

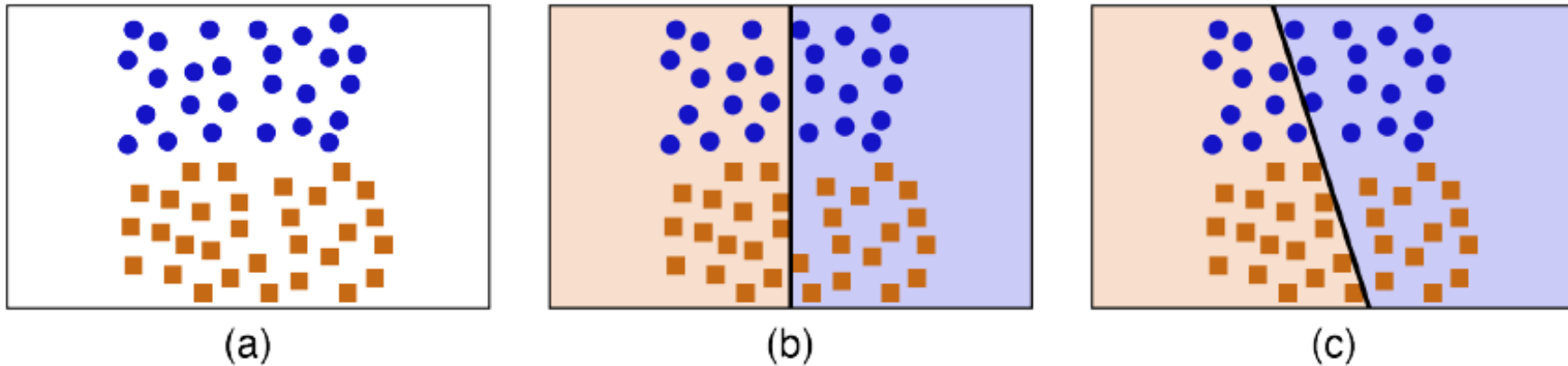
# **Adaptive Boosting (AdaBoost) and Gradient Boosting Algorithms**

國立陽明交通大學  
生醫光電所 吳育德

# Boosting

- Boosting lets us combine a large number of small, fast, and inaccurate learners into a single accurate learner.
- Let's suppose our data contains samples of just two categories.
- What if a binary classifier does just a tiny bit better than chance?

## Some bad binary classifiers.



(a) Our training data. The best split would be a horizontal line across the middle of the box.

(b) A terrible binary classifier. This is no better than chance.

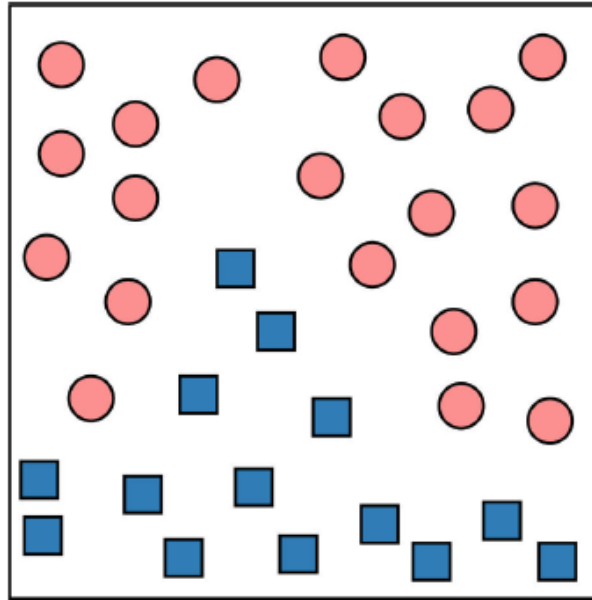
(c) A binary classifier that is only a little bit better than chance. It is a terrible classifier, but thanks to the small tilt in the boundary line, it is slightly less terrible than the classifier in part (b).

- We call the classifier in (c) a **weak learner**. A weak learner in this situation is any classifier that is even the slightest bit accurate.

# Boosting

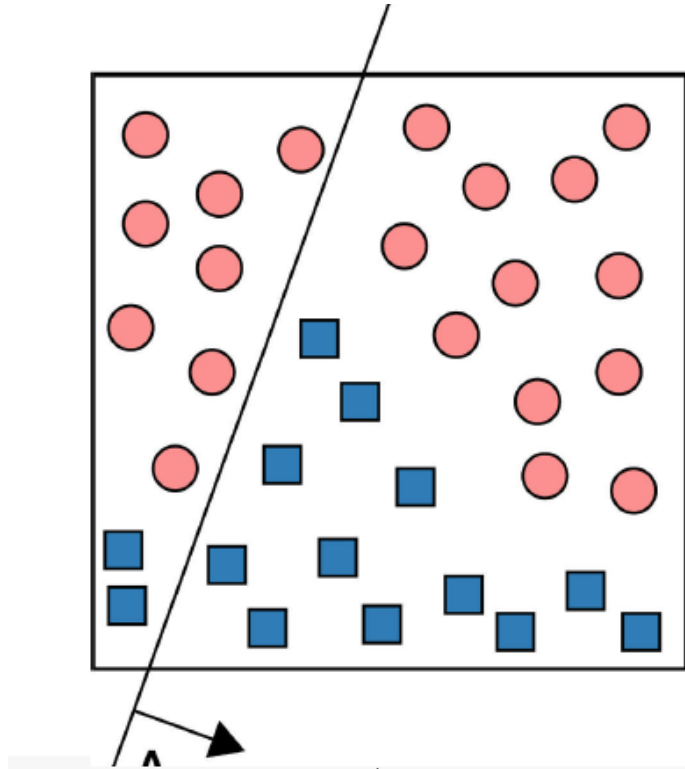
- Weak classifiers are easy to find. The most commonly-used weak classifier is a tree made up of just a root node and its two children.
- This tree is called a **decision stump**. It's small, fast, and a little better than random.
- The idea behind boosting is to combine multiple weak classifiers into an ensemble that acts like a strong classifier.
- Note we can combine lots of strong classifiers if we want to, though weak ones are common because they're usually faster.

## A collection of samples we're going to classify using boosting



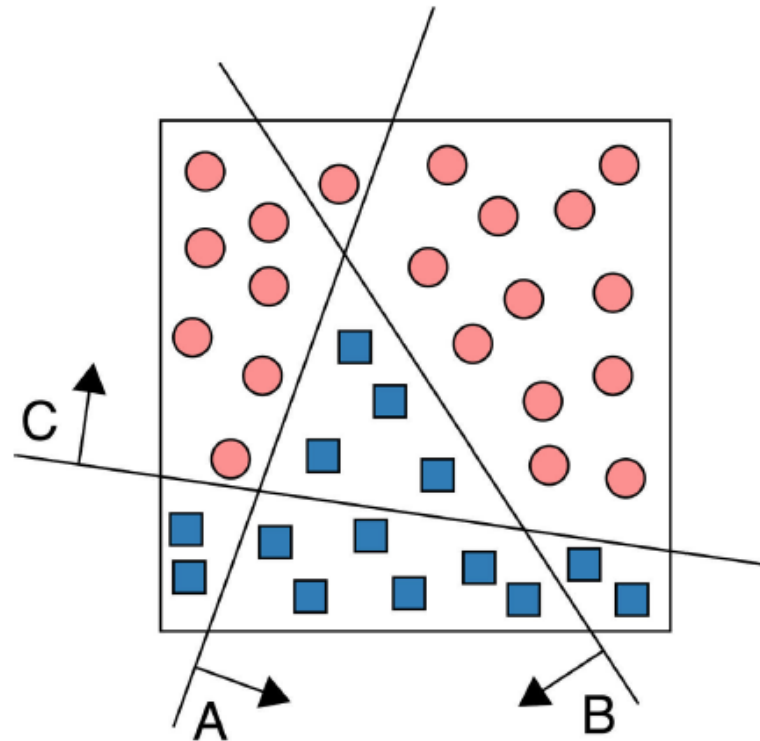
- Even though no single straight line can separate this data, multiple straight lines can, so let's use straight lines as our weak classifiers.
- Remember, these classifiers don't have to do a great job on the whole dataset. They only have to do a bit better than chance.

## Placing one line called A to cut the two big clusters apart



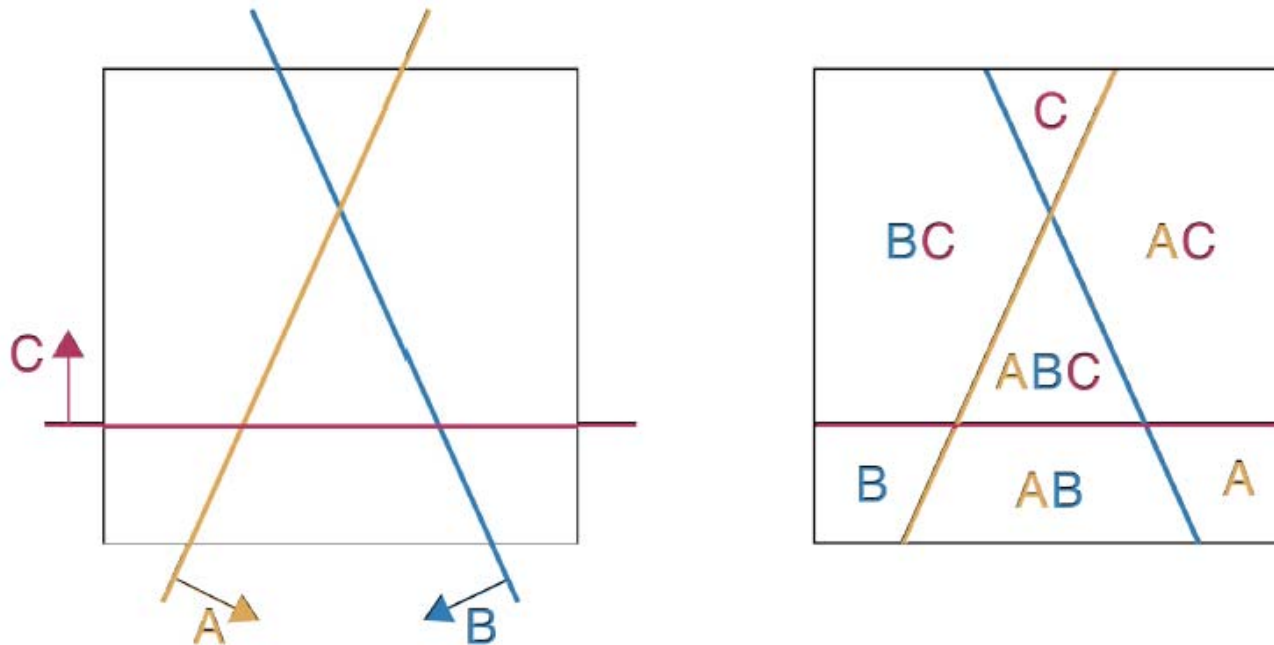
- The arrow points to the “positive” side of the line, which we’re associating with square samples. Unfortunately, there are some misclassified data points on both sides of the line.

## Two more lines added to the Figure



- To use boosting, we'll want to add more lines (that is, additional weak learners) so that **every region formed by the lines contains samples of a single color**.
- Now each region formed by these three lines is made up of samples of **only one class**.

## Combine A, B, C to do a far better job



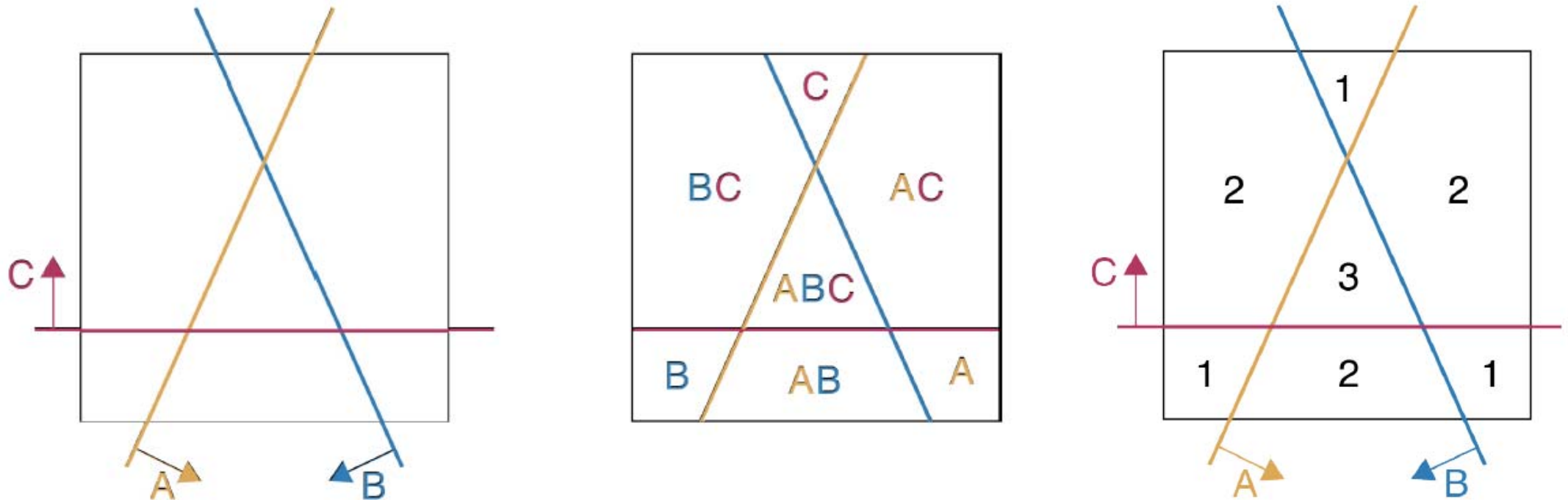
- On the left, we show three lines named A, B, and C.
- On the right, each region is marked with the names of the learners that put that region **on the positive side** of their respective lines.



