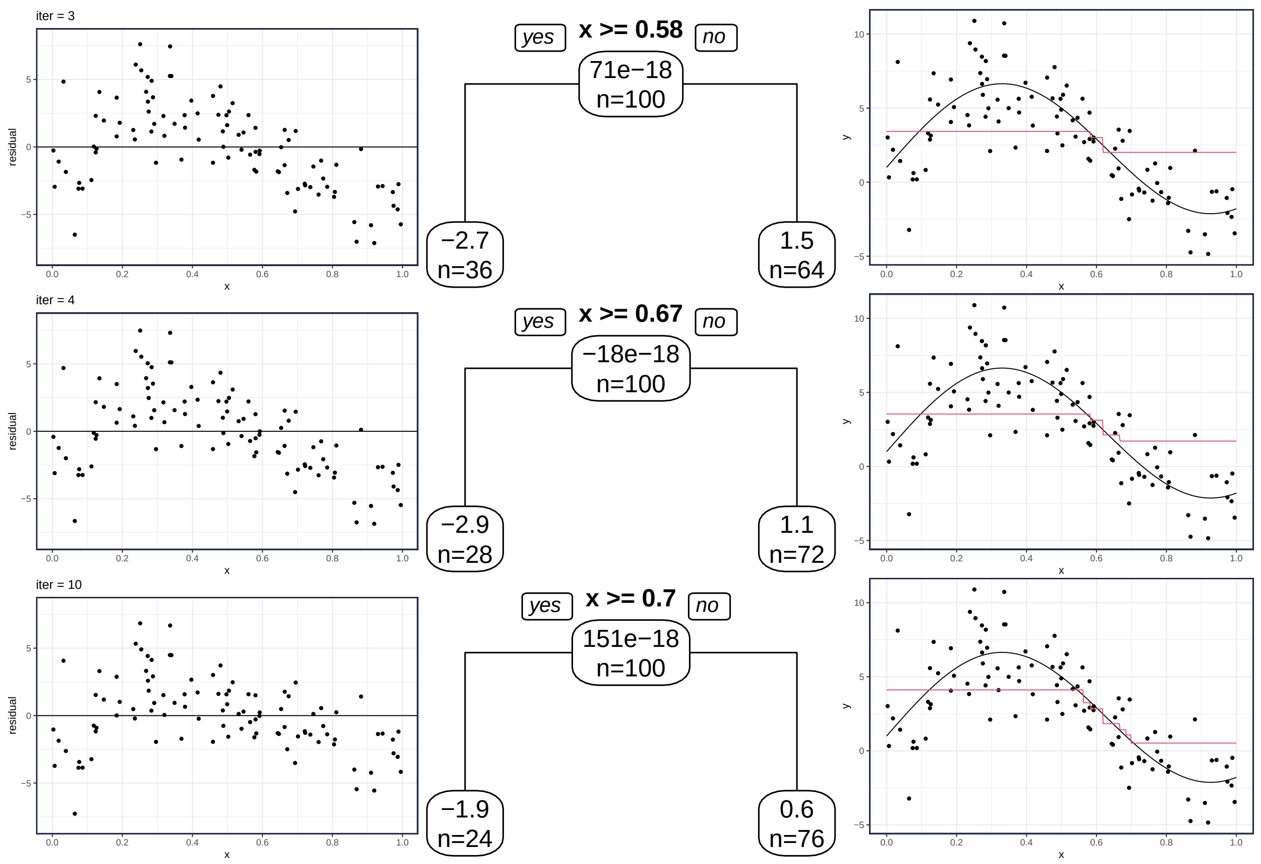


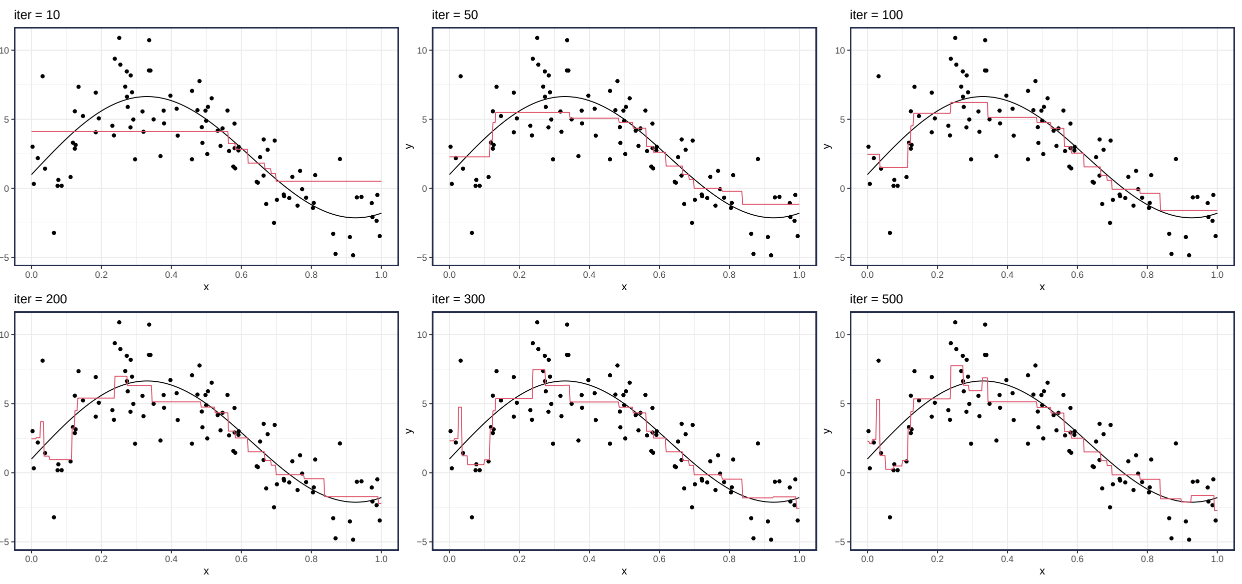









iter = 3
yes
x >= 0.58
no
71e−18
n=100
−2.7
n=36
1.5
n=64
iter = 4
yes
x >= 0.67
no
−18e−18
n=100
−2.9
n=28
1.1
n=72
iter = 10
yes
x >= 0.7
no
151e−18
n=100
−1.9
n=24
0.6
n=76
residual
y
x


iter = 10
iter = 50
iter = 100
iter = 200
iter = 300
iter = 500
y
x

3.3 GBM (Gradient Boosting Machine)

- R package `gbm`
- GBM Documentation
- GBM is a first order approach. It does not consider Hessian.

3.3.1 Model/Tree Tuning Parameters

- Tree depth (`interaction.depth`)
  - Grows trees to a depth specified by `interaction.depth` (unless there are not enough observations in the terminal nodes)
- Minimum number of observations allowed in the terminal nodes (`n.minobsinnode`)
- Sub-sampling (`bag.fraction`)
  - *Stochastic Gradient Boosting*
  - Sample (without replacement) at each iteration
- Loss Function (`distribution`)
  - The loss function is determined by the `distribution` argument
  - Use `distribution="gaussian"` for squared error
  - Other options are: `bernoulli` (for logistic regression), `poisson` (for Poisson regression), `pairwise` (for ranking/LambdaMart), `adaboost` (for the adaboost exponential loss), etc.

3.3.2 Boosting Tuning Parameters

- Number of iterations/trees (`n.trees`)
  - Use cross-validation (or out-of-bag) to find optimal value
  - Can use the helper function `gbm.perf()` to get the optimal value
- Shrinkage parameter (`shrinkage`)
  - Set small, but the smaller the `shrinkage`, the more iterations/trees need to be used
  - "Ranges from 0.001 to 0.100 usually work"
- Cross-validation (`cv.folds`)
  - `gbm` has a built in cross-validation
  - no way to manually set the folds

3.3.3 Computational Settings

- Number of Cores (`n.cores`)
  - Only used when cross-validation is implemented

3.4 xgboost (Extreme Gradient Boosting)

