Ensemble Learning (1): Boosting

Yanrong Yang

RSFAS/CBE, Australian National University

20 September 2022

Contents of this week

Boosting is an ensemble learning technique.

- Gradient Boosting for Regression
- AdaBoost for Classification
- Generalization: performances on test data

Ensemble Learning

We have studied several methods for regression and classification problems:

- Regression - Least squares estimation, shrinkage methods (the lasso, ridge regression, the elastic net), kernel smoothing method, local polynomial method, decision trees.
- Classification - Logistic regression, LDA/QDA, decision trees, support vector classifiers

Core idea of decision trees?

- Build complex classifiers from simpler ones.
- Borrowing this idea, ensemble learning combine "simple" methods and create a final "reasonable" result.

Methods belonging to Ensemble Learning:

- Bagging
- Random forests
- Boosting

Ensemble Learning

Ensemble learning methods differ in training strategy and combination method.

- Bagging and Random Forest: parallel training with different training sets;
- Boosting: sequential training, iteratively re-weighting training observations so current classifier focuses on hard observations;
- Mixture of Experts: parallel training with objective encouraging division of labor.

Ensemble learning is also known as meta-learning.

Motivation of Boosting

Before formulating the problem, we give a little intuition for what Boosting is going to do.

- Roughly, the idea of boosting is to take a weak learning algorithm (any learning algorithm that gives a classifier that is slightly better than random guess), and then transform it into a strong classifier (which does much better than random).
- Consider a hypothetical digit recognition experiment, where we wish to distinguish 0 from 1, and we receive images we must classify. A natural weak learner is to take the middle pixel of the image, and if it is colored, call the image a 1, and if it is blank, call the image a 0. This classifier may be far from perfect, but it is likely better than random guess. Boosting procedures proceed by taking a collection of such weak classifiers, and then reweighting their contributions to form a classifier with much better accuracy than any individual classifier.

Gradient Boosting

Idea of Gradient Boosting

- The key intuition behind boosting is to take an ensemble of simple models $\{T_h : h = 1, 2, \ldots, M\}$ and additively combine them into a single more complex model.

$$T = \sum_{h=1}^{M} \lambda_h T_h. \tag{1}$$

- Questions:
  - Which simple models should we involve in this ensemble?
  - What should the weights $\{\lambda_h, h = 1, 2, \ldots, M\}$ be?

Gradient Boosting Algorithm

Framework of gradient boosting algorithm

1. Fit a simple model $T^{(0)}$ on the training data

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

Set $T \leftarrow T^{(0)}$. Compute the residuals $\{r_1, \ldots, r_N\}$ for T.

2. Fit a simple model, $T^{(1)}$, to the current ***residuals***, i.e. train using

$$\{(x_1, r_1), \ldots, (x_N, r_N)\}$$

3. Set $T \leftarrow T + \lambda T^{(1)}$
4. Compute residuals, set $r_n \leftarrow r_n - \lambda T^i(x_n),\ n = 1, \ldots, N$
5. Repeat steps 2-4 until ***stopping*** condition met.

Questions:

1. Any requirement on simple models? 2. How to determine the tuning parameter λ?

Interpretation of Gradient Boosting

Intuitively, each simple model $T^{(i)}$ we add to our ensemble model T, models the errors of T.