## **Combine A, B, C to do a far better job**

- Rather than have each learner report a category (round or square) we'll set it up to report a value associated with that learner.
- So a sample is on the positive side of a learner's line, it reports its value, otherwise it reports 0.
- What values should we use? To demonstrate how the process works, we'll assign +1 to all three learners.
- We'll see later that these values will be determined automatically for us by the algorithm.
- So if a sample is on the positive side of a learner's line (that is, the side pointed to by the arrow), then that learner will return a value of +1. Otherwise, it returns 0.

## The composite score for each of the seven regions

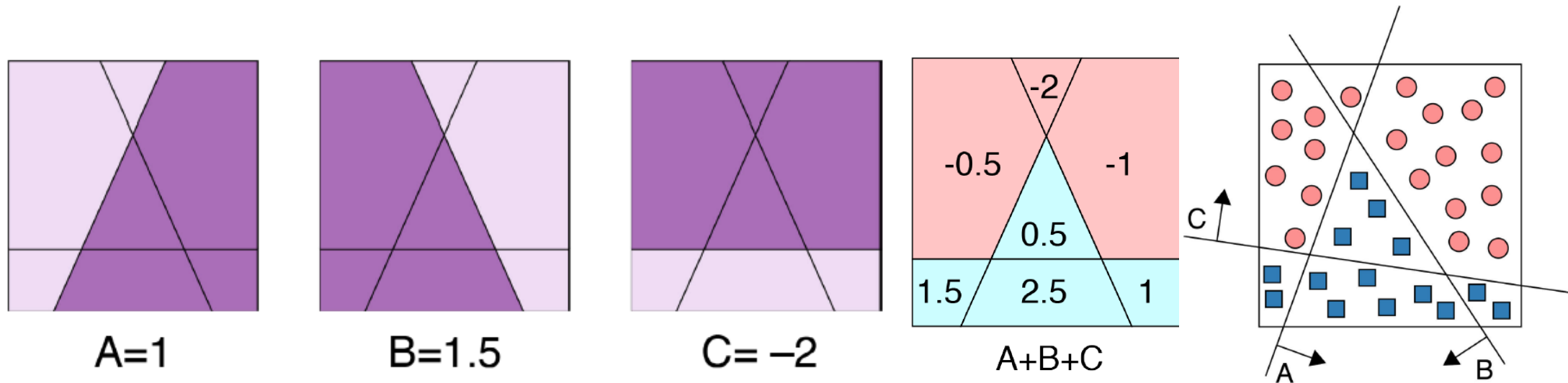


- Each letter of A, B, C earns that region 1.

## **Combine A, B, C to do a far better job**

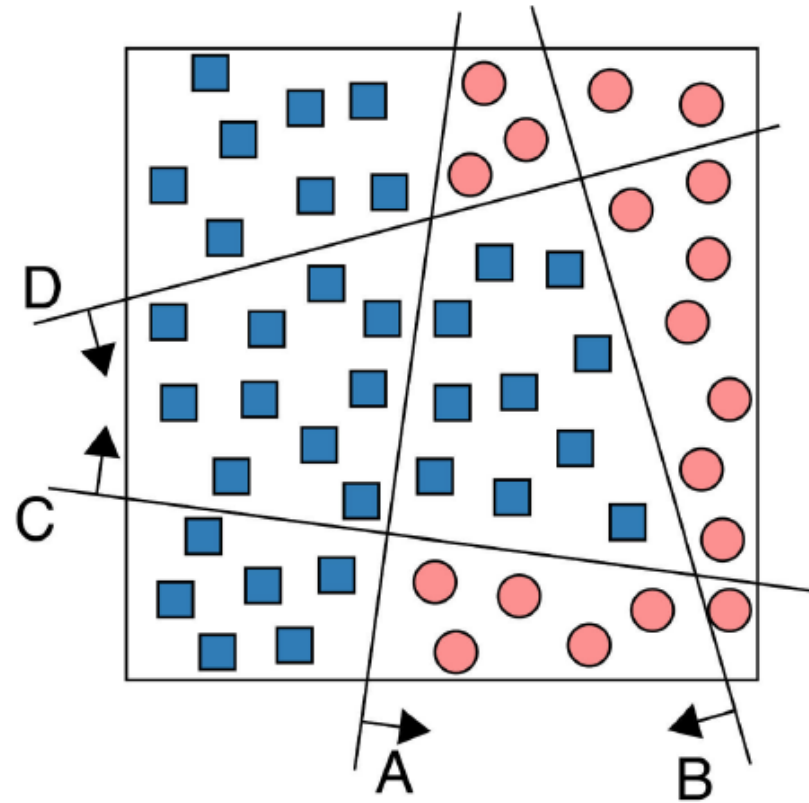
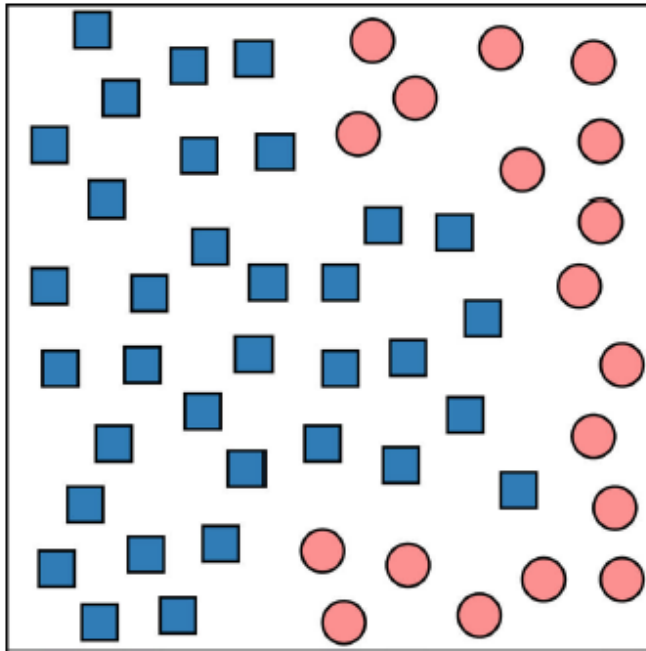
- To turn this information into a working classifier, we need to find the right weights for each region, and the right threshold for assigning categories.
- In following Figure we show the regions that are affected by the score for each learner.
- A dark region gets that learner's value, while a light region does not (so the learner's value in light regions is 0).
- Here we'll use the weights  $(1.0, 1.5, -2)$  for A, B, and C respectively.

**We assign a numerical value to each region that is classified as positive by each learner**



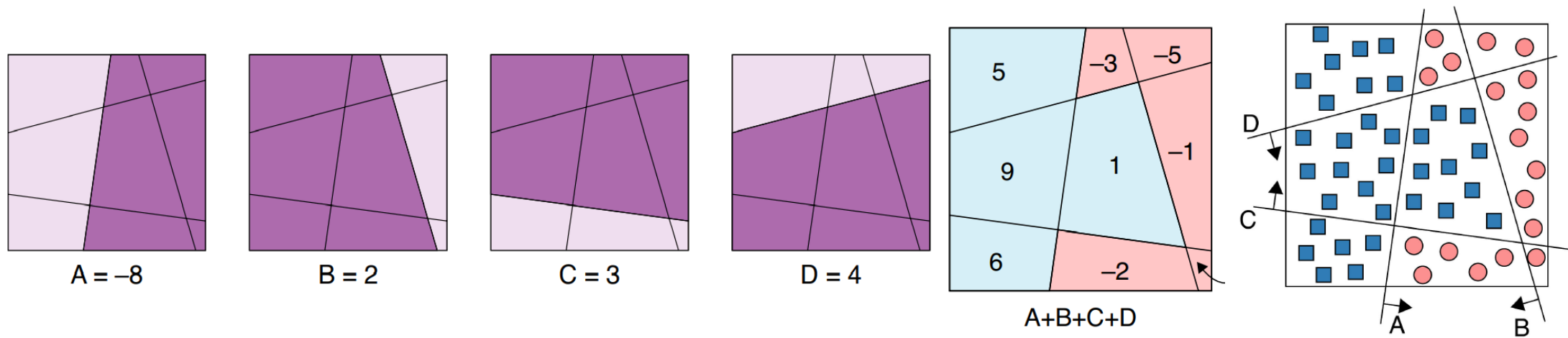
- The dark regions for each line get the weight associated with that line.
- The sums of all of these scores are shown in right Figure. The blue cells are the only ones with positive values. These are the ones that we want. We've correctly classified our data!

## Another set of data to classify using boosting



- Right: Four lines that let us classify the data.
- As before, we'll manually assign weights to these learners.
- This time we'll use  $-8$ ,  $2$ ,  $3$ , and  $4$  for learners A, B, C, and D respectively.

# The regions corresponding to each learner



- Right: Positive regions are shown in blue, and they correctly classify the points.

# Boosting

- The beauty of boosting is that it combines classifiers that are simple and fast, but lousy, into a single great classifier.
- It even figures out which simple classifiers to use, and what weights they should have when voting.
- The only thing we have to pick is how many classifiers we want. In boosting, a rule of thumb is to start with about as many classifiers as there are categories available for them to determine.

## Pseudocode of the AdaBoost algorithm

- **for**  $i$  from 1 to  $N$ ,  $w_i^{(1)} = \frac{1}{N}$ ,  $N$  is the total number of training data.
- **for**  $m = 1$  to  $M$  **do**  
Fit weak classifier  $f_m(\mathbf{x}_n) = \{y_n = 1 \text{ or } -1\}$  to the training data and minimize the objective function:

$$\epsilon_m = \frac{\sum_{i=1}^N w_i^{(m)} 1(f_m(\mathbf{x}_i) \neq y_i)}{\sum_{i=1}^N w_i^{(m)}}, \text{ where } 1(f_m(\mathbf{x}_i) \neq y_i) = \begin{cases} 1, & \text{if } f_m(\mathbf{x}_i) \neq y_i \\ 0, & \text{if } f_m(\mathbf{x}_i) = y_i \end{cases}$$

$$\alpha_m = \frac{1}{2} \log \left\{ \frac{1 - \epsilon_m}{\epsilon_m} \right\}$$