- R package `xgboost`
- xgboost Documentation
- xgboost Model
- xbgoost Paper

3.4.1 Model/Tree Tuning Parameters

- Different base leaners (`booster`)
  - `gbtree` is a tree
  - `gblinear` creates a (generalized) liner model (forward stagewise linear model)
- Tree building (`tree_method`)
  - To speed up the fitting, only consider making splits at certain quantiles of the input vector (rather than considering every unique value)
- Sub-sampling (`subsample`)
  - *Stochastic Gradient Boosting*
  - Sample (without replacement) at each iteration
- Feature sampling (`colsample_bytree`, `colsample_bylevel`, `colsample_bynode`)
  - Like used in Random Forest, the features/columns are subsampled
  - Can use a subsample of features for each tree, level, or node

Model Complexity Parameters

- Tree depth (`max_depth`)
  - Grows trees to a depth specified by `max_depth` (unless there are not enough observations in the terminal nodes)
  - Trees may not reach `max_depth` if the `gamma` or `min_child_weight` arguments are set.
- Minimum number of observations (or sum of weights) allowed in the terminal nodes (`min_child_weight`)
- Pruning (`gamma` or `min_split_loss`)
  - Minimum loss reduction required to make a further partition on a leaf node of the tree
  - The larger gamma is, the more conservative the algorithm will be
- ElasticNet type penalty (`lambda` and `alpha`)
  - `lambda` is an L_2 penalty
  - `alpha` is an L_1 penalty

Note

- Recall that trees model the response as a *constant* in each region

$$\hat{f}_T(x) = \sum_{m=1}^{M} \hat{c}_m \, \mathbb{1}(x \in \hat{R}_m)$$

- Cost-complexity pruning found the optimal tree as the one that minimized the penalized loss objective function:

$$C_\gamma(T) = \sum_{m=1}^{|T|} \text{Loss}(T) + \gamma|T|$$

- XGBoost selects a tree at each iteration using the following penalized loss:

$$C_{\gamma,\lambda,\alpha}(T) = \sum_{m=1}^{|T|} \text{Loss}(T) + \gamma|T| + \frac{\lambda}{2}\sum_{m=1}^{|T|} \hat{c}_m^2 + \alpha \sum_{m=1}^{|T|} |\hat{c}_m|$$

- Loss Function (`objective`)
  - The loss function is determined by the `objective` argument
  - Use `reg:squarederror` for squared error
  - Other options are: `reg:logistic` or `binary:logistic` (for logistic regression), `count:poisson` (for Poisson regression), `rank:pairwise` (for ranking/LambdaMart), etc.

3.4.2 Boosting Tuning Parameters

- Shrinkage parameter (`eta` or `learning_rate`)
  - Set small, but the smaller the `eta`, the more iterations/trees need to be used
- Number of iterations/trees (`num_rounds`)
  - Use cross-validation (or out-of-bag) to find optimal value
- Cross-validation (`xgb.cv`)
  - `xgboost` has a built in cross-validation
  - It is possible to manually set the folds

3.4.3 Computational Settings

- Number of Threads (`nthread`)
- GPU Support (https://xgboost.readthedocs.io/en/latest/gpu/index.html)
  - Used for finding tree split points and evaluating/calculating the loss function

3.5 CatBoost

- R package: (https://github.com/catboost/catboost/tree/master/catboost/R-package)
- CatBoost Documentation
- Model/Tree Tuning Parameters:

- Boosting Tuning Parameters:

3.6 LightGBM

- R Package: https://github.com/microsoft/LightGBM/tree/master/R-package
- LightGBM Documentation
- Model/Tree Tuning Parameters:

- Boosting Tuning Parameters:

4 Appendix: L2 Tree Boosting R Code

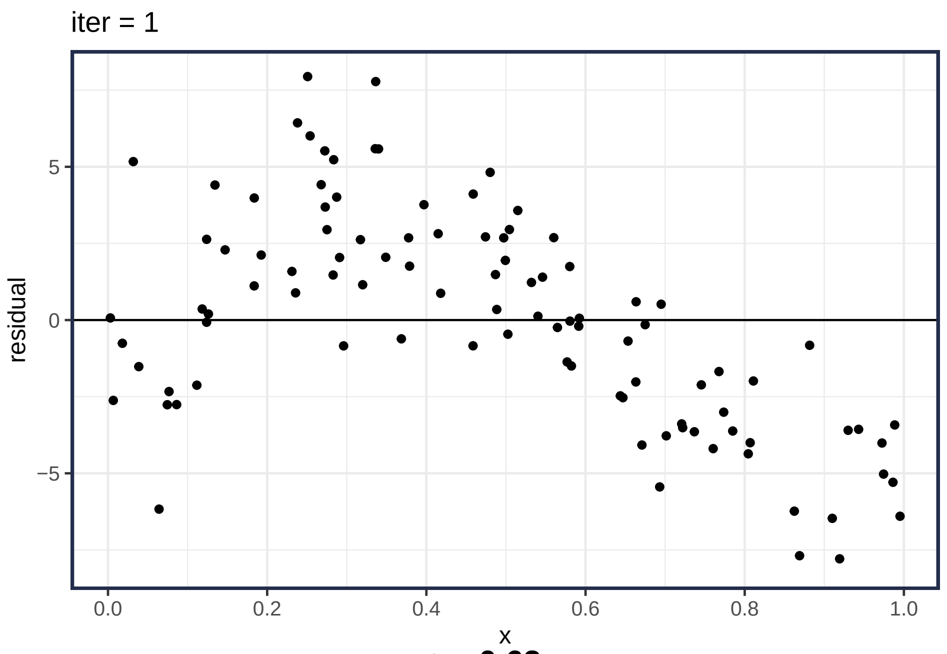
```r
#: L2 Boost Algorithm
library(rpart)

# L2boost()
#-------------------------------------------------------------------------------#
# L2 boosted trees (boosted regression trees)
# Inputs:
#  x,y: training data. x should be data frame or matrix, y a vector
#  xtest optional test data (data frame or matrix)
#  M: number of iterations
#  depth: tree depth. depth = 2 gives 4 leaf nodes.
#  nu: shrinkage parameter
# Outputs:
#  YHAT: matrix of in-sample predictions (predicting x)
#  R: matrix of residuals
#  YHAT.test: matrix of predictions for xtest
#  TREE: list of rpart trees
#-------------------------------------------------------------------------------#

L2boost <- function(x, y, xtest=NULL, M=100, depth=2, nu=.1){

  #- use training data if test data is not specified
  if(is.null(xtest)) {
    xtest = x
  }

  #- storage
  n = length(y)
  R = YHAT =  matrix(NA_real_, n, M)
  YHAT_test = matrix(NA_real_, nrow(xtest), M)
  colnames(YHAT) = colnames(YHAT_test) = colnames(R) = paste0("iter = ", 1:M)
  TREE = vector("list", M)
  names(TREE) = paste0("iter = ", 1:M)

  #-- 1) initialize model with mean
  mu = mean(y)
  yhat = rep(mu, nrow(x))
  yhat_test = rep(mu, nrow(xtest))

  for(m in 1:M){
    #-- 2a) Calculate Residuals
    r = y - yhat
    R[,m] = r

    #-- 2b) Fit regression tree
    tree = rpart(r ~ ., data = x,
                 maxdepth = depth,  # control tree depth
                 cp = -1,           # no pruning
                 minsplit = 0,      # allow all splits
                 minbucket = 1,     # no minimum on leaf size
                 method = "anova",  # least-squares loss function
                 xval = 0)          # no cross-validation

    TREE[[m]] = tree

    #-- 2c) Update model
    yhat = yhat + nu*predict(tree, x)
    YHAT[, m] = yhat

    yhat_test = yhat_test + nu*predict(tree, xtest)
    YHAT_test[, m] = yhat_test

  }

  #-- 3) Output
  return(list(YHAT=YHAT, R=R, YHAT.test=YHAT_test, TREE=TREE))
}
```