Thus, with each addition of $T^{(i)}$, the residual is reduced

$$r_n - \lambda T^{(i)}(x_n)$$

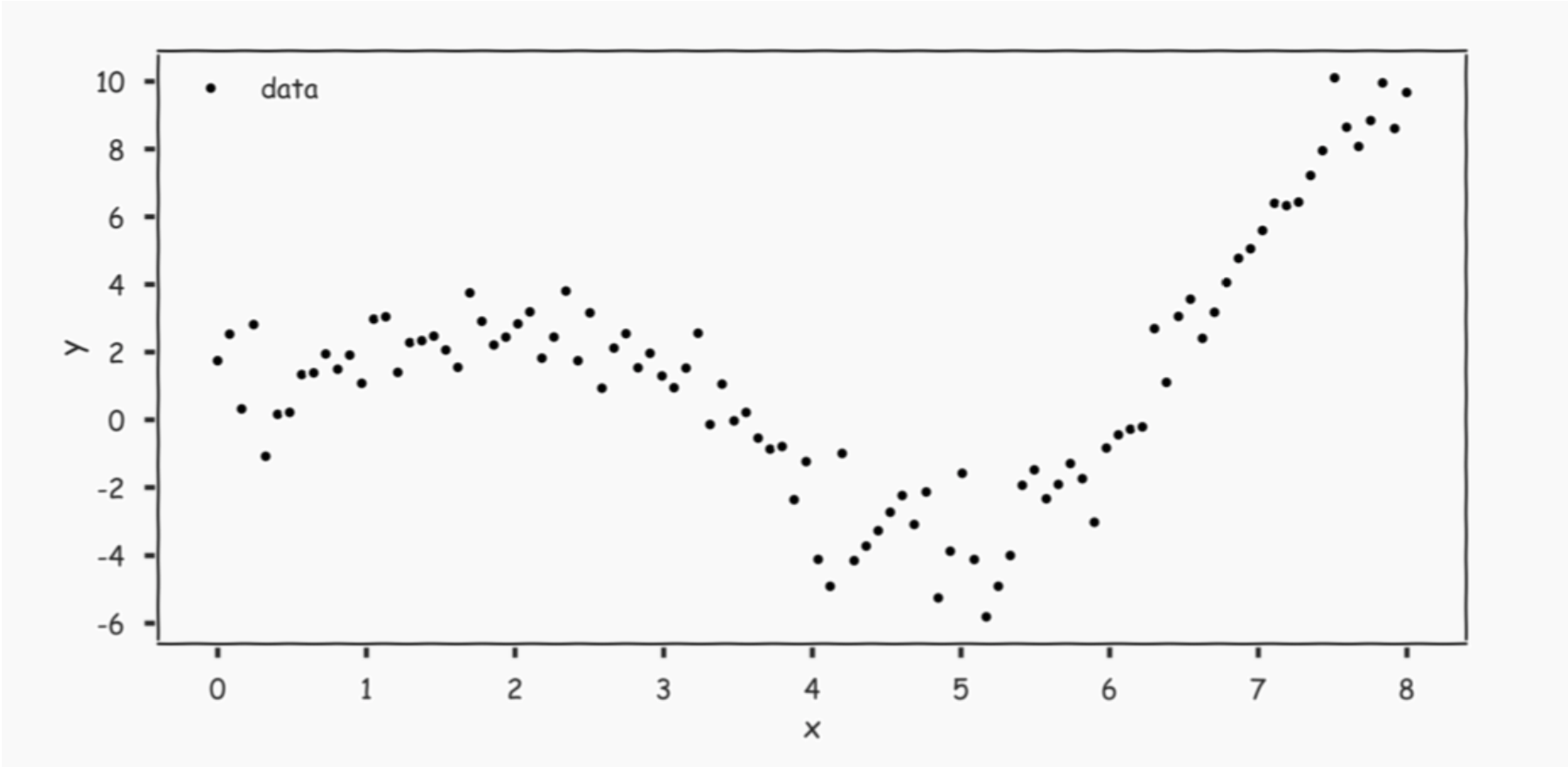
Note that gradient boosting has a tuning parameter, λ.

If we want to easily reason about how to choose λ and investigate the effect of λ on the model T, we need a bit more mathematical formalism.