**for** all  $i = 1$  **to**  $N$  **do**

$$w_i^{(m+1)} = w_i^{(m)} e^{\alpha_m 1(f_m(\mathbf{x}_i) \neq y_i)}.$$

**end for**

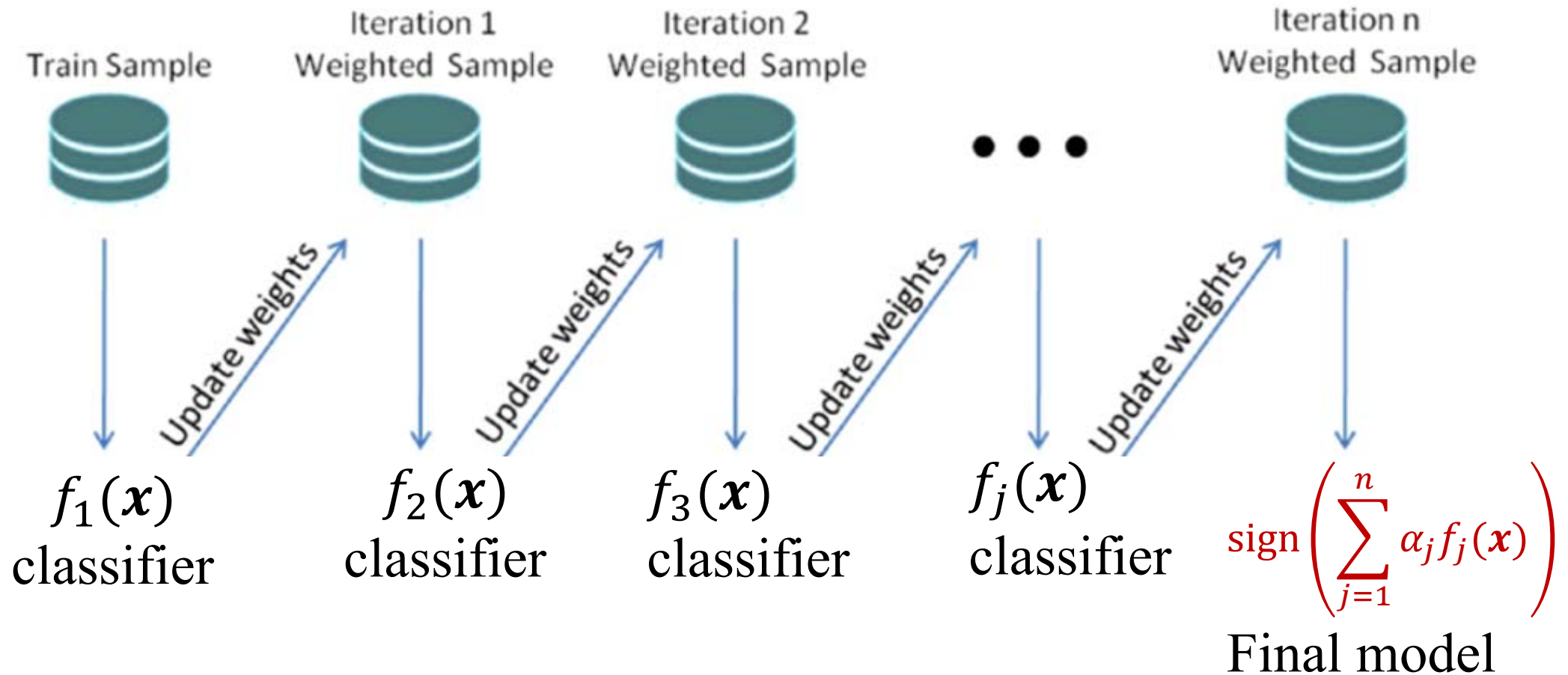
提高被分錯類的樣本之權重使之在下輪訓練中得到更多的關注

**end for**

- The final model is  $g(\mathbf{x}) = \text{sign}(\sum_{m=1}^M \alpha_m f_m(\mathbf{x}))$



# AdaBoosting process



## Why does it work? The loss function

- Define  $f(\mathbf{x}) = \frac{1}{2} \sum_{m=1}^M \alpha_m f_m(\mathbf{x})$ ,  $g(x) = \text{sign}(f(x))$
- AdaBoost can be viewed as minimizing the **exponential loss**:

$$L_{exp}(\mathbf{x}, y) = e^{-yf(\mathbf{x})}$$

- Given training data  $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ , the full learning objective function :

$$E = \sum_{i=1}^N e^{-\frac{1}{2} y_i \sum_{m=1}^M \alpha_m f_m(\mathbf{x}_i)}$$

- Minimize  $E$  w.r.t. the weights  $\alpha_m$
- The minimization process is **greedy** and **sequential**: we add one weak classifier at a time, choosing it and minimize  $E$  w.r.t  $\alpha_m$ , and then never change it again.

# Comparison of different loss functions for binary classification

$$D(\mathbf{x}_i) = \mathbf{w}^T \mathbf{x}_i - b$$

The loss function to be minimized is

$$\frac{1}{N} \sum_{i=1}^N L_{\text{fun}}(\mathbf{y}_i, D(\mathbf{x}_i))$$

$$L_{0|1}(\mathbf{y}_i, D(\mathbf{x}_i)) = \begin{cases} 1, & \text{if } \mathbf{y}_i D(\mathbf{x}_i) < 0 \\ 0, & \text{otherwise} \end{cases}$$

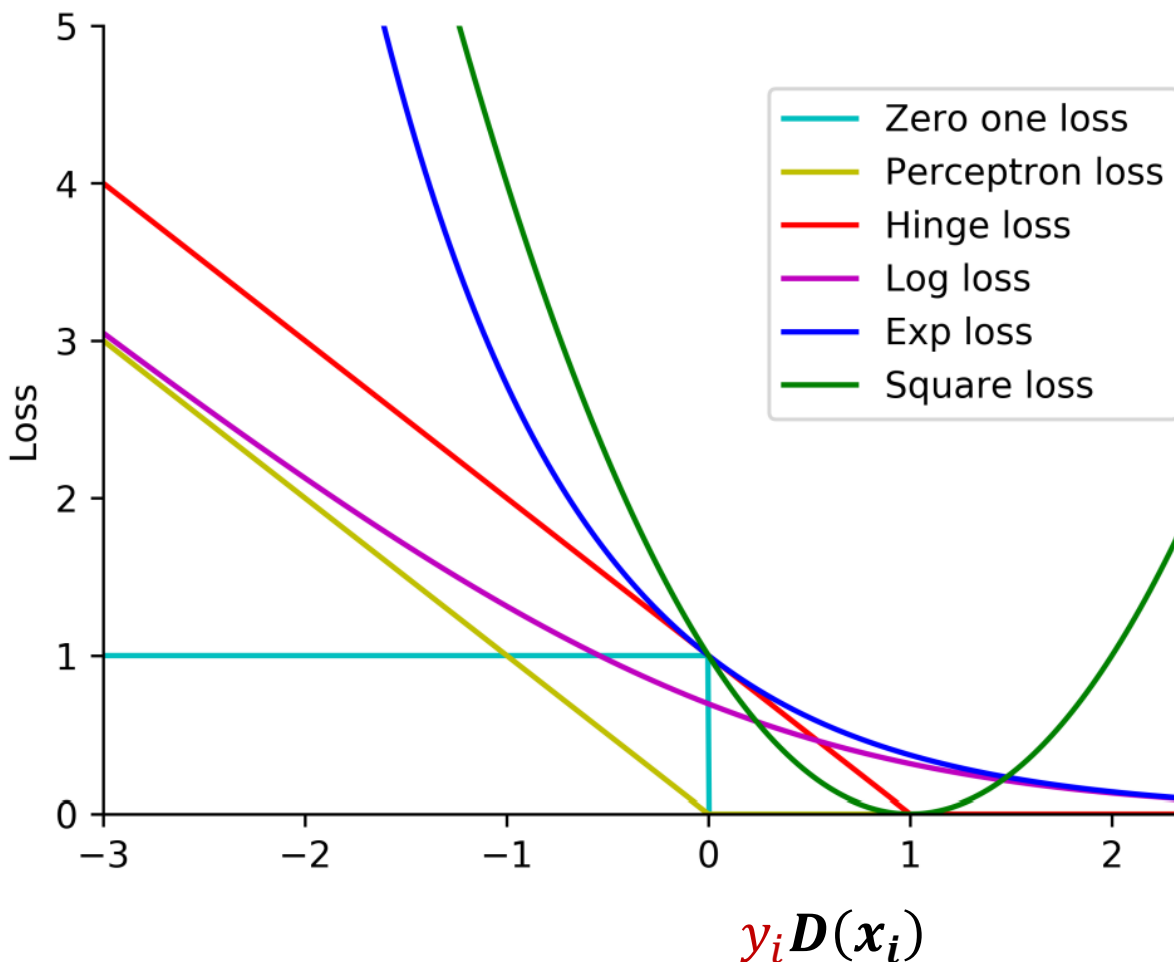
$$L_{\text{per}}(\mathbf{y}_i, D(\mathbf{x}_i)) = \max(0, -\mathbf{y}_i D(\mathbf{x}_i))$$

$$L_{\text{hin}}(\mathbf{y}_i, D(\mathbf{x}_i)) = \max(0, 1 - \mathbf{y}_i D(\mathbf{x}_i))$$

$$L_{\text{log}}(\mathbf{y}_i, D(\mathbf{x}_i)) = \log(1 + e^{-\mathbf{y}_i D(\mathbf{x}_i)})$$

$$L_{\text{exp}}(\mathbf{y}_i, D(\mathbf{x}_i)) = e^{-\mathbf{y}_i D(\mathbf{x}_i)}$$

$$L_{\text{squ}}(\mathbf{y}_i, D(\mathbf{x}_i)) = (1 - \mathbf{y}_i D(\mathbf{x}_i))^2$$



## Minimize $E$ with respect to $\alpha_m$

- Consider the weak classifier  $f_m$  to be added at step  $m$

$$E = \sum_{i=1}^N e^{-y_i \sum_{j=1}^m \alpha_j f_j(x_i)} = \sum_{i=1}^N e^{-y_i \sum_{j=1}^{m-1} \alpha_j f_j(x_i) - y_i \alpha_m f_m(x_i)}$$

- Due to **greedy** and **sequential** property, the first  $m-1$  can be considered to be constant and we can replace them with a single constant

$$w_i^{(m)} = e^{-y_i \sum_{j=1}^{m-1} \alpha_j f_j(x_i)}$$

- Therefore

$$E = \sum_{i=1}^N w_i^{(m)} e^{-y_i \alpha_m f_m(x_i)}$$

- Note weights can be computed recursively

$$w_i^{(m)} = e^{-y_i \sum_{j=1}^{m-2} \alpha_j f_j(x_i) - y_i \alpha_{m-1} f_{m-1}(x_i)} = w_i^{(m-1)} e^{-y_i \alpha_{m-1} f_{m-1}(x_i)}$$

## Minimize $E$ with respect to $\alpha_m$

- We can split this into two summations, one for data correctly classified by  $f_m$ , and one for those misclassified

$$\begin{aligned} E &= \sum_{i: f_m(\mathbf{x}_i) = y_i}^N w_i^{(m)} e^{-\alpha_m} + \sum_{i: f_m(\mathbf{x}_i) \neq y_i}^N w_i^{(m)} e^{\alpha_m}, 1(f_m(\mathbf{x}_i) \neq y_i) = \begin{cases} 1, & \text{if } f_m(\mathbf{x}_i) \neq y_i \\ 0, & \text{if } f_m(\mathbf{x}_i) = y_i \end{cases} \\ &= e^{-\alpha_m} \sum_{i=1}^N w_i^{(m)} \{1 - 1(f_m(\mathbf{x}_i) \neq y_i)\} + e^{\alpha_m} \sum_{i=1}^N w_i^{(m)} 1(f_m(\mathbf{x}_i) \neq y_i) \\ &= (e^{\alpha_m} - e^{-\alpha_m}) \sum_{i=1}^N w_i^{(m)} 1(f_m(\mathbf{x}_i) \neq y_i) + e^{-\alpha_m} \sum_{i=1}^N w_i^{(m)} \\ \Rightarrow \quad \frac{dE}{d\alpha_m} &= \alpha_m (e^{\alpha_m} + e^{-\alpha_m}) \sum_{i=1}^N w_i^{(m)} 1(f_m(\mathbf{x}_i) \neq y_i) - \alpha_m e^{-\alpha_m} \sum_{i=1}^N w_i^{(m)} = 0 \end{aligned}$$

**Minimize  $E$  with respect to  $\alpha_m$**

$$\begin{aligned}\Rightarrow \frac{dE}{d\alpha_m} &= \alpha_m (e^{\alpha_m} + e^{-\alpha_m}) \sum_{i=1}^N w_i^{(m)} 1(f_m(\mathbf{x}_i) \neq y_i) - \alpha_m e^{-\alpha_m} \sum_{i=1}^N w_i^{(m)} = 0 \\ \Rightarrow & (e^{\alpha_m} + e^{-\alpha_m}) \frac{\sum_{i=1}^N w_i^{(m)} 1(f_m(\mathbf{x}_i) \neq y_i)}{\sum_{i=1}^N w_i^{(m)}} - e^{-\alpha_m} = 0\end{aligned}$$

Define

$$\begin{aligned}\epsilon_m &\equiv \frac{\sum_{i=1}^N w_i^{(m)} 1(f_m(\mathbf{x}_i) \neq y_i)}{\sum_{i=1}^N w_i^{(m)}} \\ \Rightarrow & (e^{\alpha_m} + e^{-\alpha_m}) \epsilon_m - e^{-\alpha_m} = 0 \\ \Rightarrow & e^{\alpha_m} \epsilon_m = (1 - \epsilon_m) e^{-\alpha_m} \\ \Rightarrow & \alpha_m + \ln \epsilon_m = -\alpha_m + \ln(1 - \epsilon_m) \\ \Rightarrow & 2\alpha_m = \ln(1 - \epsilon_m) - \ln \epsilon_m = \ln \frac{1 - \epsilon_m}{\epsilon_m} \\ \Rightarrow & \alpha_m = \frac{1}{2} \ln \frac{1 - \epsilon_m}{\epsilon_m}\end{aligned}$$

## Pseudocode of the AdaBoost algorithm

- **for**  $i$  from 1 to  $N$ ,  $w_i^{(1)} = \frac{1}{N}$ ,  $N$  is the total number of training data.
- **for**  $m = 1$  to  $M$  **do**  
Fit weak classifier  $f_m(\mathbf{x}_n) = \{y_n = 1 \text{ or } -1\}$  to the training data and minimize the objective function:

$$\epsilon_m = \frac{\sum_{i=1}^N w_i^{(m)} 1(f_m(\mathbf{x}_i) \neq y_i)}{\sum_{i=1}^N w_i^{(m)}}, \text{ where } 1(f_m(\mathbf{x}_i) \neq y_i) = \begin{cases} 1, & \text{if } f_m(\mathbf{x}_i) \neq y_i \\ 0, & \text{if } f_m(\mathbf{x}_i) = y_i \end{cases}$$

$$\alpha_m = \frac{1}{2} \log \left\{ \frac{1 - \epsilon_m}{\epsilon_m} \right\}$$

