Boosting

AdaBoost, Gradient Boosting, XGBoost

EN5422/EV4238 | Fall 2023

w10_boosting.pdf

(Week 10 – 1/1)

Contents

1 BOOSTING 2

2 ADABOOST 3

2.1 AdaBoost Algorithm 4

2.1.1 Illustration with Stumps (depth=1, n_nodes=2) 6

2.1.2 Illustration with Stumps (depth=2, n_nodes=4) 9

2.2 AdaBoost Details 12

2.3 Python library 14

2.4 AdaBoost Summary 17

3 GRADIENT BOOSTING 18

3.1 Gradient Descent 18

3.2 L_2 Boosting 18

3.2.1 Illustration using stumps (depth=1, n_nodes=2, v=0.1) 21

3.3 GBM (Gradient Boosting Machine) 22

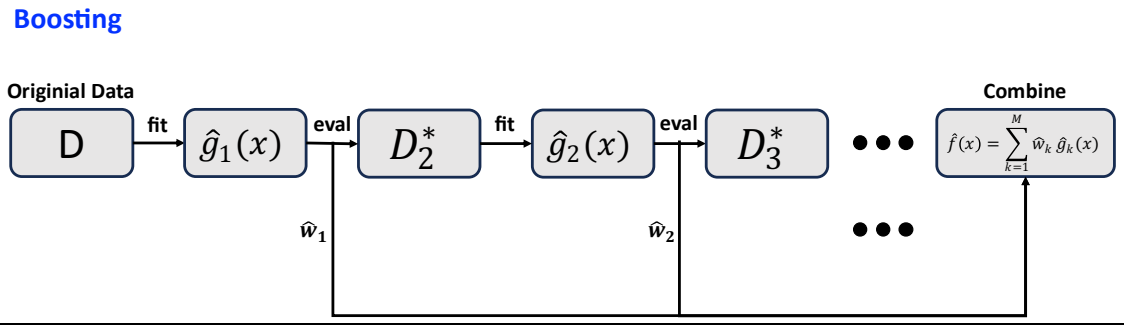
3.4 xgboost (Extreme Gradient Boosting) 23

3.5 CatBoost 24

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 k^{th} base learner ($\hat{a}_k \geq 0$)
  - $\hat{g}_k$ is the prediction from the k^{th} base learner

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

- 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.

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

- 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

Adaptive Boosting (AdaBoost) was motivated by the idea that many *weak* learners 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 threshold are set at $p > 0.50$
  - 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 $\hat{\boldsymbol{y}} = \hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$ is

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

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* $w_i = \frac{1}{n}$ 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

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

   Note: $0 \leq e_k \leq 0.5$ 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: $0 \leq a_k \leq \infty$.

   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.

$$\tilde{w}_i = w_i \cdot \exp\left(\hat{a}_k \cdot 1\big(y_i \neq \hat{g}_k(x_i)\big)\right)$$

$$= \begin{cases} w_i \left(\frac{1 - e_k}{e_k}\right) & \text{if obs } i \text{ is misclassified} \\ w_i & \text{if obs } i \text{ is correctly classified} \end{cases}$$

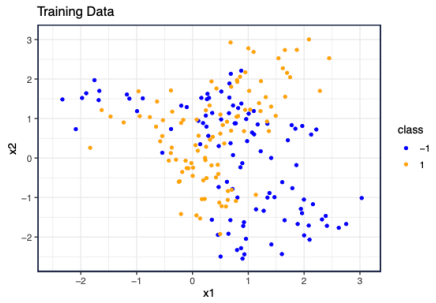
$$w_i = \frac{\tilde{w}_i}{\sum_{j=1}^n \tilde{w}_j} \text{ (re-normalize weights)}$$

