Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X$, $y_i \in Y = \{-1, +1\}$

Initialize $D_1(i) = 1/m$

For t = 1,..., T:

- Train weak learner using distribution D_t
- Get weak hypothesis h_t: $X \rightarrow \{-1, +1\}$ with error

$$\epsilon_t = Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$$

- Choose

$$\alpha_t = \frac{1}{2}\log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

- Update:

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \exp(-\alpha_t) & \text{if } h_t(x_i) = y_i \\ \exp(\alpha_t) & \text{if } h_t(x_i) \neq y_i \end{cases}$$

$$= \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution)

Output

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

9. Adaptive Boosting

(AdaBoost)

www.youtube.com/@meanxai

MX-AI

MX-AI

▪ Adaptive Boosting (AdaBoost) : Overview

- AdaBoost is a type of boosting algorithm proposed by Yoav Freund at AT&T Bell Labs in 1995. Boosting is a method of gradually strengthening a weak learner model. In the estimation stage, the results of multiple models are combined.
- AdaBoost is an algorithm that increases accuracy by focusing more on misclassified data points during training stage.

Boosting a weak learning algorithm by majority

Yoav Freund
AT&T Bell Laboratories
New Jersey
January 17, 1995

Abstract

We present an algorithm for improving the accuracy of algorithms for learning binary concepts. The improvement is achieved by combining a large number of hypotheses, each of which is generated by training the given learning algorithm on a different set of examples. Our algorithm is based on ideas presented by Schapire and represents an improvement over his results, The analysis of our algorithm provides general upper bounds on the resources required for learning in Valiant's polynomial PAC learning framework, which are the best general upper bounds known today. We show that the number of hypotheses that are combined by our algorithm is the smallest number possible. Other outcomes of our analysis are results regarding the representational power of threshold circuits, the relation between learnability and compression, and a method for parallelizing PAC learning algorithms. We provide extensions of our algorithms to cases in which the concepts are not binary and to the case where the accuracy of the learning algorithm depends on the distribution of the instances.

Journal of Japanese Society for Artificial Intelligence, 14(5):771-780, September, 1999.
(In Japanese, translation by Naoki Abe.)

A Short Introduction to Boosting

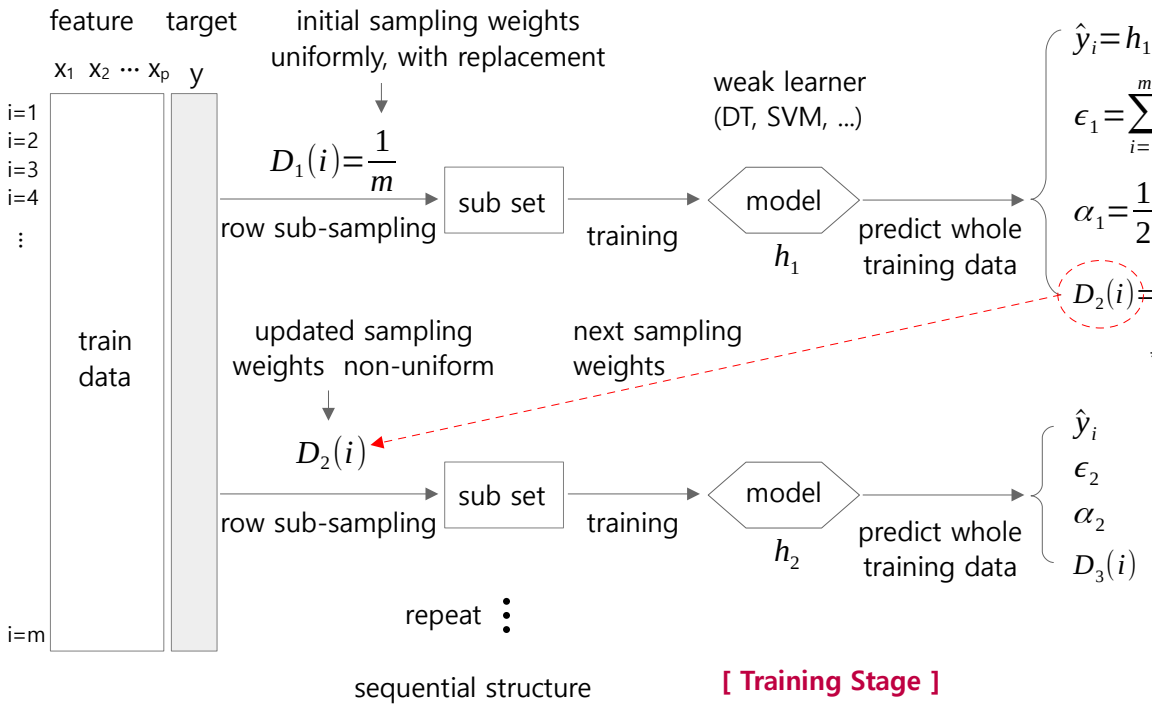
Yoav Freund Robert E. Schapire
AT&T Labs - Research
Shannon Laboratory
180 Park Avenue
Florham Park, NJ 07932 USA
www.research.att.com/~{yoav, schapire}
{yoav, schapire}@research.att.com

Abstract

Boosting is a general method for improving the accuracy of any given learning algorithm. This short overview paper introduces the boosting algorithm AdaBoost, and explains the underlying theory of boosting, including an explanation of why boosting often does not suffer from overfitting as well as boosting's relationship to support-vector machines. Some examples of recent applications of boosting are also described.

MX-AI

▪ AdaBoost : Boosting – Training and Prediction stage



- A weak learner is a model that performs slightly better than a random learner. For example, for DT, use max_depth=1 or 2, and for SVM, adjust C and γ so that the decision boundary is not detailed.
- The output of h with large ε is unreliable, so it is multiplied by small α. The output of h with small ε is reliable, so it is multiplied by large α.

MX-AI

▪ AdaBoost : Algorithm (binary classification), (y = {-1, +1})

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X$, $y_i \in Y = \{-1, +1\}$ ← Binary classification

Initialize $D_1(i) = 1/m$ ← Initialize the sampling weights to 1/m, uniform distribution.

For t = 1,..., T:

- Train weak learner using distribution D_t ← Perform sampling of training data according to weights D_t to create a subset. Use the subset to train a model, weak learner, h_t.
- Get weak hypothesis $h_t: X \rightarrow \{-1, +1\}$ with error ← Use the model (h_t) to predict the target class for the entire training data.

$$\epsilon_t = Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$$ ← $$\epsilon_t = \sum_{i=1}^{m} D_t(i) \cdot I(y_i \neq \hat{y}_i)$$ ← calculate the error = (1-accuracy)

- Choose

$$\alpha_t = \frac{1}{2} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$ ← For $\alpha > 0$, $\varepsilon < 0.5$. That is, h should be slightly better than a random prediction.