**for** all  $i = 1$  **to**  $N$  **do**

$$w_i^{(m+1)} = w_i^{(m)} e^{\alpha_m 1(f_m(\mathbf{x}_i) \neq y_i)}.$$

**end for**

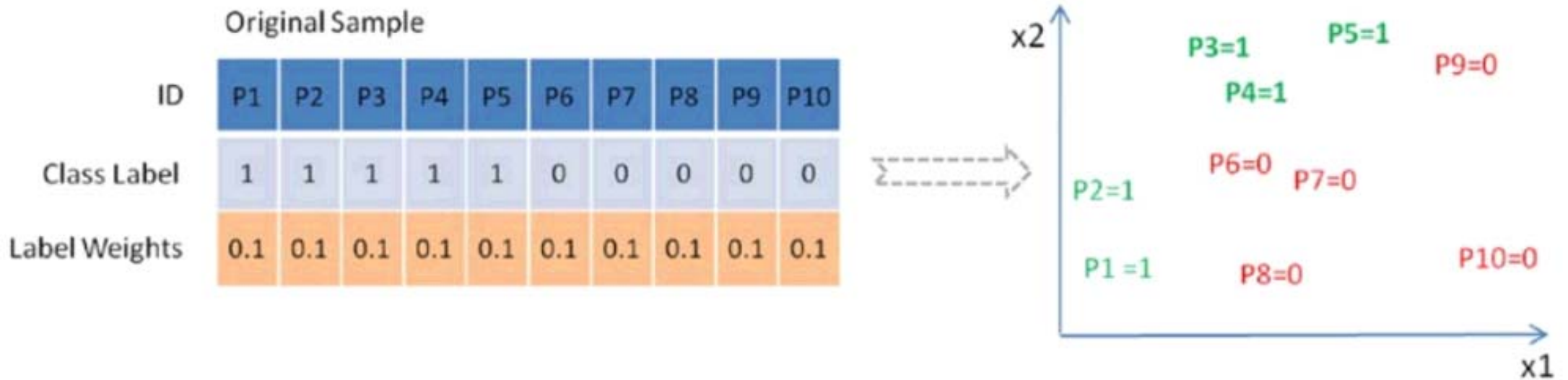
提高被分錯類的樣本之權重使之在下輪訓練中得到更多的關注

**end for**

- The final model is  $g(\mathbf{x}) = \text{sign}(\sum_{m=1}^M \alpha_m f_m(\mathbf{x}))$

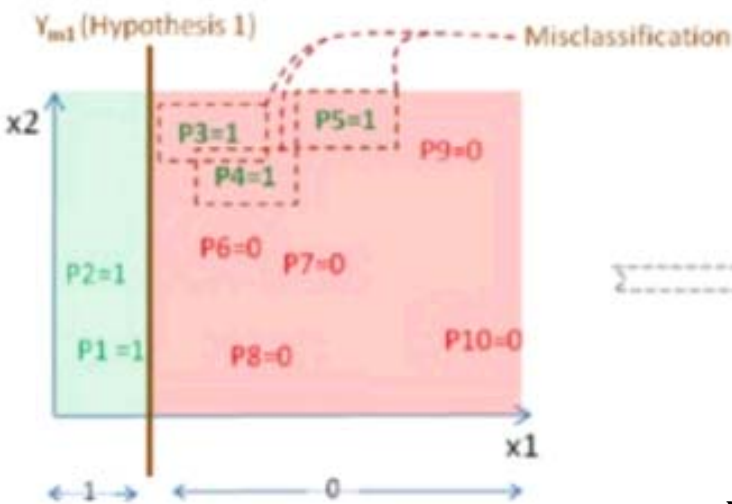
## Sample data set with ten data points

Let's consider training data with two class labels of ten data points. Initially, all the data points will have equal weights given by  $1/10$ .





# Boosting Iteration 1



$$\begin{aligned}\epsilon_{m1} &= \frac{(0.1 + 0.1 + 0.1)}{1} = 0.30 \\ \alpha_{m1} &= \frac{1}{2} \log \left( \frac{1 - 0.30}{0.30} \right) = 0.42 \\ W_{m1} &= 0.1 * e^{0.42} = 0.15\end{aligned}$$

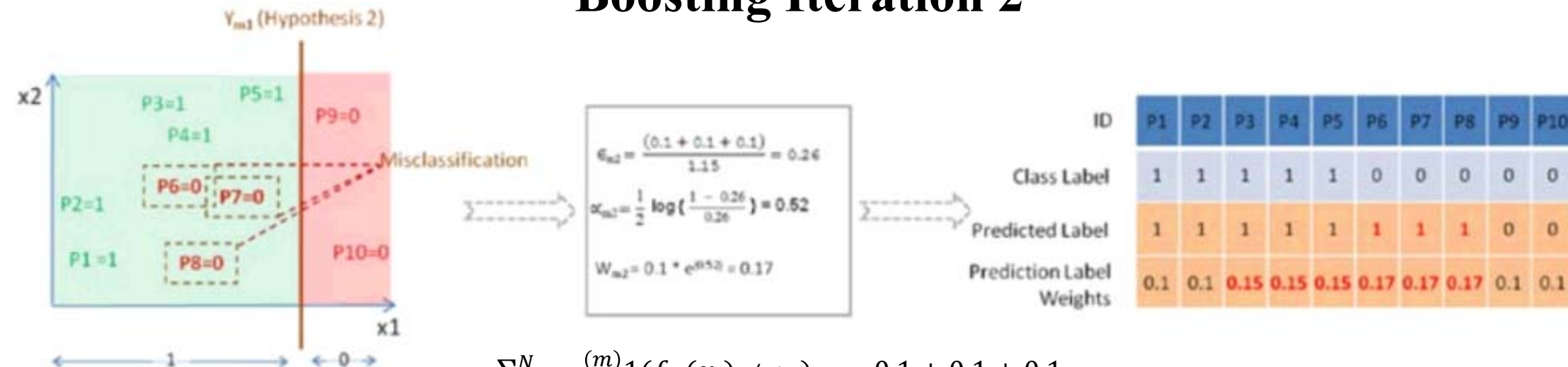
ID  
Class Label  
Predicted Label  
Prediction Label  
Weights

ID	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Class Label	1	1	1	1	1	0	0	0	0	0
Predicted Label	1	1	0	0	0	0	0	0	0	0
Prediction Label Weights	0.1	0.1	0.15	0.15	0.15	0.1	0.1	0.1	0.1	0.1

$$\begin{aligned}\epsilon_m &= \frac{\sum_{i=1}^N w_i^{(m)} 1(f_m(x_i) \neq y_i)}{\sum_{i=1}^N w_i^{(m)}} = \frac{0.1 + 0.1 + 0.1}{\sum_{i=1}^{10} 0.1} = 0.3 \\ \alpha_m &= \frac{1}{2} \log \left\{ \frac{1 - \epsilon_m}{\epsilon_m} \right\} = \frac{1}{2} \log \left\{ \frac{1 - 0.3}{0.3} \right\} = 0.42 \\ w_i^{(m+1)} &= w_i^{(m)} e^{e^{\alpha_m 1(f_m(x_i) \neq y_i)}} = 0.1 e^{0.42} = 0.15\end{aligned}$$

Three points of the positive class are misclassified by the first simple classification model, so they will be assigned higher weights. Error term and loss function (learning rate) are calculated as 0.30 and 0.42, respectively. The data points P3, P4, and P5 will get higher weight (0.15) due to misclassification, whereas other data points will retain the original weight (0.1). Manohar Swamynathan, Mastering Machine Learning with Python in Six Steps, 2ed, 2019

## Boosting Iteration 2



$$\epsilon_m = \frac{\sum_{i=1}^N w_i^{(m)} 1(f_m(x_i) \neq y_i)}{\sum_{i=1}^N w_i^{(m)}} = \frac{0.1 + 0.1 + 0.1}{0.1 \times 7 + 0.15 \times 3} = 0.26$$

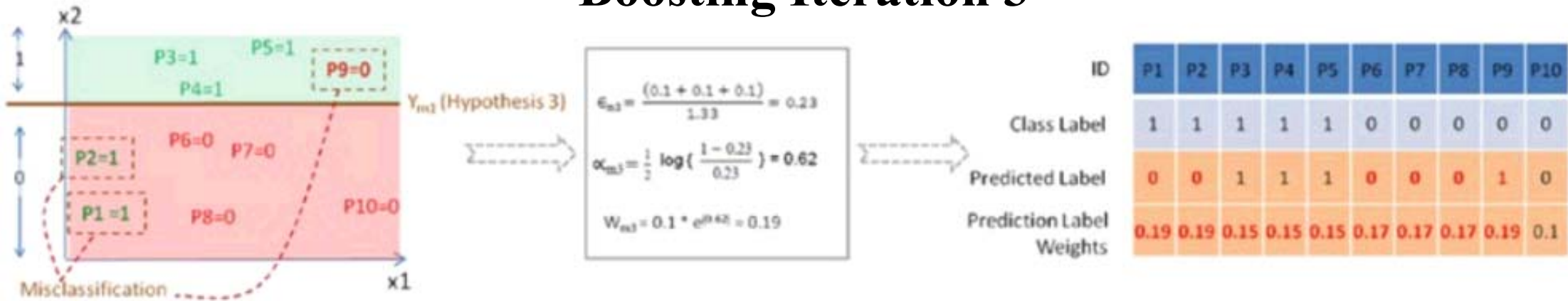
$$\alpha_m = \frac{1}{2} \log \left\{ \frac{1 - \epsilon_m}{\epsilon_m} \right\} = \frac{1}{2} \log \left\{ \frac{1 - 0.26}{0.26} \right\} = 0.52$$

$$w_i^{(m+1)} = w_i^{(m)} e^{e^{\alpha_m 1(f_m(x_i) \neq y_i)}} = 0.1 e^{0.52} = 0.17$$

Let's fit another classification model and notice that three data points (P6, P7, and P8) of the negative class are misclassified. Hence, these will be assigned higher weights of 0.17 as calculated, whereas the remaining data points' weights will remain the same because they are correctly classified.

Manohar Swamynathan, Mastering Machine Learning with Python in Six Steps, 2ed, 2019

## Boosting Iteration 3



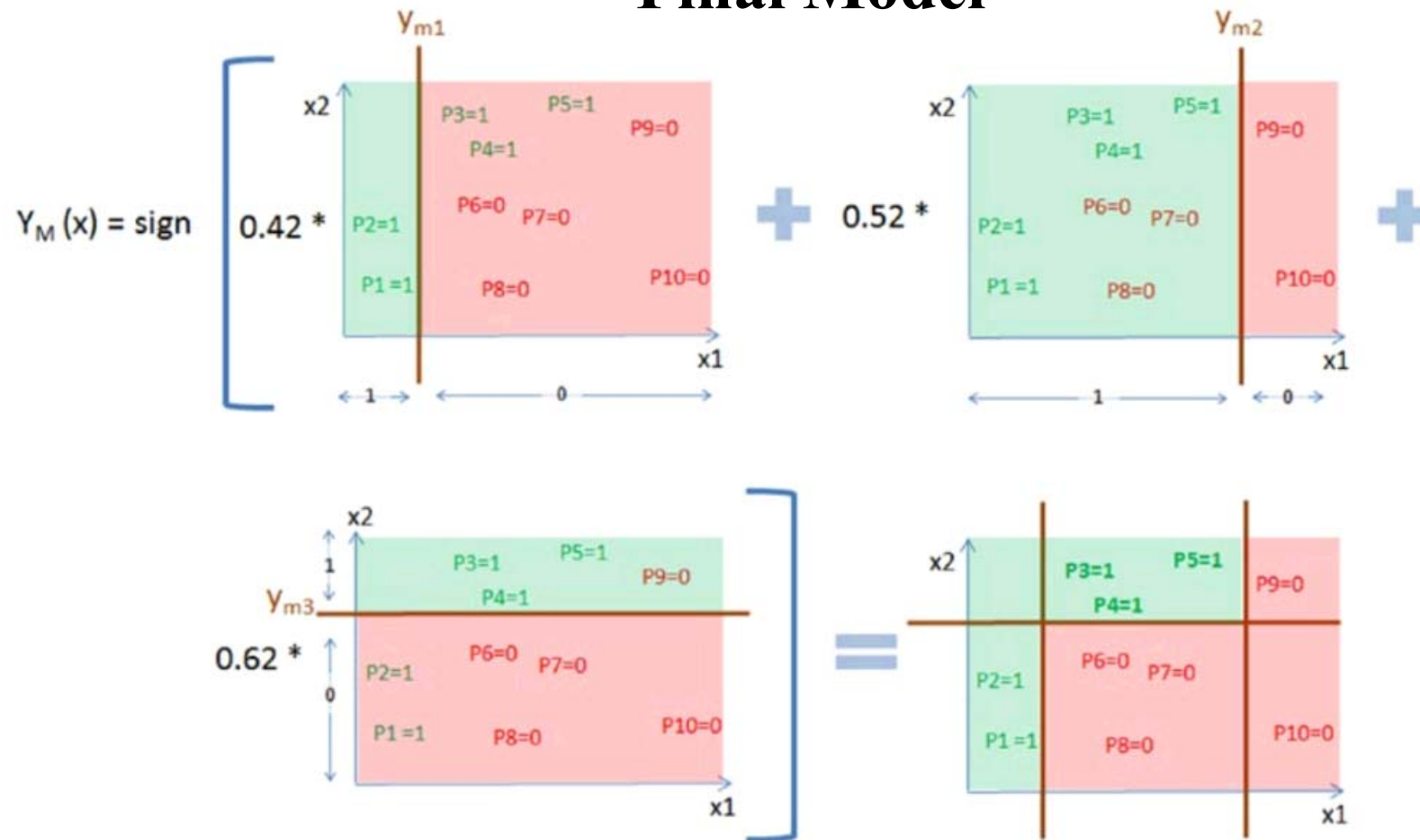
$$\epsilon_m = \frac{\sum_{i=1}^N w_i^{(m)} 1(f_m(x_i) \neq y_i)}{\sum_{i=1}^N w_i^{(m)}} = \frac{0.1 + 0.1 + 0.1}{0.1 \times 4 + 0.15 \times 3 + 0.17 \times 3} = 0.22$$