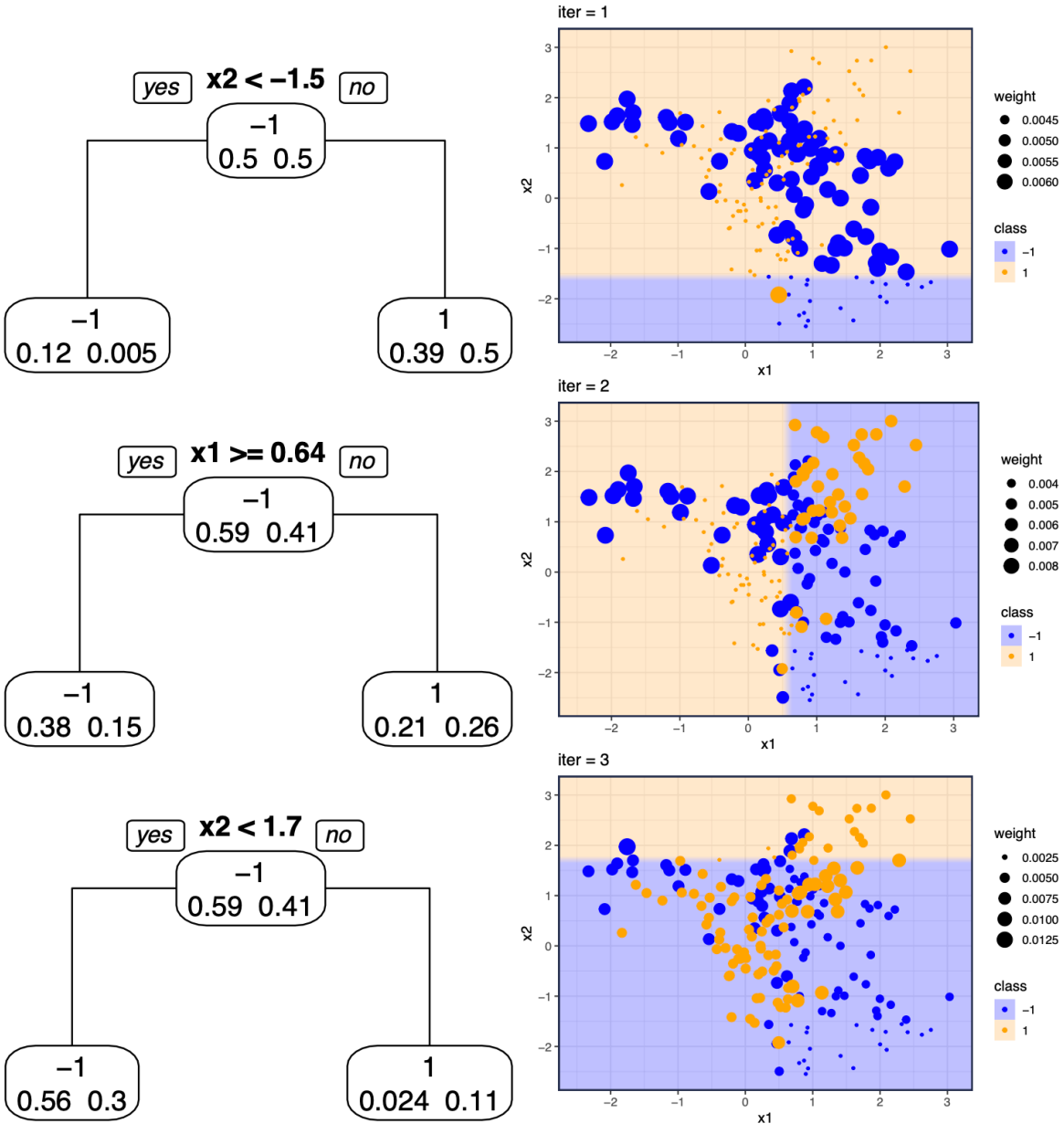
3. Output final ensemble $\hat{f}_M(x)$

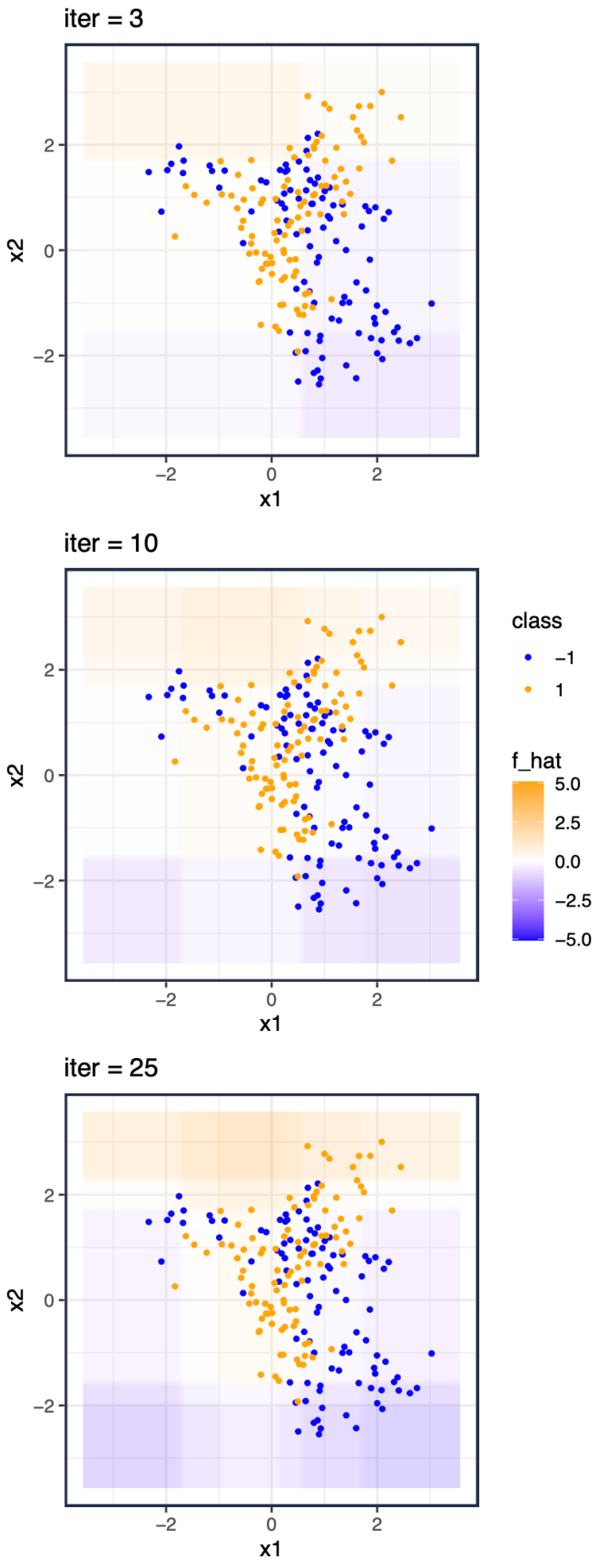
$$\hat{f}_M(x) = \sum_{k=1}^{M} \hat{a}_k \hat{g}_k(x)$$