- Update:

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \exp(-\alpha_t) & \text{if } h_t(x_i) = y_i \\ \exp(\alpha_t) & \text{if } h_t(x_i) \neq y_i \end{cases}$$ ← Update the sampling weight (D_t) of each data point for the next sampling.

$$= \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

- $h_t(x_i) = y_i$: $D_{t+1}(i)$ decreases. Well-matched data points are less likely to be selected for the next round.
- $h_t(x_i) \neq y_i$: $D_{t+1}(i)$ increases. Mismatched data points are more likely to be selected for the next round.

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution)

Output

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$ ← Input test data into the trained model (h_t) and synthesize the results for each h. α: the weight multiplied by each $h_t(x)$. inversely proportional to ε.

* Reference: Yoav Freund et, al., 1999, A Short Introduction to Boosting. Figure 1: The boosting algorithm AdaBoost

MX-AI

▪ Derivation of α and ε formula

▪ Reference : Raul Rojas, 2009, AdaBoost and the Super Bowl of Classifiers A Tutorial Introduction to Adaptive Boosting

Output: $H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$ ← 1999, Yoav Freund

$$C_{m-1}(x_i)=\alpha_1 k_1(x_i)+\alpha_2 k_2(x_i)+\ldots+\alpha_{m-1} k_{m-1}(x_i)$$

← Linear combination of weak learners.

α: weights; k_{m-1}: The (m-1)th weak learner

$\hat{y}_i$

$$C_m(x_i)=C_{m-1}(x_i)+\alpha_m k_m(x_i)$$

$$E=\sum_{i=1}^{N} \exp(-y_i \hat{y}_i)$$ ← total loss (exponential loss)

$y_i=\{-1,+1\}$

- $y_i=\hat{y}_i$ → E decreases
- $y_i\neq\hat{y}_i$ → E increases

$$E=\sum_{i=1}^{N} \exp\{-y_i(C_{m-1}(x_i)+\alpha_m k_m(x_i))\}$$

$$E=\sum_{i=1}^{N} w_i^{(m)} \exp\{-y_i \alpha_m k_m(x_i)\}$$ ← $w_i^{(m)}=\exp\{-y_i C_{m-1}(x_i)\}$

$$E=\sum_{y_i=k_m(x_i)} w_i^{(m)} e^{-\alpha_m}+\sum_{y_i\neq k_m(x_i)} w_i^{(m)} e^{\alpha_m}$$

$$E=W_c e^{-\alpha_m}+W_e e^{\alpha_m}$$

$$\frac{dE}{d\alpha_m}=-W_c e^{-\alpha_m}+W_e e^{\alpha_m}=0$$

$$\alpha_m=\frac{1}{2}\log\left(\frac{W_c}{W_e}\right)$$

$$\alpha_m=\frac{1}{2}\log\left(\frac{W-W_e}{W_e}\right)$$ ← $W=W_c+W_e$

$$\boxed{\alpha_m=\frac{1}{2}\log\left(\frac{1-\epsilon_m}{\epsilon_m}\right)}$$ ← $\epsilon_m=\frac{W_e}{W}$

- $\epsilon_m \leq 0.5$ ← by assumption
- $\epsilon_m=0.5$ → $\alpha=0$
- $\epsilon_m=0$ → $\alpha=\infty$

$$\boxed{\epsilon_m=\sum_{i=1}^{N} w_i^{(m)} \cdot I(y_i\neq\hat{y}_i)}$$

* the total cost is the weighted cost of all hits plus the weighted cost of all misses.

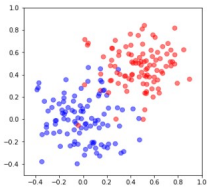
* α and ε are inversely proportional. If the m-th learner's error is large, α is applied as a small value to reduce the influence of the m-th learner, and vice versa.

MX-AI

- Implementation of AdaBoost using DecisionTreeClassifier, (y = {-1, +1})

```python
# [MXML-9-01] 1.AdaBoost(binary1).py
# [1] Yoav Freund et, al., 1999, A Short Introduction to Boosting
import numpy as np
import random as rd
from sklearn.datasets import make_blobs
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Create training data
x, y = make_blobs(n_samples=200, n_features=2,
                  centers=[[0., 0.], [0.5, 0.5]],
                  cluster_std=0.2, center_box=(-1., 1.))
y = y * 2 - 1   # [0, 1] --> [-1, 1]
m = x.shape[0]
R = np.arange(m)
T = 50

# [1] Figure 1: The boosting algorithm AdaBoost
# Given: (x1, y1), ..., (xm, ym) where xi ∈ X, yi ∈ Y = {-1, +1}
# Initialize D1(i) = 1/m
weights = [np.array(np.ones(shape=(m,)) / m)]
eps, alphas, models = [], [], []
for t in range(T):
    # sampling according to the weights
    s_idx = np.array(rd.choices(R, weights=weights[-1], k=m))
    sx = x[s_idx]      # sample x
    sy = y[s_idx]      # sample y

    # Train weak learner using distribution Dt. (Dt: weights)
    model = DecisionTreeClassifier(max_depth=2) # base learner
    model.fit(sx, sy)        # fit the model to sample data
```

$$\epsilon_t = \sum_{i=1}^{m} D_t(i) \cdot I(y_i \neq \hat{y}_i)$$



```python
    # Get weak hypothesis ht : X -> {-1, +1} with error
    y_pred = model.predict(x)  # predict entire training data
    i_not = np.array(y_pred != y).astype(int)  # I(y_pred ≠ y)
    eps.append(np.sum(weights[-1] * i_not))

    # Choose αt=(1/2)ln((1-εt)/εt).  (α: alpha, ε: eps)
    # For αt to be positive, εt must be less than 0.5.
    # If εt is greater than 0.5, it means it is worse than a
    # random prediction. If so, initialize the weights to 1/m
    # again.
    if eps[-1] > 0.5:
        weights.append(np.array(np.ones(shape=(m,)) / m))
        alphas.append(0.0)
        print('weight re-initialized at t =', t)
    else:
        alpha = 0.5 * np.log((1 - eps[-1]) / (eps[-1] + 1e-8))
        alphas.append(alpha)

        # Update Dt.
        new_weights = weights[-1] * np.exp(-alpha * y * y_pred)
        weights.append( new_weights / new_weights.sum())
    models.append(model)

# Output the final hypothesis:
x_test = np.random.uniform(-0.5, 1.5, (1000, 2))
H = np.zeros(shape=x_test.shape[0])
for t in range(T):
    h = models[t].predict(x_test)
    H += alphas[t] * h
y_pred = np.sign(H)
```