$$\alpha_m = \frac{1}{2} \log \left\{ \frac{1 - \epsilon_m}{\epsilon_m} \right\} = \frac{1}{2} \log \left\{ \frac{1 - 0.22}{0.22} \right\} = 0.62$$

$$w_i^{(m+1)} = w_i^{(m)} e^{e^{\alpha_m 1(f_m(x_i) \neq y_i)}} = 0.1 e^{0.62} = 0.19$$

The third classification model has misclassified a total of three data points: two positive classes, P1 and P2; and one negative class, P9. So these misclassified data points will be assigned a new higher weight of 0.19 as calculated, and the remaining data points will retain their earlier weights.

# Final Model



Let's combine the weak classification models. The final combined model will have a minimum error term and maximum learning rate, leading to a higher degree of accuracy.

## AdaBoost algorithm for Lecture 14\_1 adaboost\_oop.jpynb

- **for**  $i$  from 1 to  $N$ ,  $w_i^{(1)} = \frac{1}{N}$ ,  $N$  is the total number of training data.

- **for**  $m = 1$  to  $M$  **do**

Fit weak classifier  $f_m(\mathbf{x}_n) = \{y_n = 1 \text{ or } -1\}$  to the training data and minimize the objective function:

$$\epsilon_m = \frac{\sum_{i=1}^N w_i^{(m)} 1(f_m(\mathbf{x}_i) \neq y_i)}{\sum_{i=1}^N w_i^{(m)}}, \text{ where } 1(f_m(\mathbf{x}_i) \neq y_i) = \begin{cases} 1, & \text{if } f_m(\mathbf{x}_i) \neq y_i \\ 0, & \text{if } f_m(\mathbf{x}_i) = y_i \end{cases}$$

$$\alpha_m = \frac{1}{2} \log \left\{ \frac{1 - \epsilon_m}{\epsilon_m} \right\}$$

**for** all  $i = 1$  **to**  $N$  **do**

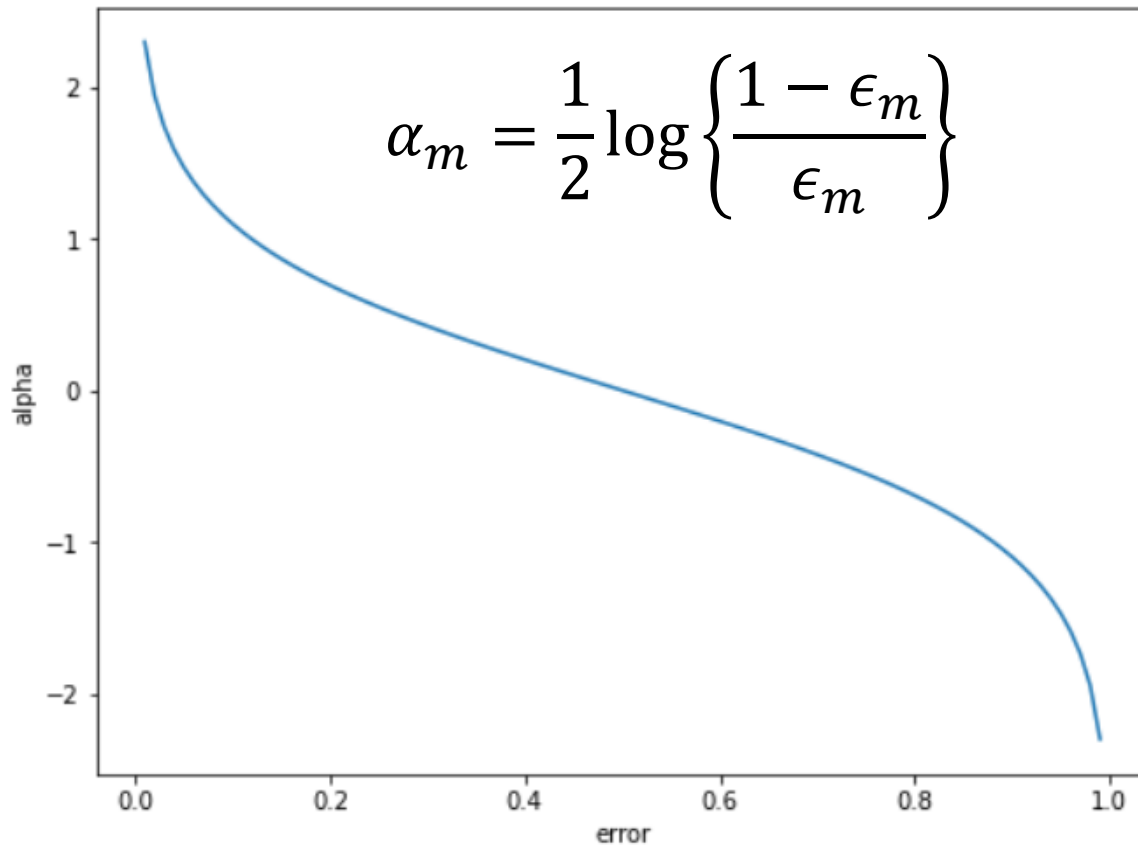
$$w_i^{(m+1)} = w_i^{(m)} e^{-\alpha_m * y_i * f_m(\mathbf{x}_i)}. \quad (\text{等同 } w_i^{(m)} e^{\alpha_m 1(f_m(\mathbf{x}_i) \neq y_i)})$$

**end for**

**end for**

- The final model is  $g(\mathbf{x}) = \text{sign}(\sum_{m=1}^M \alpha_m f_m(\mathbf{x}))$

```
import numpy as np
import matplotlib.pyplot as plt
alpha = lambda x: 0.5 * np.log((1.0-x)/x)
error = np.arange(0.01, 1.0, 0.01)
plt.figure(figsize = (8,6))
plt.xlabel('error')
plt.ylabel('alpha')
plt.plot(error,alpha(error))
plt.show()
```



Note when  $\epsilon_m \leq 0.5$

$\Rightarrow \alpha_m \geq 0$  and if  $\epsilon_m \downarrow \alpha_m \uparrow$

$$g(\mathbf{x}) = \text{sign} \left( \sum_{m=1}^M \alpha_m f_m(\mathbf{x}) \right)$$

$\alpha_m f_m(\mathbf{x})$

$\Rightarrow$  分類誤差越小的  $f_m(\mathbf{x})$  在最終集成時占比也越大

$\Rightarrow$  AdaBoost能夠適應各種  $f_m(\mathbf{x})$  的訓練誤差率  $\epsilon_m$ ，因此命名 adaptive Boosting



# Lecture 14\_1 adaboost\_oop.jpynb

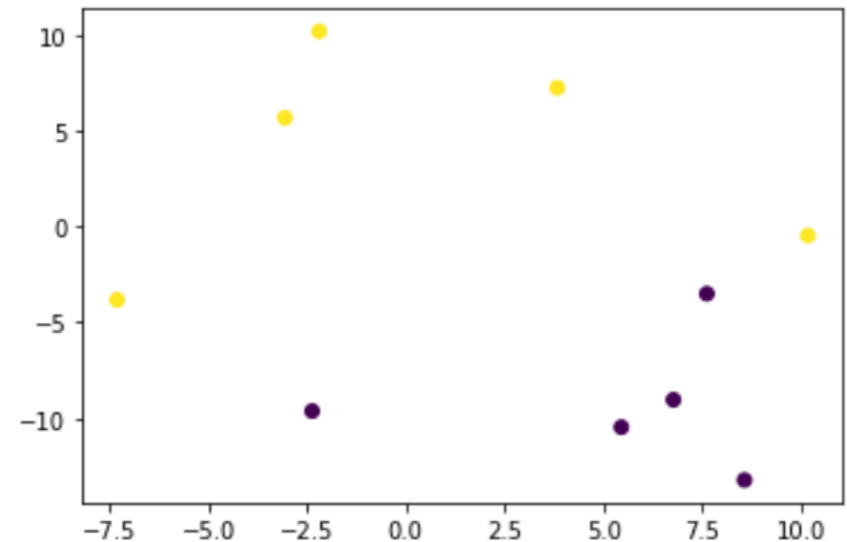
```
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
```

```
X, y = datasets.make_blobs(n_samples=10, n_features=2, centers=2, cluster_std=5.05, random_state=10)
print(y)
y[y == 0] = -1
print(y)
print(X)
```

```
[0 1 1 1 0 1 0 0 0 1]
[-1  1  1  1 -1  1 -1 -1 -1  1]
[[ 8.56415953 -13.22139309]
 [ 3.82754686  7.2240226 ]
 [10.16987656 -0.47693702]
 [-3.06687647  5.65851889]
 [-2.37785861 -9.62729945]
 [-2.20061694 10.16886174]
 [ 6.76724637 -9.03679096]
 [ 5.44808459 -10.46669208]
 [ 7.61319512 -3.50962228]
 [-7.31456312 -3.82795244]]
```

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
plt.scatter(X[:, 0], X[:, 1], marker="o", c=y)
```

<matplotlib.collections.PathCollection at 0x1efe864e280>



## Lecture 14\_1 adaboost\_oop.jpynb

```
n_samples=10
n_features=2
clfs = []
w = np.full(n_samples, (1 / n_samples))
print('w =',w)
min_error = float("inf")
feature_i = 0
X_column = X[:, feature_i]
print('feature_i=',feature_i,',X_column=',X_column)
thresholds = np.unique(X_column)
print('thresholds=',thresholds)
```

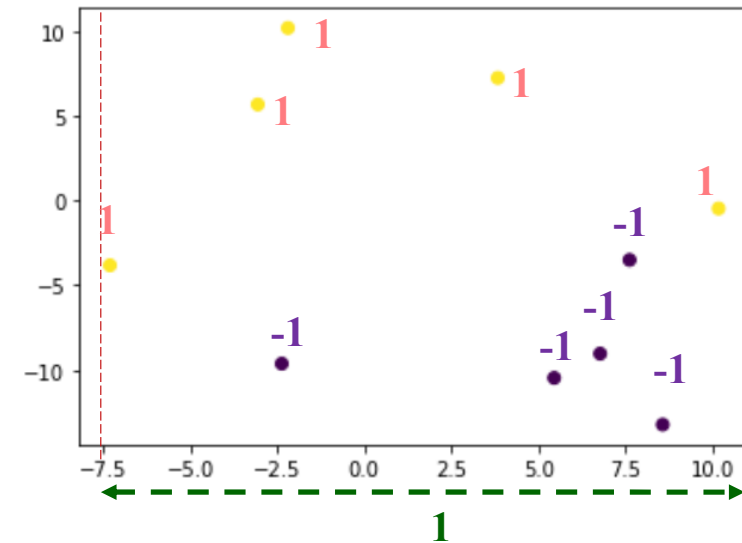
```
w = [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
feature_i= 0 ,X_column= [ 8.56415953  3.82754686 10.16987656 -3.06687647 -2.37785861 -2.20061694
 6.76724637  5.44808459  7.61319512 -7.31456312]
thresholds= [-7.31456312 -3.06687647 -2.37785861 -2.20061694  3.82754686  5.44808459
 6.76724637  7.61319512  8.56415953 10.16987656]
```



# Lecture 14\_1 adaboost\_oop.jpynb