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

2.1.1 Illustration with Stumps (depth=1, n_nodes=2)



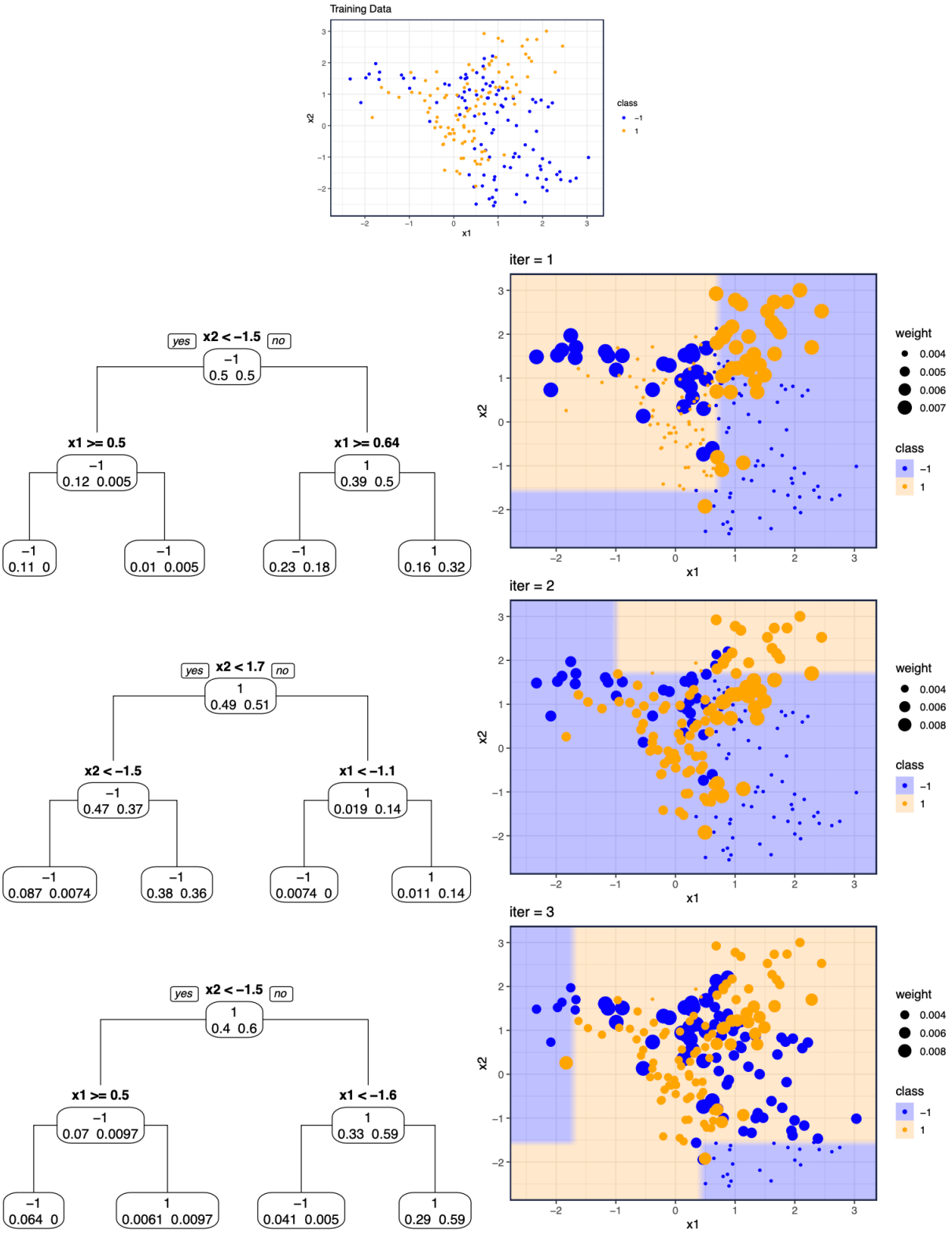


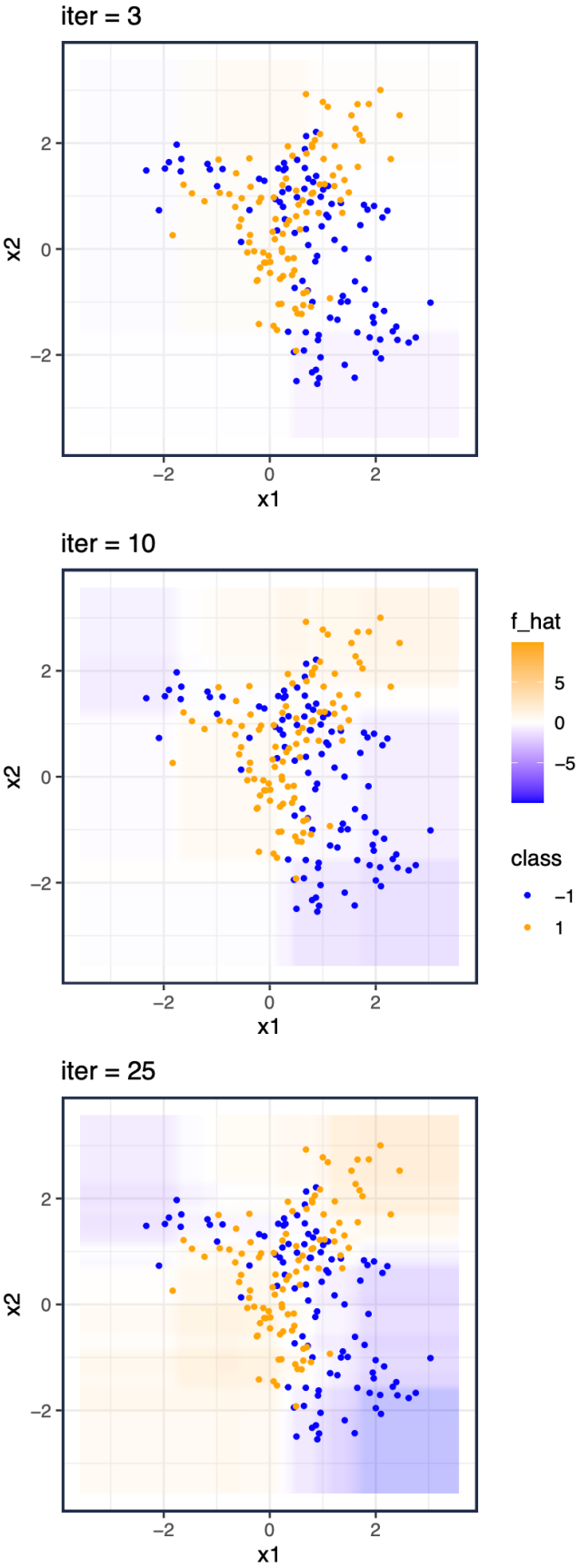

iter = 3
iter = 10
iter = 25
x1
x2
class
−1
1
f_hat
5.0
2.5
0.0
−2.5
−5.0

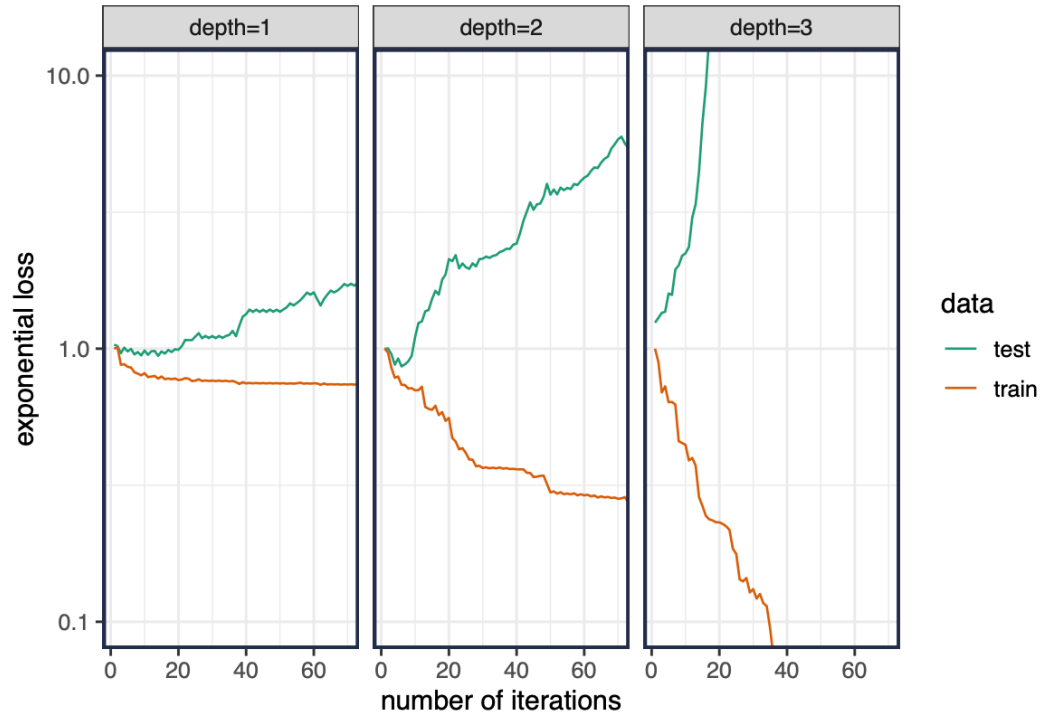
Interpretation

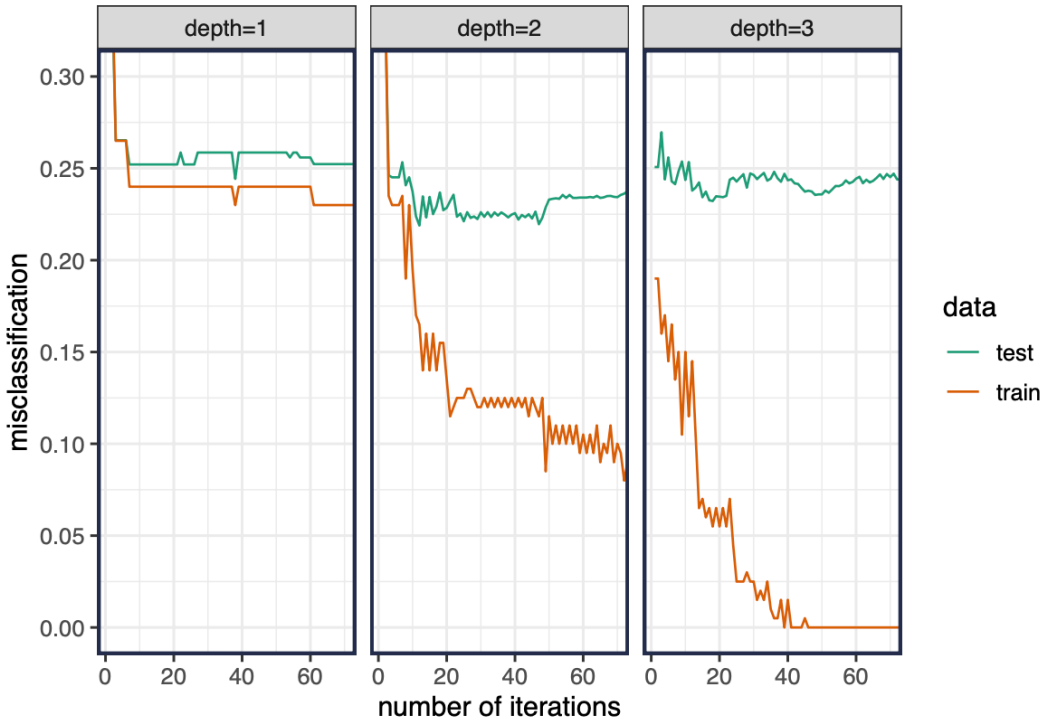
1. **Decision Stump**: On the left side of the image, there is a simple decision tree called a stump. This stump is a part of the ensemble that AdaBoost will create. It makes a decision based on a threshold value of one of the features (`x2`). If `x2` is less than -1.5, it predicts class `-1`, otherwise, it predicts class `1`. Below each decision, there are two numbers. These numbers are the probabilities of the stump predicting each class or the weights assigned to that particular prediction. For example, for `x2 <= -1.5`, the probability or weight for predicting class `-1` is 0.12, and for the wrong class, it is 0.005.
2. **Weighted Scatter Plot**: On the right side, there is a scatter plot with two iterations of AdaBoost shown. The plot displays data points with two features `x1` and `x2`, and each point is color-coded and sized according to its class and weight, respectively. The points belonging to one class are blue and the other class are orange. The weight of each point (signifying its importance in the AdaBoost algorithm) is indicated by the size of the point. Larger points have a higher weight, meaning the algorithm will focus more on correctly classifying them in the next iteration. The two iterations (`iter = 1` and `iter = 2`) are probably showing how the weights of the points are updated after each round of boosting. The blue shaded area might indicate the region where the current stump is predicting class `-1`.