$$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

$$D_{t+1}(i) = \frac{1}{Z_t} D_t(i) \exp\left(-\alpha_t y_i h_t(x_i)\right)$$

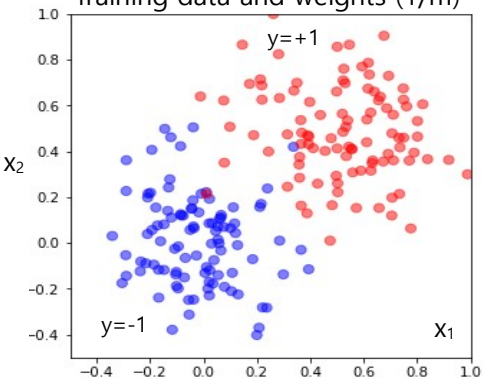
$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

MX-AI

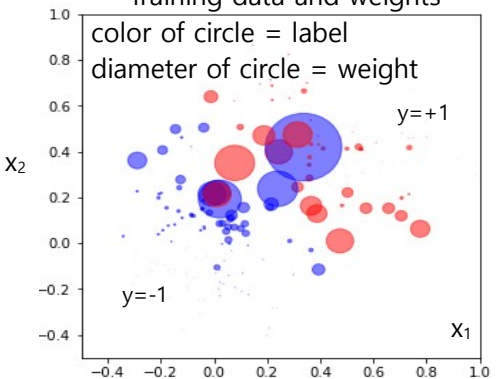
- Implementation of AdaBoost using DecisionTreeClassifier, (y = {-1, +1})

```
# visualize training data and the sampling weights
def plot_train(x, y, w):
    plt.figure(figsize=(5,5))
    color = ['red' if a == 1 else 'blue' for a in y]
    plt.scatter(x[:, 0], x[:, 1], s=w*10000, c=color, alpha=0.5)
    plt.xlim(-0.5, 1.0); plt.ylim(-0.5, 1.0)
    plt.show()

# visualize decision boundary
def plot_boundary(x, y, x_test, y_pred):
    plt.figure(figsize=(5,5))
    color = ['red' if a == 1 else 'blue' for a in y_pred]
    plt.scatter(x_test[:, 0], x_test[:, 1], s=100, \
                c=color, alpha=0.3)
    plt.scatter(x[:, 0], x[:, 1], s=80, c='black')
    plt.scatter(x[:, 0], x[:, 1], s=10, c='yellow')
    plt.xlim(-0.5, 1.0); plt.ylim(-0.5, 1.0)
    plt.show()

plot_train(x, y, w=np.array(np.ones(shape=(m,)) / m))
plot_train(x, y, w=weights[-1])
plot_boundary(x, y, x_test, y_pred)

# Check the changes in α (alpha), ε (eps).
# Check that ε are all less than 0.5 and that α and ε are
# inversely proportional.
plt.plot(eps, marker='o', markersize=4, c='red', lw=1,\
         label='epsilon')
plt.plot(alphas, marker='o', markersize=4, c='blue', lw=1, \
         label='alphas')
plt.legend()
plt.show()
```

Training data and weights (1/m)

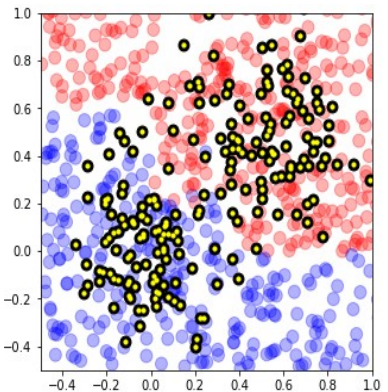


Training data and weights

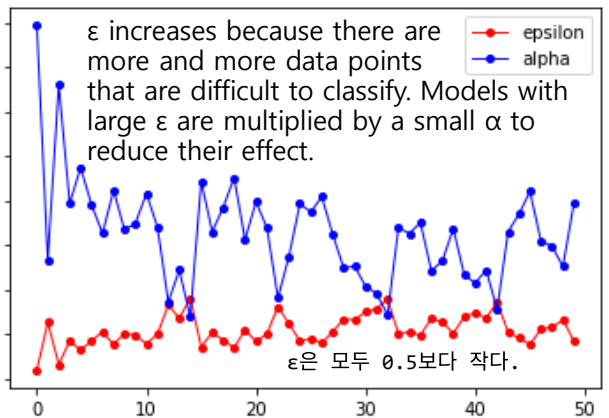


The sample weights of data points near the decision boundary are increasing.
This is because data points in this area are often misclassified.

Decision boundary



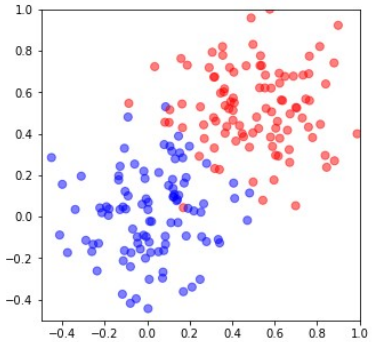
Change in ε, α



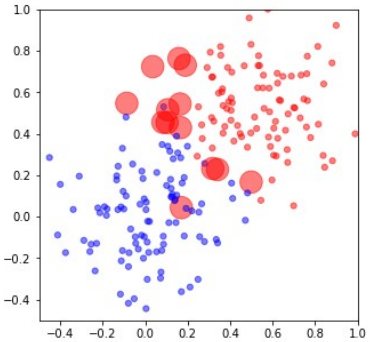
MX-AI

- Implementation of AdaBoost using DecisionTreeClassifier, (y = {-1, +1})

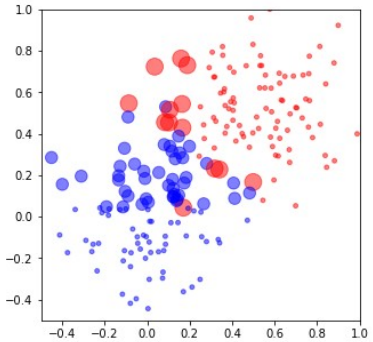
plot_train(x, y, w=weights[0])



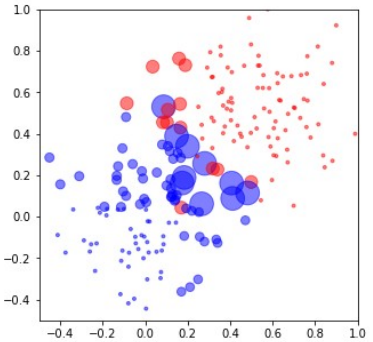
plot_train(x, y, w=weights[1])



plot_train(x, y, w=weights[2])