```
print('y=',y)
for threshold in thresholds:
    p = 1      # predict with polarity 1
    predictions = np.ones(n_samples)
    predictions[X_column < threshold] = -1
    print('threshold=',threshold,',predictions=',predictions)
    # Error = sum of weights of misclassified samples
    misclassified = w[y != predictions]
    print('y != predictions',y != predictions)
    print('misclassified=w[y != predictions]=',misclassified)
    error = sum(misclassified)
    print('error=sum(misclassified)=', error)
    if error > 0.5:
        error = 1 - error
        print('error=1-error', error)
        p = -1
        print('polarity=',p)
```

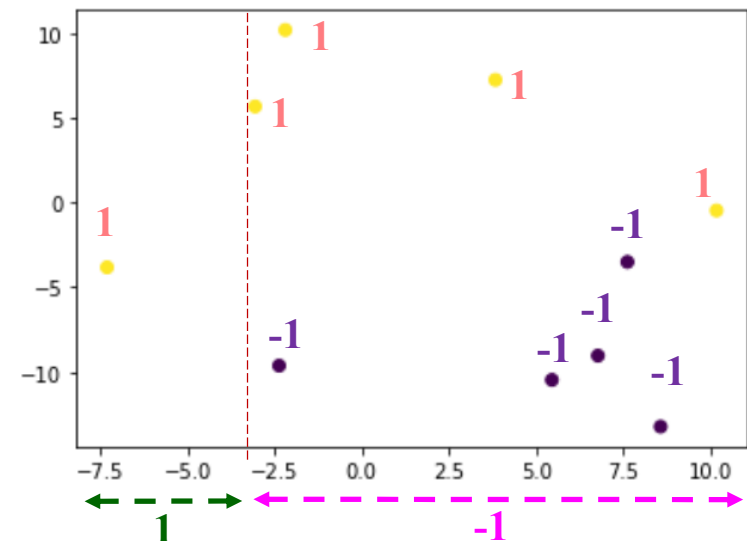
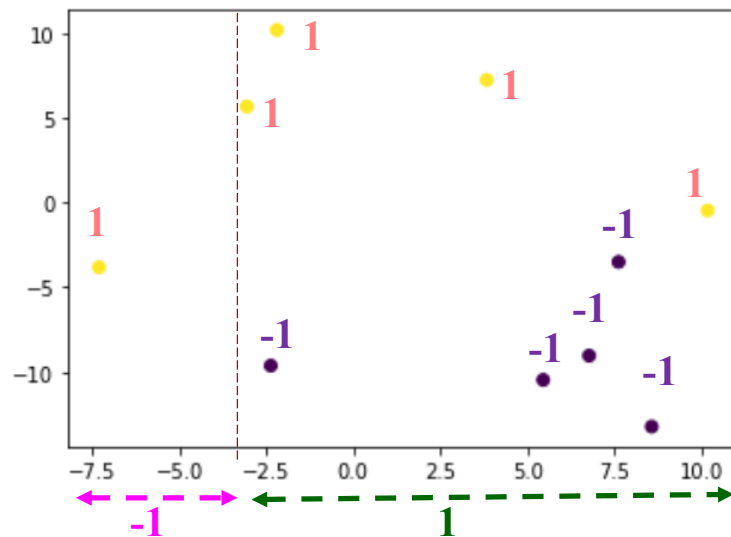
```
y= [-1  1  1  1 -1  1 -1 -1 -1  1]
threshold= -7.314563118796922 ,predictions= [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
y != predictions [ True False False False  True False  True  True  True False]
misclassified=w[y != predictions]= [0.1 0.1 0.1 0.1 0.1]
error=sum(misclassified)= 0.5
```



# Lecture 14\_1 adaboost\_oop.jpynb

```
y= [-1  1  1  1 -1  1 -1 -1 -1  1]
threshold= -3.0668764712212644 ,predictions= [ 1.  1.  1.  1.  1.  1.  1.  1.  1. -1.]
y != predictions [ True False False False  True False  True  True  True  True]
misclassified=w[y != predictions]= [0.1 0.1 0.1 0.1 0.1 0.1]
error=sum(misclassified)= 0.6
error=1-error 0.4
polarity= -1
```

```
if self.polarity == 1:
    predictions[X_column < self.threshold] = -1
else:
    predictions[X_column > self.threshold] = -1
```



# Lecture 14\_1 adaboost\_oop.jpynb

```
y= [-1  1  1  1 -1  1 -1 -1 -1  1]
```

```
threshold= -2.3778586098269843 ,predictions= [ 1.  1.  1. -1.  1.  1.  1.  1.  1. -1.]
```

```
y != predictions [ True False False  True  True False  True  True  True  True]
```

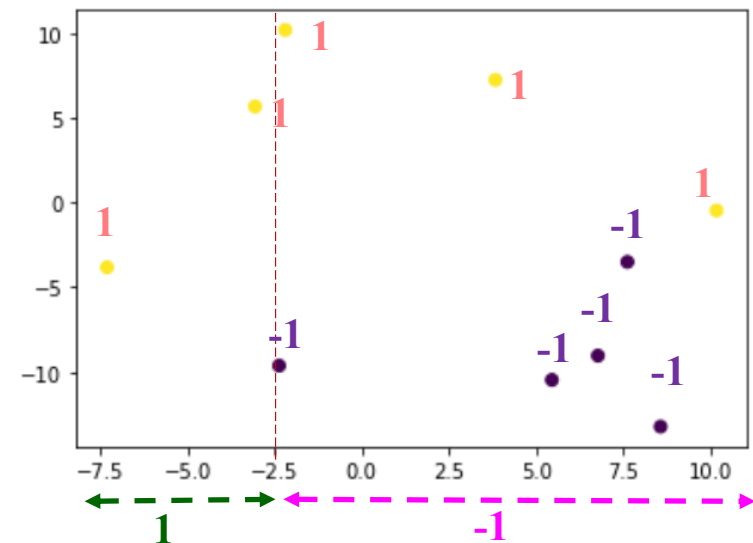
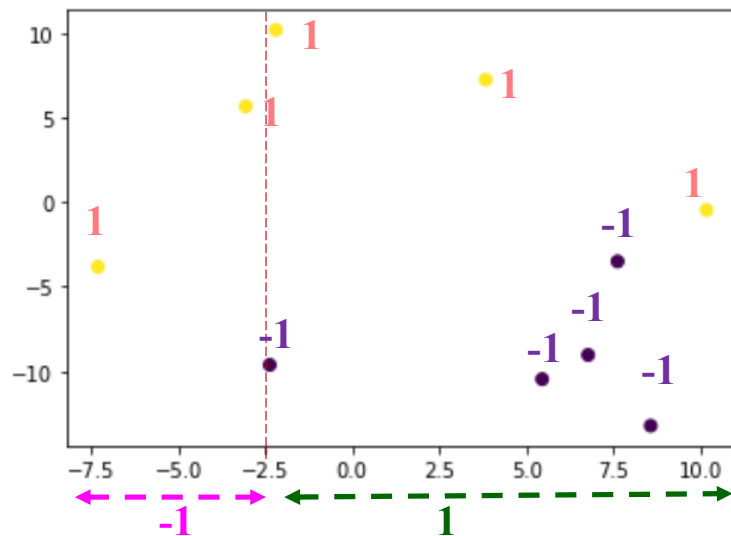
```
misclassified=w[y != predictions]= [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
```

```
error=sum(misclassified)= 0.7
```

```
error=1-error 0.30000000000000004
```

```
polarity= -1
```

```
if self.polarity == 1:  
    predictions[X_column < self.threshold] = -1  
else:  
    predictions[X_column > self.threshold] = -1
```



# Lecture 14\_1 adaboost\_oop.jpynb

```
import numpy as np

# Decision stump used as weak classifier
class DecisionStump:
    def __init__(self):
        self.polarity = 1
        self.feature_idx = None
        self.threshold = None
        self.alpha = None

    def predict(self, X):
        n_samples = X.shape[0]
        X_column = X[:, self.feature_idx]
        predictions = np.ones(n_samples)
        if self.polarity == 1:
            predictions[X_column < self.threshold] = -1
        else:
            predictions[X_column > self.threshold] = -1

        return predictions
```

# Lecture 14\_1 adaboost\_oop.jpynb

```
clf = DecisionStump()
n_samples=10
n_features=2
clfs = []
w = np.full(n_samples, (1 / n_samples))
min_error = float("inf")
for feature_i in range(n_features):
    X_column = X[:, feature_i]
    thresholds = np.unique(X_column)
    for threshold in thresholds:
        # predict with polarity 1
        p = 1
        predictions = np.ones(n_samples)
        predictions[X_column < threshold] = -1
        # Error = sum of weights of misclassified samples
        misclassified = w[y != predictions]
        error = sum(misclassified)
        if error > 0.5:
            error = 1 - error
            p = -1
        # store the best configuration
        if error < min_error:
            clf.polarity = p
            clf.threshold = threshold
            clf.feature_idx = feature_i
            min_error = error
            print('min_error=', error)
```

# Lecture 14\_1 adaboost\_oop.jpynb

```
# calculate alpha
EPS = 1e-10
clf.alpha = 0.5 * np.log((1.0 - min_error + EPS) / (min_error + EPS))
print('clf.alpha=',clf.alpha)

# calculate predictions and update weights
predictions = clf.predict(X)
print('clf.predict(X)=',predictions)

w *= np.exp(-clf.alpha * y * predictions)
# Normalize to one
w /= np.sum(w)
print('w=',w)

# Save classifier
clfs.append(clf)

min_error= 0.5
min_error= 0.4
min_error= 0.30000000000000004
min_error= 0.20000000000000007
min_error= 0.2
min_error= 0.1
y= [-1  1  1  1 -1  1 -1 -1 -1  1]
clf.alpha= 1.0986122882236653
clf.predict(X)= [-1.  1.  1.  1. -1.  1. -1. -1.  1.  1.]
w= [0.05555556 0.05555556 0.05555556 0.05555556 0.05555556 0.05555556
    0.05555556 0.05555556 0.5 0.05555556]
```

## Lecture 14\_1 adaboost\_oop.jpynb

```
clf = DecisionStump()
for feature_i in range(n_features):
    X_column = X[:, feature_i]
    thresholds = np.unique(X_column)
    for threshold in thresholds:
        # predict with polarity 1
        p = 1
        predictions = np.ones(n_samples)
        predictions[X_column < threshold] = -1
        # Error = sum of weights of misclassified samples
        misclassified = w[y != predictions]
        error = sum(misclassified)
        if error > 0.5:
            error = 1 - error
            p = -1
        # store the best configuration
        if error < min_error:
            clf.polarity = p
            clf.threshold = threshold
            clf.feature_idx = feature_i
            min_error = error
            print('min_error=', error)
```

## Lecture 14\_1 adaboost\_oop.jpynb

```
# calculate alpha
EPS = 1e-10
clf.alpha = 0.5 * np.log((1.0 - min_error + EPS) / (min_error + EPS))
print('clf.alpha=',clf.alpha)

# calculate predictions and update weights
predictions = clf.predict(X)
print('clf.predict(X)=',predictions)

w *= np.exp(-clf.alpha * y * predictions)
# Normalize to one
w /= np.sum(w)
print('w=',w)