2.1.2 Illustration with Stumps (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
10.0
1.0
0.1
exponential loss
0
20
40
60
number of iterations
data
test
train


depth=1
depth=2
depth=3
0.30
0.25
0.20
0.15
0.10
0.05
0.00
misclassification
0
20
40
60
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:

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

Interpretation
f represents the combined decision function of the ensemble of weak learners. AdaBoost builds this ensemble by iteratively adding weak learners that are trained to focus on the examples that the current ensemble finds most difficult. The outcome variable yy takes values in $\{-1,1\}$, which correspond to the two classes that AdaBoost is trying to distinguish between. The loss function L(y, f) used by AdaBoost is the exponential loss, given by e^{-yf}. This loss function quantifies how well the ensemble ff is doing at classifying a given example with label y. Specifically: • If y = +1 and f (the decision function) produces a large positive number, then e^{-f} will be small, indicating a low loss, which is good because it means ff is confident in the correct classification. • Conversely, if f produces a large negative number when y = +1, the loss e^{-f} will be large, indicating a high loss, which is bad because it means f is confident in the wrong classification. • If y = −1, the roles are reversed: a large positive f leads to a high loss efef, and a large negative ff leads to a low loss e^{f}. In each iteration of AdaBoost, a weak learner is added to the ensemble to minimize this exponential loss. The final decision function f is a weighted sum of the decision functions of the individual weak learners. The weights in this sum are determined by how well each weak learner does on the weighted training data provided to it. The training data weights are updated on each iteration to give higher weights to the examples that were misclassified by the current ensemble, thus encouraging the new weak learner to focus on those difficult examples.

- AdaBoost estimates the probability that y = +1 as

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

where $p(x) = \Pr(Y = +1 \mid X = x)$ and $f_M(x)$ is AdaBoost's final model

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

$$\hat{f}_M(x) = \frac{1}{2}\log\left(\frac{\hat{p}(x)}{1-\hat{p}(x)}\right) = \frac{1}{2}\operatorname{logit}\hat{p}(x)$$

- This shows that the final model $f_M(x)$ of AdaBoost is an estimate of half the logit of the estimated probability $\hat{p}(x)$. In other words, AdaBoost is implicitly fitting the log odds of the positive class, and through the iterative process of boosting, it is refining this estimate.

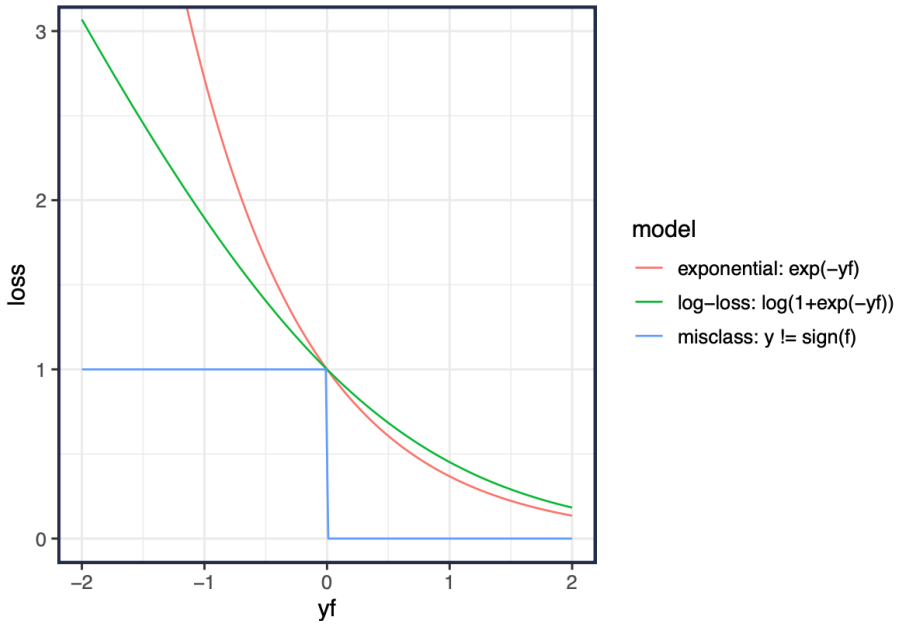
- The redefinition of the AdaBoost model in terms of the logit function is significant because it connects AdaBoost to logistic regression, revealing that AdaBoost can be seen as fitting an *additive logistic regression model*.