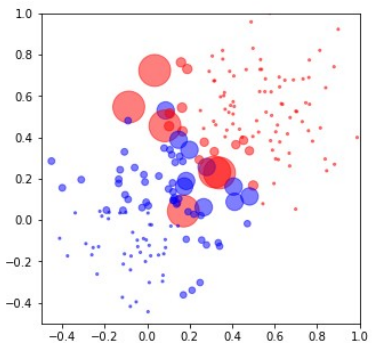


plot_train(x, y, w=weights[3])

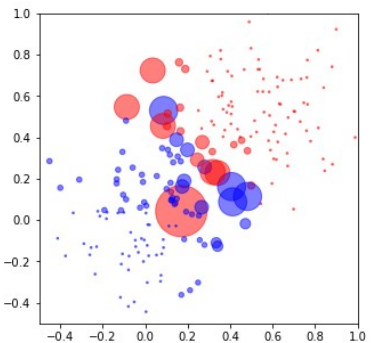


As t increases, the weights around the decision boundary increase. ← Misclassified data points are sampled more and trained more.

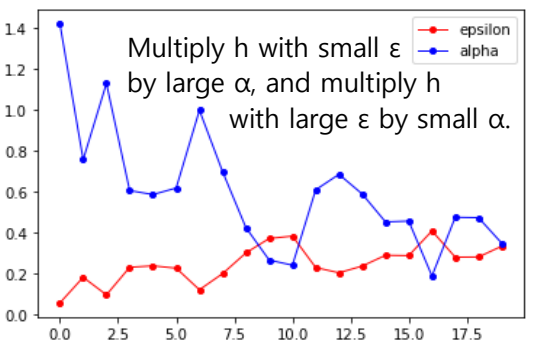
plot_train(x, y, w=weights[4])



plot_train(x, y, w=weights[5])



Changes ε, α



Prediction result

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Training frequently on misclassified samples can lead to overfitting, but as frequency increases (as weights increase) alpha decreases, which reduces the likelihood of overfitting.

MX-AI

▪ AdaBoost : Algorithm (binary classification), (y = {0, 1})

▪ Use y = {0, 1} instead of y = {-1, +1}. It can be easily extended to multiclass classification.

Algorithm 1: AdaBoost (Freund & Schapire 1997)

1. Initialize the observation weights $w_i = 1/n$, i=1,2,...,n
2. **For** m = 1 **to** M:

   a) Fit a classifier $T^{(m)}(x)$ to the training data using weights w_i

   b) Compute

$$err^{(m)} = \sum_{i=1}^{n} w_i \cdot I(c_i \neq T^{(m)}(x_i)) \Big/ \sum_{i=1}^{n} w_i$$

   c) Compute

$$\alpha^{(m)} = \log \frac{1 - err^{(m)}}{err^{(m)}}$$

   d) Set

$$w_i \leftarrow w_i \cdot \exp(\alpha^{(m)} \cdot I(c_i \neq T^{(m)}(x_i))), \quad i = 1,2,\ldots,n$$

   e) Re-normalize w_i

3. Output

$$C(x) = \underset{k}{argmax} \sum_{m=1}^{M} \alpha^{(m)} \cdot I(T^{(m)}(x) = k)$$

* Reference: Ji Zhu, et, al., 2006, Multi-class AdaBoost, Algorithm 1

In this form, multiple class classification is also possible, but α <0 problem occurs if $err^{(m)}$> 0.5. In binary classification, α <0.5 is possible even in weak learners, but in three classes, α <0 occurs easily because the accuracy of random prediction is 1/3 and $err^{(m)}$ = 0.67. → Need to be improved → SAMME algorithm in the next video.

It is always 1 because it is normalized.

Not multiplied by 1/2. The reason was not explained.

Use argmax instead of the sign function.

example

$T^{(m)}(x)$ $\quad I(T^{(m)}(x)=k)$

k=0 1 $\quad \alpha^{(m)}$

m=1: 0 → [1 0] * 0.5 = [0.5 0]
m=2: 0 → [1 0] * 0.7 = [0.7 0]
m=3: 1 → [0 1] * 0.2 = [0 0.2]
m=4: 0 → [1 0] * 0.6 = [0.6 0]
sum = [1.8 0.2]
argmax = 0

MX-AI

- AdaBoost : Algorithm (binary classification), (y = {0, 1})

```
# [MXML-9-02] 2.AdaBoost(binary2).py
# Using y = {0, 1} instead of y = {-1, +1}
#
# [1] Yoav Freund et, al., 1999, A Short Introduction to Boosting
# [2] Ji Zhu, et, al., 2006, Multi-class AdaBoost
import numpy as np
import random as rd
from sklearn.datasets import make_blobs
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

# Create training data y = {0, 1}
x, y = make_blobs(n_samples=200, n_features=2,
                  centers=[[0., 0.], [0.5, 0.5]],
                  cluster_std=0.2, center_box=(-1., 1.))
m = x.shape[0]
R = np.arange(m)
T = 50

weights = [np.array(np.ones(shape=(m,)) / m)]
eps = []      # epsilon history
alphas = []   # alpha history
models = []   # base learner models
for t in range(T):
    s_idx = np.array(rd.choices(R, weights=weights[-1], k=m))
    sx = x[s_idx]      # sample x
    sy = y[s_idx]      # sample y

    model = DecisionTreeClassifier(max_depth=2)
    model.fit(sx, sy)  # fit the model to sample data
    y_pred = model.predict(x)  # predict entire training data
    i_not = np.array(y_pred != y).astype(int)  # I(y_pred ≠ y)
    eps.append(np.sum(weights[-1] * i_not))

    if eps[-1] > 0.5:
        weights.append(np.array(np.ones(shape=(m,)) / m))
        alphas.append(0.0)
        print('weight re-initialized at t =', t)
    else:
        alpha = 0.5 * np.log((1 - eps[-1]) / (eps[-1] + 1e-8))
        alphas.append(alpha)

        # w_i ← w_i·exp(α^(m)·I(c_i≠T^(m)(x_i)))
        new_weights = weights[-1] * np.exp(alpha * i_not)
        weights.append(new_weights / new_weights.sum())
    models.append(model)

x_test = np.random.uniform(-0.5, 1.5, (1000, 2))

# C(x) = argmax_k Σ_{m=1}^{M} α^(m)·I(T^(m)(x)=k)
H = np.zeros(shape=(x_test.shape[0], 2))
for t in range(T):
    h = models[t].predict(x_test)
    oh = np.eye(2)[h]    # one-hot encoding
    H += alphas[t] * oh
y_pred = np.argmax(H, axis=1)
```

MX-AI

▪ Multiclass Classification – SAMME Algorithm (**S**tagewise **A**dditive **M**odeling using a **M**ulti-class **E**xponential loss function)

- In 2006, Ji Zhu et al. proposed the following algorithm to solve the multiclass problem of Freund's (1995) AdaBoost algorithm.
- The err(m) < 0.5 condition in Freund's model was modified to err(m) < 1 – 1/K (K: the number of classes). If K=3, then err(m) < 0.67.