# Save classifier
clfs.append(clf)
```

```
min_error= 0.05555555558024692
clf.alpha= 1.416606670945755
clf.predict(X)= [-1.  1.  1.  1. -1.  1. -1. -1. -1. -1.]
w= [0.02941176 0.02941176 0.02941176 0.02941176 0.02941176 0.02941176
    0.02941176 0.02941176 0.26470588 0.5      ]
y= [-1  1  1  1 -1  1 -1 -1 -1  1]
```



# Gradient Boosting

- Gradient boosting builds a series of trees, where each tree is fit on the error—the difference between the label and the predicted value—of the previous tree.
- In each round, the tree ensemble improves as we are nudging each tree more in the right direction via small updates.
- These updates are based on **a loss gradient**, which is how gradient boosting got its name.

## Logit function (log odds)

- Consider a binary classification for class  $C_1$  and  $C_2$
- Based on the Bayesian theorem, the posterior probability for class  $C_1$

$$p(C_1|\mathbf{x}) = \frac{p(\mathbf{x}|C_1)p(C_1)}{p(\mathbf{x}|C_1)p(C_1) + p(\mathbf{x}|C_2)p(C_2)} \equiv \frac{1}{1 + e^{-z}} = \sigma(z)$$

where  $\sigma(z)$  is the **logistic sigmoid function** and we have defined

$$z = \log \frac{p(\mathbf{x}|C_1)p(C_1)}{p(\mathbf{x}|C_2)p(C_2)}$$

- The inverse of the logistic sigmoid is

$$z = \log\left(\frac{\sigma}{1 - \sigma}\right) = \log\left(\frac{p(C_1|\mathbf{x})}{1 - p(C_1|\mathbf{x})}\right) = \log\left(\frac{p(C_1|\mathbf{x})}{p(C_2|\mathbf{x})}\right)$$

is known as the logit function representing the log of the ratio of

probabilities  $\log\left(\frac{p(C_1|\mathbf{x})}{p(C_2|\mathbf{x})}\right)$  for the two classes, also known as the **log odds**.

## Binary Cross Entropy Loss

- Given the training data  $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)$ , we can use maximal likelihood to estimate the model parameters  $\mathbf{w}$  and  $b$
- Let  $z = \text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \mathbf{w}\mathbf{x} + b$
- $\sigma(z) = \frac{1}{1+e^{-z}}$  is a sigmoid function interpreted as the probability of a particular sample  $\mathbf{x}$  belonging to class 1,  $\sigma(z) = p(y = 1|\mathbf{x}; \mathbf{w}, b)$
- output label (class) =  $\begin{cases} 1, & \text{if } \sigma(z) \geq 0.5 \\ 0, & \text{otherwise} \end{cases}$
- Assume  $\mathbf{x}_1, \dots, \mathbf{x}_N$  are independent, we maximize the likelihood  $L(\mathbf{w}, b)$

$$L(\mathbf{w}, b) = p(y|\mathbf{X}; \mathbf{w}, b) = \prod_{i=1}^N p(y_i|\mathbf{x}_i; \mathbf{w}, b) = \prod_{i=1}^N \sigma(z_i)^{y_i} (1 - \sigma(z_i))^{1-y_i}$$

$$\Rightarrow \min_{\mathbf{w}, b} -\log L(\mathbf{w}, b) = \min_{\mathbf{w}, b} \sum_{i=1}^N \{-y_i \sigma(z_i) - (1 - y_i)(1 - \sigma(z_i))\}$$

i.e. minimize the binary cross entropy loss

# Gradient Boosting Algorithm

1. Initialize a **decision stump** to return a constant prediction value  $\hat{y}$ .

$$F_0(x) = \underset{\hat{y}}{\operatorname{argmin}} \sum_{i=1}^N L(y_i, \hat{y})$$

2. **for**  $m = 1$  to  $M$  **do**

**2a.** Compute the difference between a predicted  $F(x_i) = \hat{y}_i$  and label  $y_i$  which is pseudo-residual  $r_{im} = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)}$ ,  $i = 1, \dots, N$

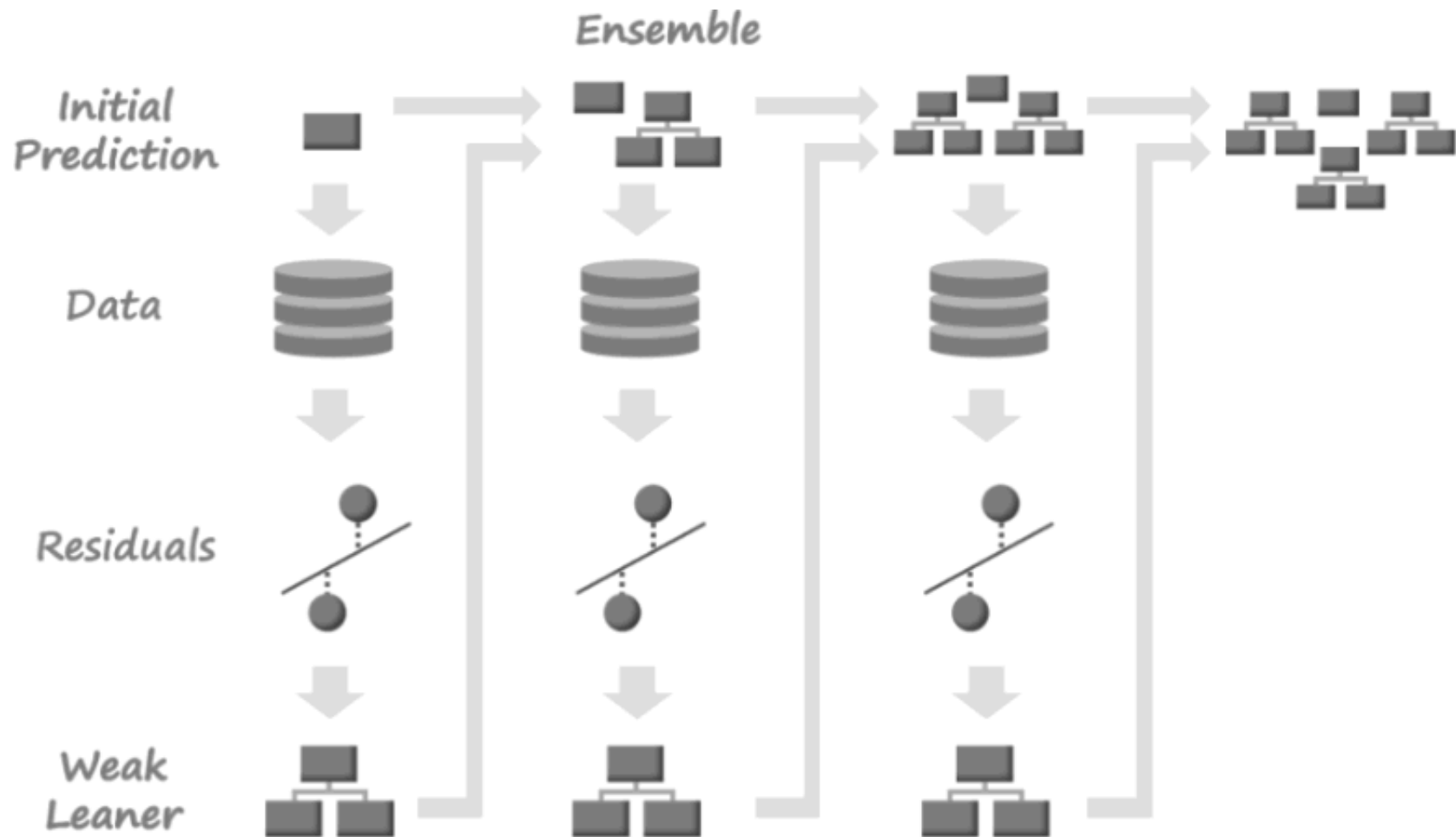
**2b.** Fit a tree to the pseudo-residuals  $r_{im}$  and create terminal node  $R_{jm}, j = 1, \dots, J_m$  of the resulting tree in iteration  $m$

**2c.** For each terminal node  $R_{jm}$  compute output value

$$\gamma_{jm} = \underset{\gamma}{\operatorname{argmin}} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x) + \gamma), \quad j = 1, \dots, J_m$$

**2d.** Update the model:  $F_m(x) = F_{m-1}(x) + \eta \sum_{j=1}^{J_m} \gamma_{jm} 1(x \in R_{jm})$

# Gradient Boosting Algorithm



## Gradient Boosting Derivations

- Introduce the log(odds):  $\hat{y} = \log(\text{odds}) = \log(\frac{p}{1-p})$ , note  $\mathbf{z} \rightsquigarrow \hat{y}$   
 $\Rightarrow e^{\hat{y}} = \frac{p}{1-p} \Rightarrow e^{\hat{y}}(1-p) = p \Rightarrow p(1 + e^{\hat{y}}) = e^{\hat{y}} \Rightarrow p = \frac{e^{\hat{y}}}{1+e^{\hat{y}}} = \frac{1}{1+e^{-\hat{y}}}$
- For logistic regression,  $\sigma(\hat{y}_i) = p(y = 1|\mathbf{x}_i) \equiv p_i$ , the logistic loss for a single training example  $\mathbf{x}_i$  is

$$\begin{aligned} L_i &= -y_i \log p_i - (1 - y_i) \log(1 - p_i) \\ &= -(y_i(\log p_i - \log(1 - p_i))) - \log(1 - p_i) \\ &= -\left(y_i \log\left(\frac{p_i}{1 - p_i}\right) + \log(1 - p_i)\right) \\ &= -(y_i \hat{y}_i + \log(1 - p_i)) \\ &= -\left(y_i \hat{y}_i + \log\left(1 - \frac{e^{\hat{y}_i}}{1+e^{\hat{y}_i}}\right)\right), 1 - \frac{e^{\hat{y}_i}}{1+e^{\hat{y}_i}} = \frac{1}{1+e^{\hat{y}_i}} \\ &= -\left(y_i \hat{y}_i + \log\left(\frac{1}{1+e^{\hat{y}_i}}\right)\right) = \log(1 + e^{\hat{y}_i}) - y_i \hat{y}_i \end{aligned}$$

# Gradient Boosting Derivations

1. Initialize a **decision stump** to return a constant prediction value  $\hat{y}$ .

$$\min_{\hat{y}} \sum_{i=1}^N L(y_i, \hat{y}) \Rightarrow \frac{\partial}{\partial \hat{y}} \sum_{i=1}^N (\log(1 + e^{\hat{y}}) - y_i \hat{y}) = 0$$

Note

$$\frac{\partial}{\partial \hat{y}} (\log(1 + e^{\hat{y}}) - y_i \hat{y}) = \frac{e^{\hat{y}}}{1 + e^{\hat{y}}} - y_i = p - y_i$$

$$0 = \frac{\partial}{\partial \hat{y}} \sum_{i=1}^N (\log(1 + e^{\hat{y}}) - y_i \hat{y}) = \sum_{i=1}^N (p - y_i) = Np - \sum_{i=1}^N y_i \Rightarrow p = \frac{1}{N} \sum_{i=1}^N y_i \equiv \bar{y}$$

Therefore

$$F_0(x) = \operatorname{argmin}_{\hat{y}} \sum_{i=1}^N L(y_i, \hat{y}) = \hat{y}_{optimal} = \log\left(\frac{p}{1-p}\right) = \log\left(\frac{\bar{y}}{1-\bar{y}}\right)$$

## Gradient Boosting Derivations

**2a.** We compute the pseudo-residual, which is the negative partial derivative of the loss with respect to the  $\log(\text{odds}) = F(x_i) = \hat{y}_i$ , which turns out to be the difference between the class label and the predicted probability:

$$\begin{aligned} r_{im} &= - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)} \\ &= - \frac{\partial L_i(y_i, \hat{y}_i)}{\partial \hat{y}_i} \\ &= - \frac{\partial}{\partial \hat{y}_i} (\log(1 + e^{\hat{y}_i}) - y_i \hat{y}_i) \\ &= - \frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}} + y_i \\ &= y_i - p_i \end{aligned}$$



## Gradient Boosting Derivations

**2b.** Train regression tree with features  $x_i$  against  $r_{im}$  and create terminal node  $R_{jm}$  for  $j = 1, \dots, J_m$ , where  $m$  denotes the tree index and  $J$  denotes the total number of leaves.

**2c.** For each terminal node  $R_{jm}$ , compute a value  $\gamma_{jm}$  that minimizes the logistic loss function  $\sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x) + \gamma)$

- Using the Taylor expansion up to the second order

$$L(y_i, F_{m-1}(x) + \gamma) \approx L(y_i, F_{m-1}(x)) + \frac{\partial L(y_i, F_{m-1}(x))}{\partial F} \gamma + \frac{1}{2} \frac{\partial^2 L(y_i, F_{m-1}(x))}{\partial F^2} \gamma^2$$

$$\Rightarrow \frac{\partial L(y_i, F_{m-1}(x) + \gamma)}{\partial \gamma} = 0 + \frac{\partial L(y_i, F_{m-1}(x))}{\partial F} + \frac{\partial^2 L(y_i, F_{m-1}(x))}{\partial F^2} \gamma$$

## Gradient Boosting Derivations

$$0 = \frac{\partial}{\partial \gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x) + \gamma) = \sum_{x_i \in R_{jm}} \left( \frac{\partial L(y_i, F_{m-1}(x))}{\partial F} + \frac{\partial^2 L(y_i, F_{m-1}(x))}{\partial F^2} \gamma \right)$$