Note

- **Probabilistic Interpretation**: Redefining the AdaBoost model with the logit term gives a probabilistic interpretation to the outputs of AdaBoost. This allows the outputs to be interpreted as probabilities rather than just classifications, which can be useful for decision-making processes that require understanding the confidence of predictions.

- **Connection to Logistic Regression**: Logistic regression is a well-studied statistical method that models the probabilities of binary outcomes. By connecting AdaBoost to logistic regression through the logit link function, we can better understand the behavior of AdaBoost in the context of statistical learning theory.

- **Loss Function Interpretation**: The logit transformation shows that the exponential loss function used by AdaBoost is closely related to the logistic loss function used in logistic regression. This allows for the use of well-established methods for analysis and diagnostic checks from the logistic regression literature to be applied to models generated by AdaBoost.



- 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 Python library

In Python, AdaBoost's variations is available through the scikit-learn library.

- For a comprehensive understanding of the model variants – including Discrete, Real, and Gentle AdaBoost, as well as LogitBoost – refer to the seminal work by Friedman, J., Hastie, T., and Tibshirani, R. (2000) titled "Additive Logistic Regression: A Statistical View of Boosting," published in the Annals of Statistics, 28(2), 337-374.

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* $w_i = \frac{1}{n}$ for all *i*
2. For *k* = 1 to *M*:
   a. Fit a model $\hat{g}_k(x)$ that maps (x_i, w_i) to estimate a probability $\hat{p}_k(x) = \widehat{Pr}(Y = 1|X = x)$. 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.

$$\tilde{w}_i = w_i \cdot \exp\left(-y_i \hat{f}_m(x_i)\right)$$

$$w_i = \frac{\tilde{w}_i}{\sum_{j=1}^n \tilde{w}_i} \text{ (re-normalize weights)}$$

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 $\hat{p}(x) = \frac{e^{2\hat{f}(x)}}{1+e^{2\hat{f}(x)}}$ 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* $w_i = \frac{1}{n}$ 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 estimate 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.

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

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

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

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

• Or remap to a probability $\hat{p}(x) = \frac{e^{2\hat{f}(x)}}{1+e^{2\hat{f}(x)}}$ for thresholding

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 $y_i^* = \frac{y+1}{2} \in \{0, 1\}$

Algorithm:

1. Initialize *observation weights* $w_i = \frac{1}{n}$ 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 $\hat{f}_k(x) = \hat{f}_{k-1}(x) + \frac{\hat{g}_k(x)}{2}$ and $p_i = \frac{e^{2\hat{f}_k(x)}}{1+e^{2\hat{f}_k(x)}}$
3. Output final ensemble $\hat{f}_M(x)$

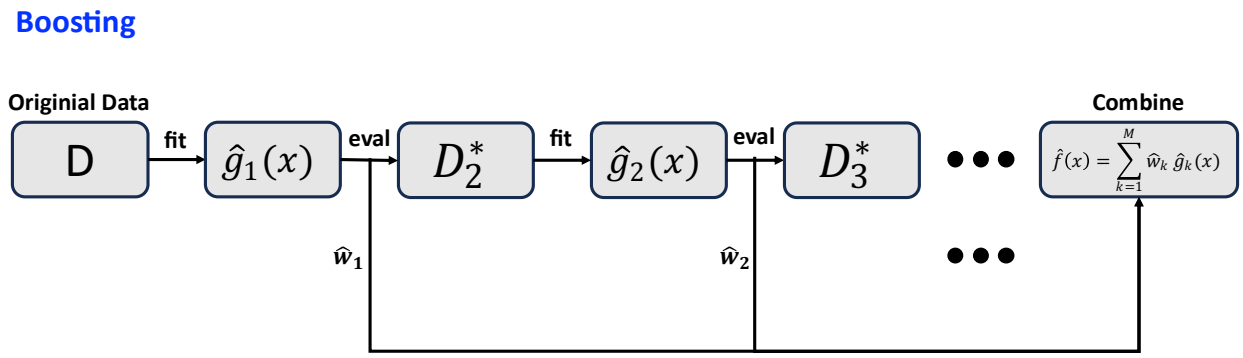
$$\hat{f}_M(x) = \sum_{k=1}^{M} \frac{1}{2}\hat{g}_k(x)$$

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

Python Code Implementation

```
from sklearn.ensemble import AdaBoostClassifier, AdaBoostRegressor
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
# Generate a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=20, random_state=42)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
# Initialize models
clf_discrete =
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=1),
                                   n_estimators=50,
                                   algorithm='SAMME')
clf_real =
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=1),
                                n_estimators=50,
                                algorithm='SAMME.R')
reg_gentle =
AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_depth=1),
                                 n_estimators=50,
                                 loss='linear')
clf_basic = AdaBoostClassifier(n_estimators=50)
```

2.4 AdaBoost Summary



AdaBoost (Adaptive Boosting) is an ensemble technique that combines multiple "weak learners" to create a strong classifier. The key concept in AdaBoost is to adaptively change the weights of training data points based on the performance of the previously built models. Here's how it generally works:

1. **Initialization**: Each data point in the training set is initially assigned the same weight, indicating that initially, each point is equally important in building the model.
2. **Iterative Training**: AdaBoost then iteratively trains weak learners (usually decision stumps - decision trees with a single split).
3. **Weight Update**: After training a weak learner, AdaBoost increases the weights of the misclassified points. That is, if a point is misclassified by the current learner, its weight is increased for the next round of training. This means that in the next round, the new learner focuses more on these harder, misclassified points.
4. **Learner Weighting**: Each weak learner is assigned a weight based on its accuracy, with more accurate learners given more weight in the final decision-making process. This weight is based on the error rate of the learner.
5. **Final Model**: The final model is an ensemble of these weak learners, where each learner votes to predict the outcome. The final prediction is a weighted vote of all these learners.