Multi-class AdaBoost

Ji Zhu, Department of Statistics University of Michigan Ann Arbor, MI 48109
Saharon Rosset, Data Analytics Group IBM Research Center Yorktown Heights, NY 10598
Hui Zou, School of Statistics University of Minnesota Minneapolis, MN 55455
Trevor Hastie, Department of Statistics Stanford University Stanford, CA 94305
January 12, 2006

Abstract

Boosting has been a very successful technique for solving the two-class classification problem. In going from two-class to multi-class classification, most algorithms have been restricted to reducing the multi-class classification problem to multiple two-class problems. In this paper, we propose a new algorithm that naturally extends the original AdaBoost algorithm to the multiclass case without reducing it to multiple two-class problems. Similar to AdaBoost in the two-class case, this new algorithm combines weak classifiers and only requires the performance of each weak classifier be better than random guessing (rather than 1/2). We further provide a statistical justification for the new algorithm using a novel multi-class exponential loss function and forward stage-wise additive modeling. As shown in the paper, the new algorithm is extremely easy to implement and is highly competitive with the best currently available multi-class classification methods.

Algorithm 2: **SAMME**

1. Initialize the observation weights $w_i = 1/n$, i=1,2,...,n
2. **For** m = 1 **to** M:
   a) Fit a classifier $T^{(m)}(x)$ to the training data using weights w_i
   b) Compute
$$err^{(m)} = \sum_{i=1}^{n} w_i \cdot I\left(c_i \neq T^{(m)}(x_i)\right) \;/\; \sum_{i=1}^{n} w_i$$
   c) Compute
$$\alpha^{(m)} = \log\frac{1-err^{(m)}}{err^{(m)}} + \log(K-1)$$
   d) Set
$$w_i \leftarrow w_i \cdot \exp\left(\alpha^{(m)} \cdot I\left(c_i \neq T^{(m)}(x_i)\right)\right), \; i=1,2,\ldots,n$$
   e) Re-normalize w_i
3. Output
$$C(x) = \underset{k}{argmax} \sum_{m=1}^{M} \alpha^{(m)} \cdot I\left(T^{(m)}(x) = k\right)$$

This part was added to the Freund (1997) model.

For $\alpha^{(m)} > 0$, $err^{(m)} < 1 - \frac{1}{K}$

$$\log\frac{1-err^{(m)}}{err^{(m)}} + \log(K-1) > 0$$

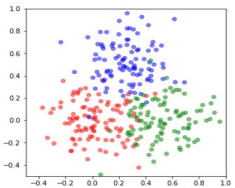
$$\frac{1-err^{(m)}}{err^{(m)}} > \frac{1}{K-1}$$

$$\frac{1}{err^{(m)}} > \frac{K}{K-1}$$

MX-AI

- Implementation of Multiclass AdaBoost using DecisionTreeClassifier, y = {0, 1, 2}

```
# [MXML-9-03] 3.AdaBoost(multiclass).py
# [1] Yoav Freund et, al., 1999, A Short Introduction to Boosting
# [2] Ji Zhu, et, al., 2006, Multi-class AdaBoost
import numpy as np
import random as rd
from sklearn.datasets import make_blobs
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_sp
import matplotlib.pyplot as plt

# Create training data. y = {0, 1, 2}
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=[[0., 0.], [0.25, 0.5], [0.5, 0.]],
                  cluster_std=0.18, center_box=(-1., 1.))

K = np.unique(y).shape[0]  # the number of target class = 3
m = x.shape[0]
R = np.arange(m)
T = 50

# [2] Algorithm 2: SAMME
weights = [np.array(np.ones(shape=(m,)) / m)]
eps = []      # epsilon history
alphas = []   # alpha history
models = []   # base learner models
for t in range(T):
    s_idx = np.array(rd.choices(R, weights=weights[-1], k=m))
    sx = x[s_idx]      # sample x
    sy = y[s_idx]      # sample y

    # Make sure to select at least one class of y.
    # Among y=0, 1, and 2, if only the data points with y=1 and 2
    # are selected, select one more data point with y=0.
```



```
    uy = np.unique(sy)
    if uy.shape[0] < K:
        unseen = list(set(np.unique(y)) - set(uy))
        for u in unseen:
            ui = rd.choices(np.where(y_train == u)[0], k=1)
            sx = np.vstack([sx, x_train[ui]])
            sy = np.hstack([sy, y_train[ui]])

    # base weak learner
    model = DecisionTreeClassifier(max_depth=1)
    model.fit(sx, sy)  # fit the model to sample data

    # calculate error (epsilon)
    y_pred = model.predict(x)     # predict entire training data
    i_not = np.array(y_pred != y).astype(int)  # I(y_pred ≠ y)
    eps.append(np.sum(weights[-1] * i_not))

    # calculate alpha using the error
    # For alpha to be positive, eps must be less than 1 - 1/K.
    # If eps is greater than 1 - 1/K, it means it is worse than a
    # random prediction. If so, initialize the weights to 1/m again.
    if eps[-1] > 1 - 1/K:
        weights.append(np.array(np.ones(shape=(m,)) / m))
        alphas.append(0.0)
        print('weight re-initialized at t =', t)
    else:
        alpha = np.log((1-eps[-1]) / (eps[-1]+1e-8)) + np.log(K - 1)
        alphas.append(alpha)

        new_weights = weights[-1] * np.exp(alpha * i_not)
        weights.append(new_weights / new_weights.sum())  # normalize
```