$$\Rightarrow \gamma_{jm} = \frac{-\sum_{x_i \in R_{jm}} \frac{\partial L(y_i, F_{m-1}(x))}{\partial F}}{\sum_{x_i \in R_{jm}} \frac{\partial^2 L(y_i, F_{m-1}(x))}{\partial F^2}} = \frac{\sum_{x_i \in R_{jm}} (y_i - \textcolor{red}{p}_i)}{\sum_{x_i \in R_{jm}} \frac{\partial}{\partial F}(-y_i + \textcolor{red}{p}_i)} = \frac{\sum_{x_i \in R_{jm}} (y_i - \textcolor{red}{p}_i)}{\sum_{x_i \in R_{jm}} \textcolor{blue}{p}_i(1 - \textcolor{blue}{p}_i)}$$

$$\begin{aligned} \frac{\partial}{\partial F}(-y_i + \textcolor{red}{p}_i) &= \frac{\partial}{\partial \hat{y}_i} \left( -y_i + \frac{\textcolor{red}{e}^{\hat{y}_i}}{1 + \textcolor{red}{e}^{\hat{y}_i}} \right) = 0 + \frac{\partial}{\partial \hat{y}_i} \left( \frac{1}{1 + \textcolor{red}{e}^{-\hat{y}_i}} \right) \\ &= -(1 + \textcolor{red}{e}^{-\hat{y}_i})^{-2} (-\textcolor{red}{e}^{-\hat{y}_i}) = \frac{1}{1 + \textcolor{red}{e}^{-\hat{y}_i}} \left( 1 - \frac{1}{1 + \textcolor{red}{e}^{-\hat{y}_i}} \right) \\ &= \textcolor{blue}{p}_i(1 - \textcolor{blue}{p}_i) \end{aligned}$$

## Gradient Boosting Derivations

**2d.** In the final step, updating the prediction of the combined model  $F_m$

$$F_m(x) = F_{m-1}(x) + \eta \sum_{j=1}^{J_m} \gamma_{jm} 1(x \in R_{jm})$$

- $\gamma_{jm} 1(x \in R_{jm})$  means that we pick the value  $\gamma_{jm}$  if a given  $x$  falls in a terminal node  $R_{jm}$ .
- As all the terminal nodes are exclusive, any given single  $x$  **falls into only a single terminal node** and corresponding  $\gamma_{jm}$  is added to the previous prediction  $F_{m-1}$  and then it makes the updated prediction

## Toy dataset for explaining gradient boosting

	Feature $x_1$	Feature $x_2$	Class label $y$
1	1.12	1.4	1 (success)
2	2.45	2.1	0 (failure)
3	3.54	1.2	1 (success)

- step 1, constructing the root node and computing the optimally constant prediction value  $\hat{y}_{optimal} = \log(\text{odds}) = \log\left(\frac{p}{1-p}\right) = \log\frac{2/3}{1/3} = \log 2 = 0.69$
- step 2a, converting the log(odds) into class-membership probabilities and computing the pseudo-residuals:

$$r_{im} = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)} = y_i - p_i, \quad p_i = \frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}} = \frac{1}{1 + e^{-\hat{y}_i}}$$

## Toy dataset for explaining gradient boosting

	Feature $x_1$	Feature $x_2$	Class label $y$	Step 1: $\hat{y} = \log(\text{odds})$	Step 2a: $p = \frac{1}{1 + e^{-\hat{y}}}$	Step 2a: $r = y - p$	
1	1.12	1.4	1	0.69	0.67	0.33	$= 1 - 0.67$
2	2.45	2.1	0	0.69	0.67	-0.67	$= 0 - 0.67$
3	3.54	1.2	1	0.69	0.67	0.33	$= 1 - 0.67$

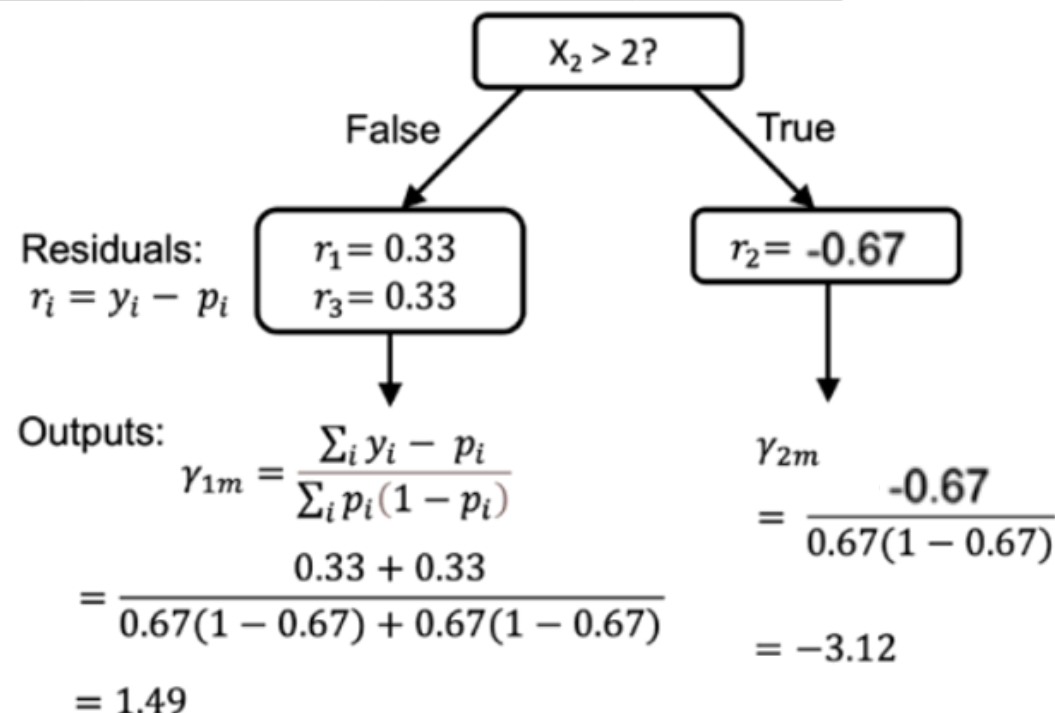
- step 1, constructing the root node and computing the optimally constant prediction value  $\hat{y}_{\text{optimal}} = \log(\text{odds}) = \log\left(\frac{p}{1-p}\right) = \log\frac{2/3}{1/3} = \log 2 = 0.69$
- step 2a, converting the  $\log(\text{odds})$  into class-membership probabilities and computing the pseudo-residuals:

$$r_{im} = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)} = y_i - p_i, \quad p_i = \frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}} = \frac{1}{1 + e^{-\hat{y}_i}}$$

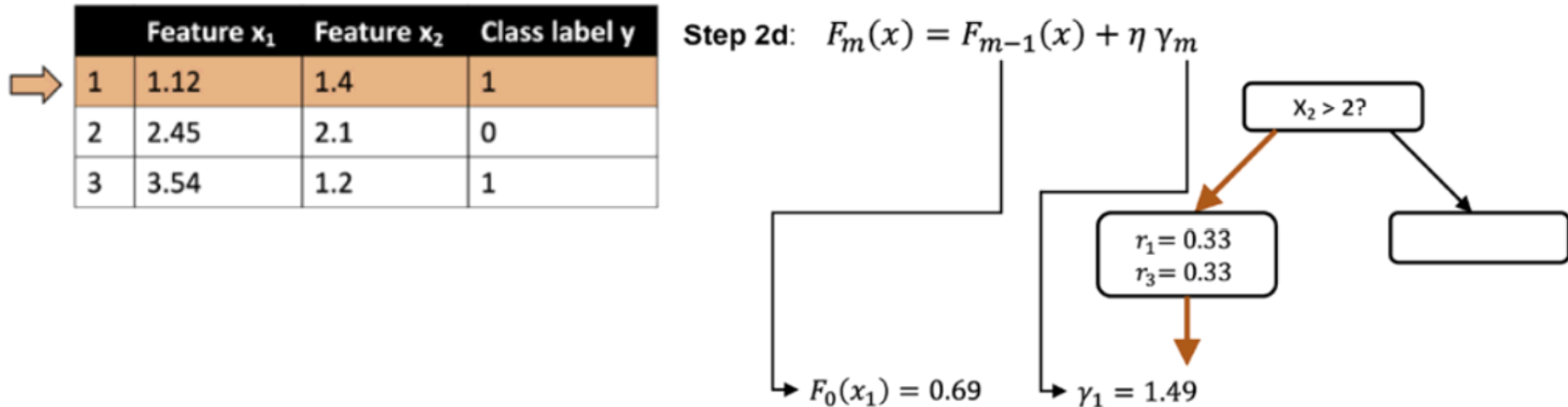
Step 2b fits a tree to the residuals, step 2c computes the output values  $\gamma_{jm}$  for each leaf node

	Feature $x_1$	Feature $x_2$	Class label $y$	Step 1: $\hat{y} = \log(\text{odds})$	Step 2a: $p = \frac{1}{1 + e^{-\hat{y}}}$	Step 2a: $r = y - p$
1	1.12	1.4	1	0.69	0.67	0.33
2	2.45	2.1	0	0.69	0.67	-0.67
3	3.54	1.2	1	0.69	0.67	0.33

$$\gamma_{jm} = \frac{\sum_{x_i \in R_{jm}} (y_i - p_i)}{\sum_{x_i \in R_{jm}} p_i(1 - p_i)}$$



**Step 2d updates the current model assuming  $\eta = 0.1$**



$$m = 1,$$


$$F_1(x_1) = F_0(x_1) + \eta \gamma_{11} = 0.69 + 0.1 \times 1.49 = 0.839$$

$$F_1(x_2) = F_0(x_2) + \eta \gamma_{21} = 0.69 + 0.1 \times (-3.12) = 0.378$$


$$F_1(x_3) = F_0(x_3) + \eta \gamma_{31} = 0.69 + 0.1 \times 1.49 = 0.839$$

## Values from the second round

	$x_1$	$x_2$	$y$	Step 1: $F_0(x) = \hat{y}$ $= \log(\text{odds})$	Step 2a: $p = \frac{1}{1 + e^{-\hat{y}}}$	Step 2a: $r = y - p$	New log(odds) $\hat{y} = F_1(x)$	Step 2a: $p$	Step 2a: $r$
1	1.12	1.4	1	0.69	0.67	0.33	0.839	0.698	0.302
2	2.45	2.1	0	0.69	0.67	-0.67	0.378	0.593	-0.593
3	3.54	1.2	1	0.69	0.67	0.33	0.839	0.698	0.302



Round  $m = 1$



Round  $m = 2$

$m = 2,$

$$F_2(x_1) = F_1(x_1) + \eta \gamma_{12} = 0.839 + 0.1 \times \gamma_{12}$$

$$F_2(x_2) = F_1(x_2) + \eta \gamma_{22} = 0.378 + 0.1 \times \gamma_{22}$$

$$F_2(x_3) = F_1(x_3) + \eta \gamma_{32} = 0.839 + 0.1 \times \gamma_{32}$$



# Gradient Boosting Algorithm

1. Initialize a **decision stump** to return a constant prediction value  $\hat{y}$ .

$$F_0(x) = \operatorname{argmin}_{\hat{y}} \sum_{i=1}^N L(y_i, \hat{y})$$

2. **for**  $m = 1$  to  $M$  **do**

**2a.** Compute the difference between a predicted  $F(x_i) = \hat{y}_i$  and label  $y_i$  which is pseudo-residual  $r_{im} = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)}, i = 1, \dots, N$

**2b.** Fit a tree to the pseudo-residuals  $r_{im}$  and create terminal node  $R_{jm}, j = 1, \dots, J_m$  of the resulting tree in iteration  $m$

**2c.** For each terminal node  $R_{jm}$  compute output value

$$\gamma_{jm} = \operatorname{argmin}_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x) + \gamma), \quad j = 1, \dots, J_m$$

**2d.** Update the model:  $F_m(x) = F_{m-1}(x) + \eta \sum_{j=1}^{J_m} \gamma_{jm} 1(x \in R_{jm})$

## Gradient Boosting Algorithm

- The predicted probabilities are higher for the positive class and lower for the negative class  $\Rightarrow$  the residuals are getting smaller.
- Steps 2a to 2d is repeated until we have fit  $M$  trees or the residuals are smaller than a user-specified threshold value.
- Once the algorithm is completed, we predict the class labels by thresholding the probability values of the final model  $F_M(x)$  at 0.5, like logistic regression.
- In contrast to logistic regression, gradient boosting consists of multiple trees and produces nonlinear decision boundaries.
- XGBoost stands for extreme gradient boosting that proposed several tricks and approximations that speed up the training process substantially and has very good predictive performances.

## Comparing AdaBoost with gradient boosting

- AdaBoost trains decision tree stumps based on errors of the previous decision tree stump.
- The errors are used to compute sample weights in each round as well as for computing a classifier weight for each decision tree stump when combining the individual stumps into an ensemble.
- We stop training once a maximum number of iterations (decision tree stumps) is reached.

## Comparing AdaBoost with gradient boosting

- Like AdaBoost, gradient boosting fits decision trees in an iterative fashion using prediction errors.
- However, gradient boosting trees are usually **deeper** than decision tree stumps and have typically a maximum depth of 3 to 6 (or a maximum number of 8 to 64 leaf nodes).
- In contrast to AdaBoost, gradient boosting does not use the prediction errors for assigning sample weights; they are used directly to form the target variable for fitting the next tree.
- Instead of having an individual weighting term for each tree in AdaBoost, gradient boosting uses a global learning rate that is the same for each tree.

# Homework: Watch StatQuest

AdaBoost:

<https://www.youtube.com/watch?v=LsK-xG1cLYA>

Gradient Boost

Part I <https://www.youtube.com/watch?v=3CC4N4z3GJc&t=50s>

Part II <https://www.youtube.com/watch?v=2xudPOBz-vs>

Part III <https://www.youtube.com/watch?v=jxuNLH5dXCs>

Part IV <https://www.youtube.com/watch?v=StWY5QWMXCw>