In summary, in AdaBoost, weights are applied in two places:

- **Training Data Weights**: Updated after each iteration to give more importance to misclassified points by the current learner.
- **Learner Weights**: Determined based on the accuracy of each learner and used in the final weighted voting process.

By focusing on the points that are harder to classify and giving more say to the more accurate learners, AdaBoost builds a robust model that often performs well on a variety of datasets.

3 Gradient Boosting

The boosting model:

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

Sequential Fitting

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

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*:

$$f_{k+1} = f_k - v_k L'(f_k)$$

  - $v_k > 0$ 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 squared 3error loss function

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

- The *negative gradients* are

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

- L_2 boosting is simply re-fitting to the residuals.

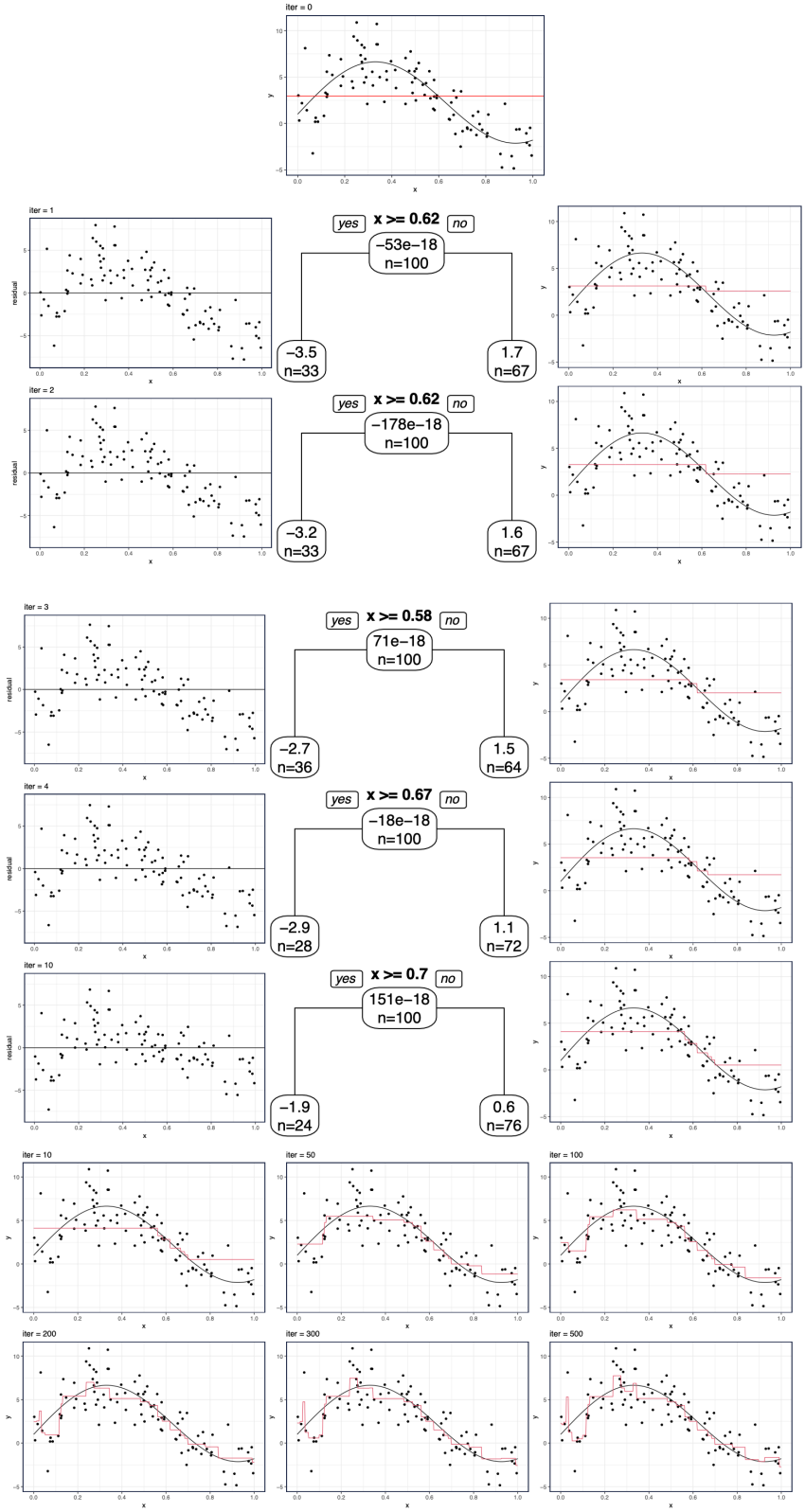
Algorithm: L_2 Boosting

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 leaner (e.g., regression trees) to the residuals $\{(x_i, r_i)\}_i^n$ to get the model $\hat{g}_k(x)$
   c. Update the overall model $\hat{f}_k(x) = f_{k-1}(x) + v\hat{g}_k(x)$
      - $0 \leq v \leq 1$ is the step-size (shrinkage)
3. Final model is $\hat{f}_M(x) = \bar{y} + \sum_{k=1}^{M} v\hat{g}_k(x)$

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

L_2 Boosting Example

3.2.1 Illustration using stumps (depth=1, n_nodes=2, v=0.1)



3.3 GBM (Gradient Boosting Machine)

3.3.1 Python Package for GBM

- **Python Package**: `scikit-learn`
- **GBM Documentation**: Refer to the `scikit-learn` documentation for Gradient Boosting.