MX-AI

- Implementation of Multiclass AdaBoost using DecisionTreeClassifier, y = {0, 1, 2}

```
    models.append(model)

# prediction
x_test = np.random.uniform(-0.5, 1.5, (1000, 2))
H = np.zeros(shape=(x_test.shape[0], K))
for t in range(T):
    h = models[t].predict(x_test)
    oh = np.eye(K)[h]
    H += alphas[t] * oh
y_pred = np.argmax(H, axis=1)

# visualize training data and the sampling weights
def plot_train(x, y, w):
    plt.figure(figsize=(5,5))
    color = [['red', 'blue', 'green'][a] for a in y]
    plt.scatter(x[:, 0], x[:, 1], s=w*10000, c=color, alpha=0.5)
    plt.xlim(-0.5, 1.0)
    plt.ylim(-0.5, 1.0)
    plt.show()

# visualize decision boundary
def plot_boundary(x, y, x_test, y_pred):
    plt.figure(figsize=(5,5))
    color = [['red', 'blue', 'green'][a] for a in y_pred]
    plt.scatter(x_test[:, 0], x_test[:, 1], s=100, c=color,
                alpha=0.3)
    plt.scatter(x[:, 0], x[:, 1], s=80, c='black')
    plt.scatter(x[:, 0], x[:, 1], s=10, c='yellow')
    plt.xlim(-0.5, 1.0)
    plt.ylim(-0.5, 1.0)
    plt.show()
```

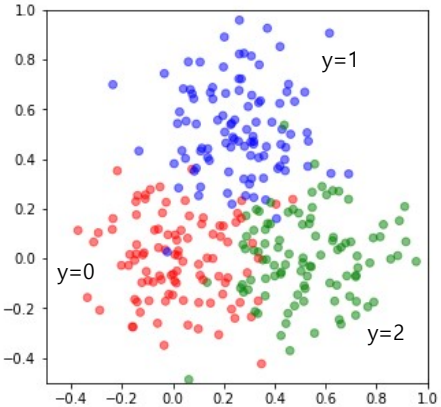
$$C(x) = \underset{k}{argmax} \sum_{m=1}^{M} \alpha^{(m)} \cdot I\left(T^{(m)}(x) = k\right)$$

```
plot_train(x, y, w=np.array(np.ones(shape=(m,)) / m))
plot_train(x, y, w=weights[-1])
plot_boundary(x, y, x_test, y_pred)

# Check the changes in α (alpha), ε (eps).
# Check that ε are all less than 1 - 1/K
# and that α and ε are inversely proportional.
plt.plot(eps, marker='o', markersize=4, c='red', lw=1,
         label='epsilon')
plt.legend()
plt.show()

plt.plot(alphas, marker='o', markersize=4, c='blue', lw=1,
         label='alpha')
plt.legend()
plt.show()
```

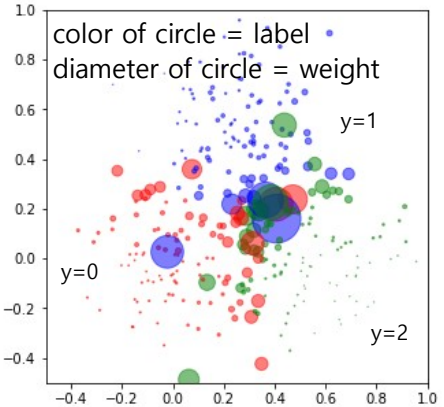
MX-AI

- Implementation of Multiclass AdaBoost using DecisionTreeClassifier, y = {0, 1, 2}

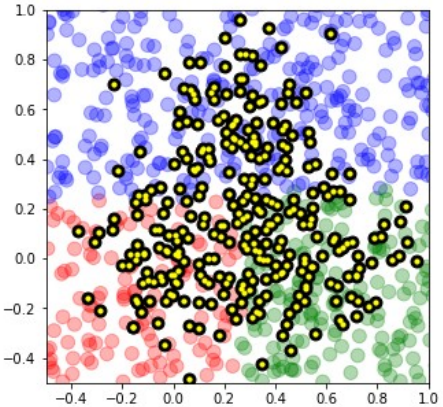
Training data and weights (1/m)

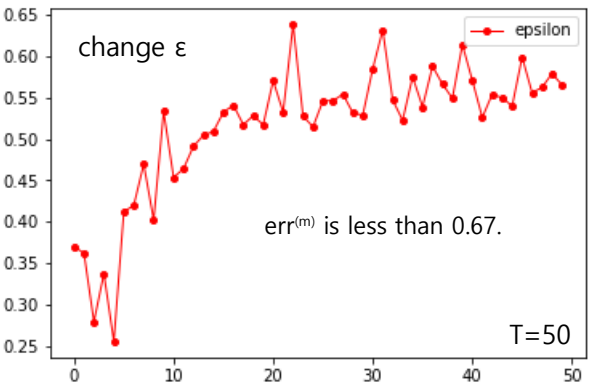


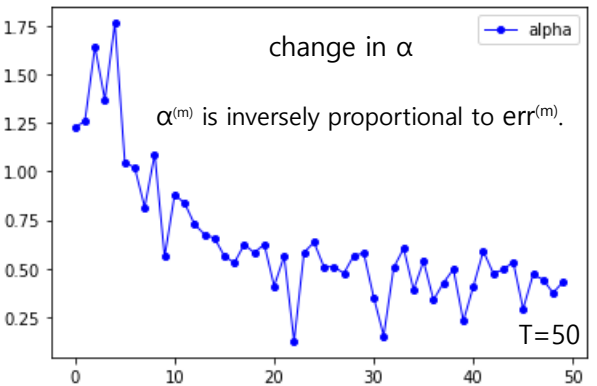
Training data and weights



decision boundary







MX-AI

▪ sklearn's AdaBoostClassifier

```
# [MXML-9-03] 4.sklearn(AdaBoost).py
# Test sklearn's AdaBoostClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier

# Read iris dataset
x, y = load_iris(return_X_y=True)

# Create training and test data
x_train, x_test, y_train, y_test = train_test_split(x, y)

# Use the decision tree as the base weak learner.
dt = DecisionTreeClassifier(max_depth = 1)

# Generate a AdaBoost model with the SAMME algorithm.
model = AdaBoostClassifier(estimator = dt,
                           n_estimators = 100,
                           algorithm = 'SAMME')  # default = 'SAMME.R'

# Fit the mode to the training data
model.fit(x_train, y_train)

# Predict the class of test data and calculate the accuracy
y_pred = model.predict(x_test)
accuracy = (y_pred == y_test).mean()

print('Accuracy = {:.4f}'.format(accuracy))
```

```
Results:

Accuracy = 0.9333

Accuracy = 0.9667   * Run multiple times
Accuracy = 1.0000
Accuracy = 0.9333
Accuracy = 0.9000

Run 3.AdaBoost(multiclass).py multiple times

Accuracy = 0.9333
Accuracy = 0.9667
Accuracy = 0.9000
Accuracy = 0.9667
Accuracy = 1.0000
```

MX-AI

▪ AdaBoost Regression - Overview

▪ In 1997, Harris Drucker proposed a regression algorithm by modifying Freund's (1995) AdaBoost.