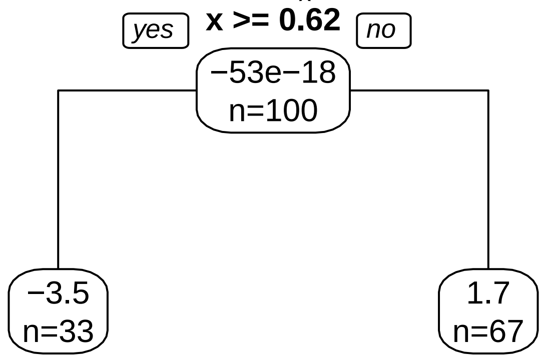
```r
#: Data Generation
n = 100                                    # number of observations
generate_x <- function(n) runif(n)         # U[0,1]
f <- function(x) 1 + 2*x + 5*sin(5*x)      # true mean function
sd = 2                                     # stdev for error

set.seed(825)                              # set seed for reproducibility
x = generate_x(n)                          # get x values
y = f(x) + rnorm(n, sd=sd)                 # get y values
data_train = data.frame(x, y)              # training data
x_eval = seq(0, 1, length=500)             # evaluation points
```

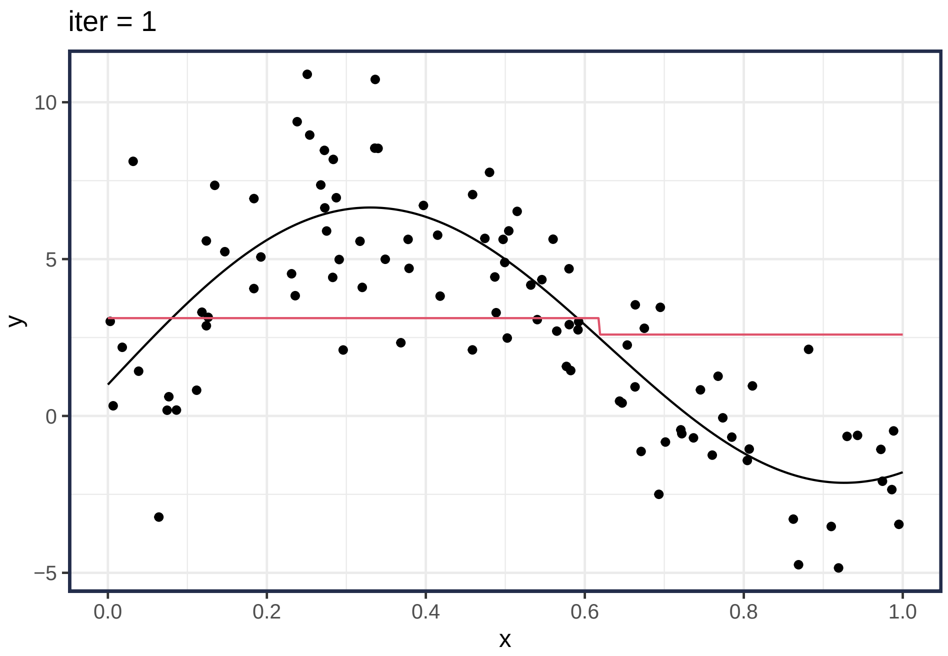
```r
#: Implement L2 boosting
L2 = L2boost(data.frame(x), y, xtest=data.frame(x=x_eval),   # data
             depth = 1, M = 100, nu = .1)              # tuning parameters
```

```r
#: Plotting
library(tidyverse)    # for ggplot2 package
library(rpart.plot)   # for prp()

# set iteration
i = 1

# Residual Plot
ggplot(data_train, aes(x)) +
  geom_point(aes(y = L2$R[,i]), col="black") +
  geom_hline(yintercept=0, col="black")  +
  scale_x_continuous(breaks=seq(0, 1, by=.20)) +
  coord_cartesian(ylim=c(-8, 8)) +
  labs(y="residual", title=colnames(L2$R)[i])
```

```r
# Tree
prp(L2$TREE[[i]], type=1, extra=1, branch=1, roundint=FALSE)

# Model prediction
ggplot(data_train, aes(x, y)) +
  geom_point() +
  annotate("line", x=x_eval, y=f(x_eval), color = "black") +
  geom_line(data=tibble(x=x_eval, y=L2$YHAT.test[,i]), col=2) +
  scale_x_continuous(breaks=seq(0, 1, by=.20)) +
  labs(title=colnames(L2$R)[i])
```






iter = 1
y
x
10
5
0
−5
0.0
0.2
0.4
0.6
0.8
1.0