3.3.2 GBM Characteristics in Python

- GBM is a first-order approach. It does not consider the Hessian.
- **Model/Tree Tuning Parameters**:
  - **Tree Depth** (`max_depth`):
    - Grows trees to a depth specified by `max_depth`, unless there are not enough observations in the terminal nodes.
  - **Minimum Number of Observations in Terminal Nodes** (`min_samples_leaf`):
    - The minimum number of samples required to be at a leaf node.
  - **Sub-sampling** (`subsample`):
    - Implements Stochastic Gradient Boosting.
    - Samples (without replacement) a fraction of the data at each iteration.
  - **Loss Function** (`loss`):
    - The loss function is determined by the `loss` argument.
    - Use `loss="squared_error"` for squared error.
    - Other options include `loss="deviance"` (for logistic regression), `loss="exponential"` (for AdaBoost-like loss), etc.

3.3.3 Boosting Tuning Parameters in Python

- **Number of Iterations/Trees** (`n_estimators`):
  - Use cross-validation to find the optimal value.
  - `scikit-learn` does not have a direct equivalent of `gbm.perf()`, but model performance can be assessed using cross-validation techniques.
- **Shrinkage Parameter** (`learning_rate`):
  - Set small, but note that smaller the `learning_rate`, the more iterations/trees will be needed.
  - Values typically range from 0.01 to 0.1.
- **Cross-validation**:
  - While `scikit-learn`'s GBM does not have built-in cross-validation for model training, you can use `GridSearchCV` or `cross_val_score` for this purpose.
  - There's no direct way to manually set the folds in the GBM model, but you can control them in `GridSearchCV` or `cross_val_score`.

3.3.4 Computational Settings in Python

- **Number of Cores** (`n_jobs`):

- The `n_jobs` parameter in `scikit-learn` allows you to specify the number of cores to use for training. Set `n_jobs=-1` to use all available cores.
- This is particularly useful when implementing cross-validation.

3.4 xgboost (Extreme Gradient Boosting)

3.4.1 Python Package for xgboost

- **Python Package**: `xgboost`
- **xgboost Documentation**: Refer to the official `xgboost` Python package documentation.
- **xgboost Model**: The Python implementation of the XGBoost model.
- **xgboost Paper**: The original paper on XGBoost can provide in-depth theoretical and practical insights.

Model/Tree Tuning Parameters

- **Different Base Learners** (`booster`):
  - `gbtree`: Uses tree-based models.
  - `gblinear`: Creates a generalized linear model (forward stagewise linear model).
- **Tree Building** (`tree_method`):
  - To speed up fitting, only consider making splits at certain quantiles of the input vector (rather than considering every unique value).
- **Sub-sampling** (`subsample`):
  - Implements Stochastic Gradient Boosting.
  - Samples (without replacement) a fraction of the data at each iteration.
- **Feature Sampling** (`colsample_bytree`, `colsample_bylevel`, `colsample_bynode`):
  - Similar to the technique used in Random Forest, features/columns are subsampled.
  - You can subsample features for each tree, level, or node.

3.2.1.1 Model Complexity Parameters

- **Tree Depth** (`max_depth`):
  - Grows trees to a depth specified by `max_depth`.
  - Trees may not reach `max_depth` if `gamma` or `min_child_weight` arguments are set.
- **Minimum Number of Observations in Terminal Nodes** (`min_child_weight`):
  - The minimum sum of instance weight (hessian) needed in a child.
- **Pruning** (`gamma` or `min_split_loss`):
  - The 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`: L2 regularization term on weights (equivalent to Ridge regression).
  - `alpha`: L1 regularization term on weights (equivalent to Lasso regression).

3.4.2 Loss Function (objective)

- **Loss Function Configuration**:
  - The loss function in XGBoost is determined by the `objective` parameter.
  - Use `objective="reg:squarederror"` for squared error in regression tasks.
  - Other options include:
    - `objective="binary:logistic"` for binary logistic regression.
    - `objective="count:poisson"` for Poisson regression.
    - `objective="rank:pairwise"` for ranking/LambdaMart.
    - And many more for different types of regression and classification tasks.

3.4.3 Boosting Tuning Parameters

- **Shrinkage Parameter** (`eta` or `learning_rate`):
  - Typically set small; note that a smaller `eta` requires more iterations/trees.
- **Number of Iterations/Trees** (`num_boost_round` in Python):
  - The optimal number of boosting rounds can be found using cross-validation.
- **Cross-validation** (`xgboost.cv`):
  - XGBoost provides built-in cross-validation capabilities.
  - It allows manually setting the folds.

3.2.1.2 Computational Settings

- **Number of Threads** (`nthread`):
  - Specifies the number of threads to be used in training.
- **GPU Support**:
  - XGBoost offers GPU support for faster computation.
  - Refer to XGBoost GPU documentation for more details.

3.5 CatBoost

- **Python Package for CatBoost**:
  - CatBoost can be used in Python via its package.
  - For installation and documentation, visit the CatBoost Python package page.
  - The CatBoost documentation provides detailed guidance on using the package effectively for various machine learning tasks.