Improving Regressors using Boosting Techniques

Harris Drucker
Monmouth University
West Long Branch, NJ 07764
drucker@monmouth.edu

Abstract

In the regression context, boosting and bagging are techniques to build a committee of regressors that may be superior to a single regressor. We use regression trees as fundamental building blocks in bagging committee machines and boosting committee machines. Performance is analyzed on three non-linear functions and the Boston housing database. In all cases, boosting is at least equivalent, and in most cases better than bagging in terms of prediction error.

1. INTRODUCTION

Both bagging [Breiman (1996a,1996b)] and boosting [Drucker et. al. (1994, 1996, 1993), Freund and Schapire (1996a,1996b), Schapire (1990)] are techniques to obtain smaller prediction errors (in regression) and lower error rates (in classification) using multiple predictors. Several studies of boosting and bagging in classification [Breiman (1996b), Freund and Schapire (1996a)] have shown the superiority of boosting over bagging but this is the first experimental study of combining regressors using boosting techniques. In both boosting and bagging, each regressor machine is trained on different subsets of the training set. In bagging, each machine is independently trained on N_1 samples picked with replacement from the N_1 original samples of the training set. Each machine is thereby trained on different (but overlapping) subsets of the original training set and will therefore give different predictions. Since each machine can be trained independently, the task of training each individual predictor may be assigned to different CPU's or a parallel processor. In bagging regressors, the ensemble prediction is the average of the predictions of all the machines.

(1)
In boosting, machines are trained sequentially. As in bagging, the first machine is trained on examples picked with replacement (of size N_1) from the original training set. We then pass all the training patterns through this first machine (2) and note which (3) ones are most in error. For regression machines, those patterns whose predicted values differ most from their observed values are defined to be "most" in error (this will be defined rigorously later). For those patterns most in (4)
error, their sampling probabilities are adjusted so that they are more likely to be picked as members of the training set for the second machine. Therefore, as we proceed in constructing machines, patterns that are difficult are more likely to appear in the training sets. Thus, different machines are better in different parts of the observation space. Regressors are combined using the weighted median, (5) whereby those predictors that are more "confident" about their predictions are weighted more heavily. The details of this weighting scheme are discussed later.

MX-AI

▪ AdaBoost Regression - Algorithm

3. BOOSTING: In bagging, each training example is equally likely to be picked. In boosting, the probability of a particular example being in the training set of a particular machine depends on the performance of the prior machines on that example. The following is a modification of Adaboost.R [Freund and Schapire (1996a)].

1. Initially, to each training pattern we assign a weight $w_i=1/N_1$, i =1, ...,N_1
   Pick N_1 samples (with replacement) to form the training set

Repeat the following while the average loss L defined below is less than 0.5 .

2. Construct a regression machine t from that training set. Each machine makes a hypothesis: h_t: x→y
3. Pass every member of the training set through this machine to obtain a prediction $y_i^{(p)}(x_i)$, i =1, ...N_1.
4. Calculate a loss for each training sample $L_i=L[|y_i^{(p)}-y_i|]$. The loss L may be of any functional form as long as L ∈ [0, 1]. If we let

$$D=\sup|y_i^{(p)}(x_i)-y_i| \quad i=1,....N_1$$

then we have three candidate loss functions:

$$L_i = \frac{|y_i^{(p)}-y_i|}{D} \quad \text{(linear)}$$

$$L_i = \frac{|y_i^{(p)}-y_i|^2}{D^2} \quad \text{(square law)}$$

$$L_i = 1-\exp\left(\frac{-|y_i^{(p)}-y_i|}{D}\right) \quad \text{(exponential)}$$

* classification

$$err^{(m)}=\sum_{i=1}^{n} w_i \cdot I\left(c_i \neq T^{(m)}(x_i)\right) \;/\; \sum_{i=1}^{n} w_i$$

$$\alpha^{(m)}=\log\frac{1-err^{(m)}}{err^{(m)}}$$

$$w_i \leftarrow w_i \cdot \exp\left(\alpha^{(m)} \cdot I\left(c_i \neq T^{(m)}(x_i)\right)\right)$$

5. Calculate an average loss: $\bar{L} = \sum_{i=1}^{N_1} L_i w_i$
6. Form $\beta = \frac{\bar{L}}{1-\bar{L}}$ ← Proportional to average loss

   β is a measure of confidence in the predictor. Low β means high confidence in the prediction.

7. Update the weights: $w_i \rightarrow w_i \beta$ ** $(1-L_i)$, where ** indicates exponentiation. The smaller the loss, the more the weight is reduced making the probability smaller that this pattern will be picked as a member of the training set for the next machine in the ensemble.

8. For a particular input x_i, each of the T machines makes a prediction h_t, t=1, ...,T. Obtain the cumulative prediction h_f using the T predictors:

$$h_f = \inf\{y \in Y : \sum_{t:h_t \le y} \log\left(\frac{1}{\beta_t}\right) \ge \frac{1}{2}\sum_t \log\left(\frac{1}{\beta_t}\right)\}$$

This is the weighted median.

Weight of each machine (α role in classification). It is inversely proportional to the average loss. The weight of machines with large losses is reduced.

MX-AI

▪ AdaBoost Regression - Algorithm

- In algorithm step 8, the weighted median is used for prediction.
- Here we will try both weighted average and weighted median.

$$w_t = \log(\frac{1}{\beta_t}) \,/\, \sum_t \log(\frac{1}{\beta_t}) \quad \leftarrow \text{Weight for the value estimated by each model}$$

1) Using weighted average

$$h_f = \sum_t \hat{y}_t w_t \quad \leftarrow \text{weighted average}$$

```
# Method-1: Using weighted average
wavg_pred = np.sum(y_pred * w, axis=1)
```

2) Using weighted median as in the paper

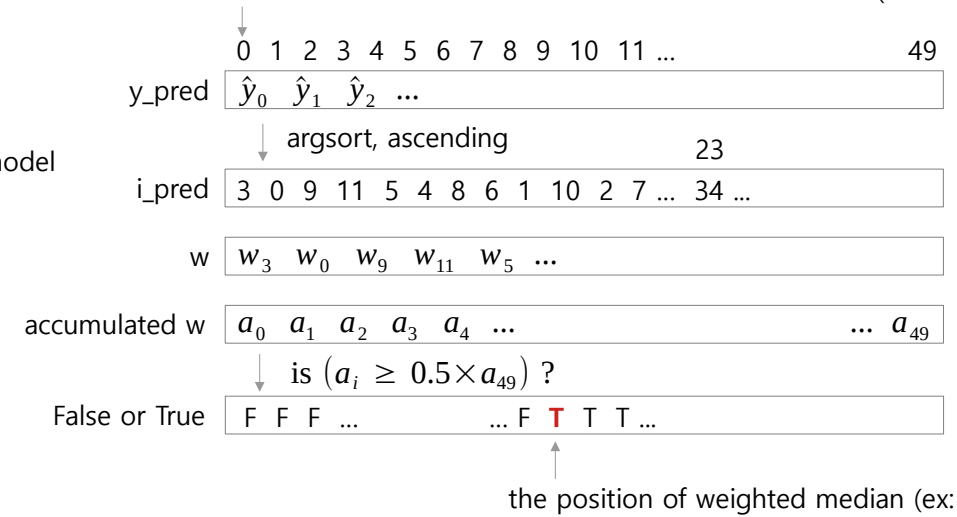
$$h_f = \inf\{y \in Y : \sum_{t:h_t \le y} \log(\frac{1}{\beta_t}) \ge \frac{1}{2}\sum_t \log(\frac{1}{\beta_t})\}$$

$$h_f = \inf\{y \in Y : \sum_{t:h_t \le y} w_t \ge \frac{1}{2}\sum_t w_t\}$$

If y is sorted in ascending order, the y at the position where the sum of the lower w is half of the total sum of w becomes the final output.

The y value estimated by the first model.

the number of models (ex: T=50)



the position of weighted median (ex: 23)

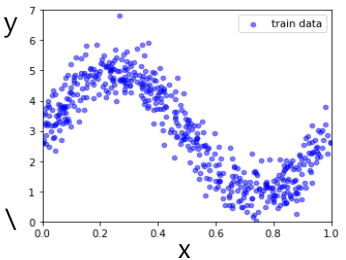
weighted median of y_pred = $\hat{y}_{34}$ ← final estimate

```
# weighted median: (sum of the lower w ≥ half of the total sum of w)
i_pred = np.argsort(y_pred, axis=1)
w_acc = np.cumsum(w[i_pred], axis=1)      # accumulated w
is_med = w_acc >= 0.5 * w_acc[:, -1][:, np.newaxis]
i_med = is_med.argmax(axis=1)             # 23
y_med = i_pred[np.arange(n_test), i_med]  # 34
wmed_pred = np.array(y_pred[np.arange(n_test), y_med]) # final estimate
```

MX-AI

- Implementation of AdaBoost regression using DecisionTreeRegressor & AdaBoostRegressor

```
# [MXML-9-04] 5.AdaBoost(regression).py
# [1] Harris Drucker et, al., 1997, Improving Regressors using
# Boosting Techniques
import numpy as np
import random as rd
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt

# Create training data
def noisy_sine_data(n, s):
    rtn_x, rtn_y = [], []
    for i in range(n):
        x = np.random.random()
        y = 2.0 * np.sin(2.0 * np.pi * x) + \
            np.random.normal(0.0, s) + 3.0
        rtn_x.append(x)
        rtn_y.append(y)
    return np.array(rtn_x).reshape(-1,1), np.array(rtn_y)
x, y = noisy_sine_data(n=500, s=0.5)

N = x.shape[0]
R = np.arange(N)
T = 100

weights = np.array(np.ones(shape=(N,)) / N)
beta = []
models = []  # save base learners for prediction
for t in range(T):
    s_idx = np.array(rd.choices(R, weights=weights, k=N))
    sx = x[s_idx]       # sample x
    sy = y[s_idx]       # sample y
```




```
    # base learner
    model = DecisionTreeRegressor(max_depth=5)
    model.fit(sx, sy)               # Fit the model to sample data

    # Calculate square loss
    y_pred = model.predict(x)       # predict entire training data
    err = np.abs(y - y_pred)
    loss = (err / err.max()) ** 2       # squared loss

    loss_avg = np.sum(weights * loss)  # average loss
    if loss_avg > 0.5:
        print('stopped at t={}, loss_avg={:.2f}'.\
              format(t, loss_avg))
        break

    # Calculate beta using average loss.
    beta.append(loss_avg / (1. - loss_avg))

    # Update weights using beta.
    new_weights = weights * np.power(beta[-1], (1. - loss))
    weights = new_weights / new_weights.sum()

    # save model
    models.append(model)

# Visualize training data and estimated curve
def plot_prediction(x, y, x_test, y_pred, title=""):
    plt.figure(figsize=(5,4))
    plt.scatter(x, y, c='blue', s=20, alpha=0.5, label='train')
    plt.plot(x_test, y_pred, c='red', lw=2.0, label='prediction')
    plt.xlim(0, 1)
```

$$L_i = \frac{|y_i^{(p)} - y_i|^2}{D^2}$$

$$\bar{L} = \sum_{i=1}^{N_1} L_i w_i$$

$$\beta = \frac{\bar{L}}{1-\bar{L}}$$

$$w_i \leftarrow w_i \beta^{1-L_i}$$

MX-AI

- Implementation of AdaBoost regression using DecisionTreeRegressor & AdaBoostRegressor

```
    plt.ylim(0, 7)
    plt.legend()
    plt.title(title)
    plt.show()

# prediction.
n_test = 50
x_test = np.linspace(0, 1, n_test).reshape(-1, 1)  # test data
log_beta = np.log(1. / np.array(beta))             # log(1/beta)
y_pred = np.array([m.predict(x_test) for m in models]).T

# Method-1: Using weighted average
w = log_beta/ log_beta.sum()    # normalize
wavg_pred = np.sum(y_pred * w, axis=1)
plot_prediction(x, y, x_test, wavg_pred, 'weighted average')

# Method-2: Using weighted median as in the paper
```

```
# weighted median: (sum of the lower w ≥ half of the total sum of w)
i_pred = np.argsort(y_pred, axis=1)
w_acc = np.cumsum(w[i_pred], axis=1)      # accumulated w
is_med = w_acc >= 0.5 * w_acc[:, -1][:, np.newaxis]
i_med = is_med.argmax(axis=1)
y_med = i_pred[np.arange(n_test), i_med]
wmed_pred = np.array(y_pred[np.arange(n_test), y_med])
plot_prediction(x, y, x_test, wmed_pred, 'weighted median')

# Let's compare the results with sklearn's AdaBoostRegressor
from sklearn.ensemble import AdaBoostRegressor
dt = DecisionTreeRegressor(max_depth=5)
model = AdaBoostRegressor(estimator=dt, n_estimators=T, loss='square')
model.fit(x, y)
sk_pred = model.predict(x_test)
plot_prediction(x, y, x_test, sk_pred, 'AdaBoostRegressor')
```