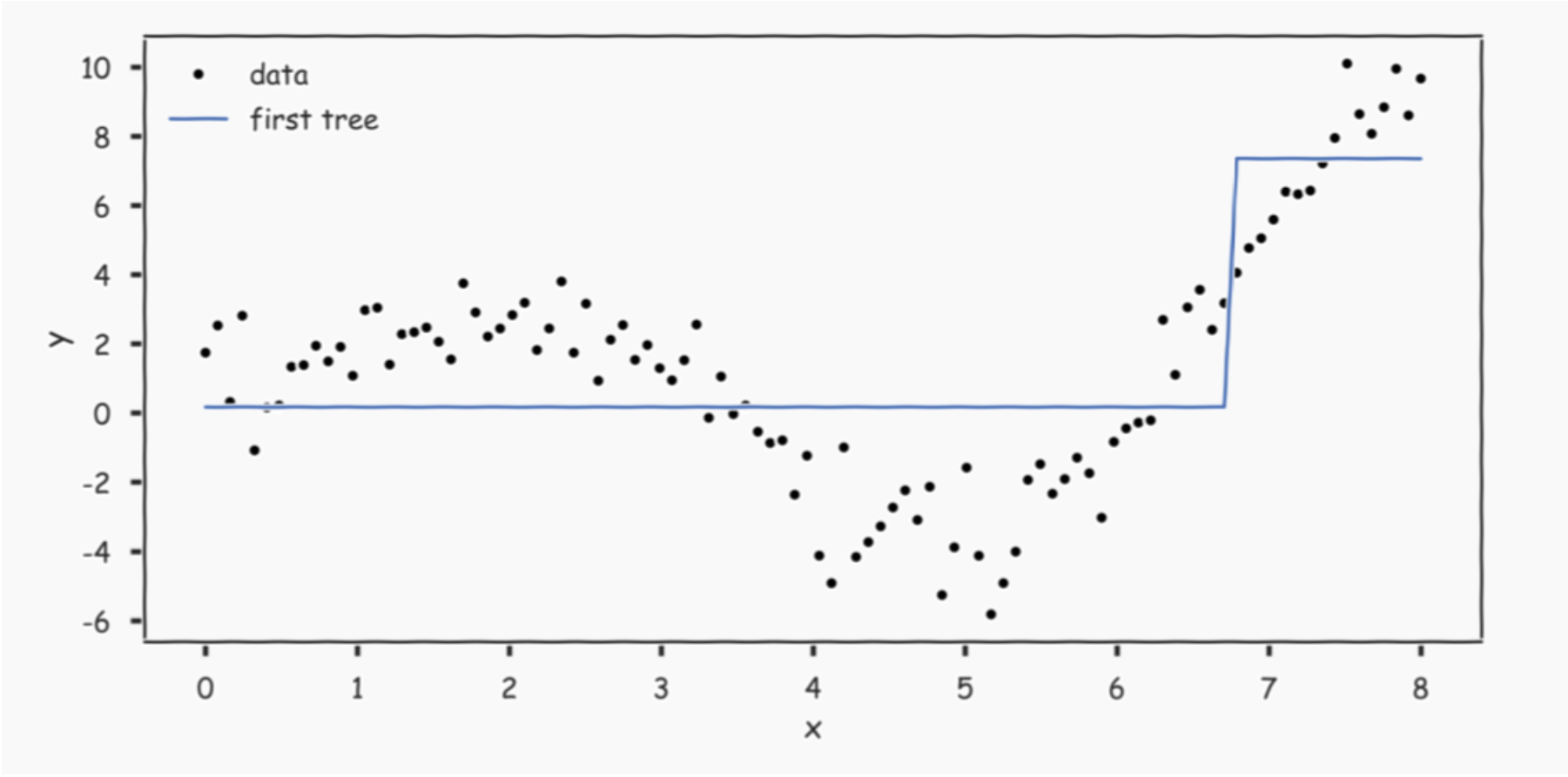
In particular, how can we effectively descend through this optimization via an iterative algorithm?

We need to formulate gradient boosting as a type of ***gradient descent***.

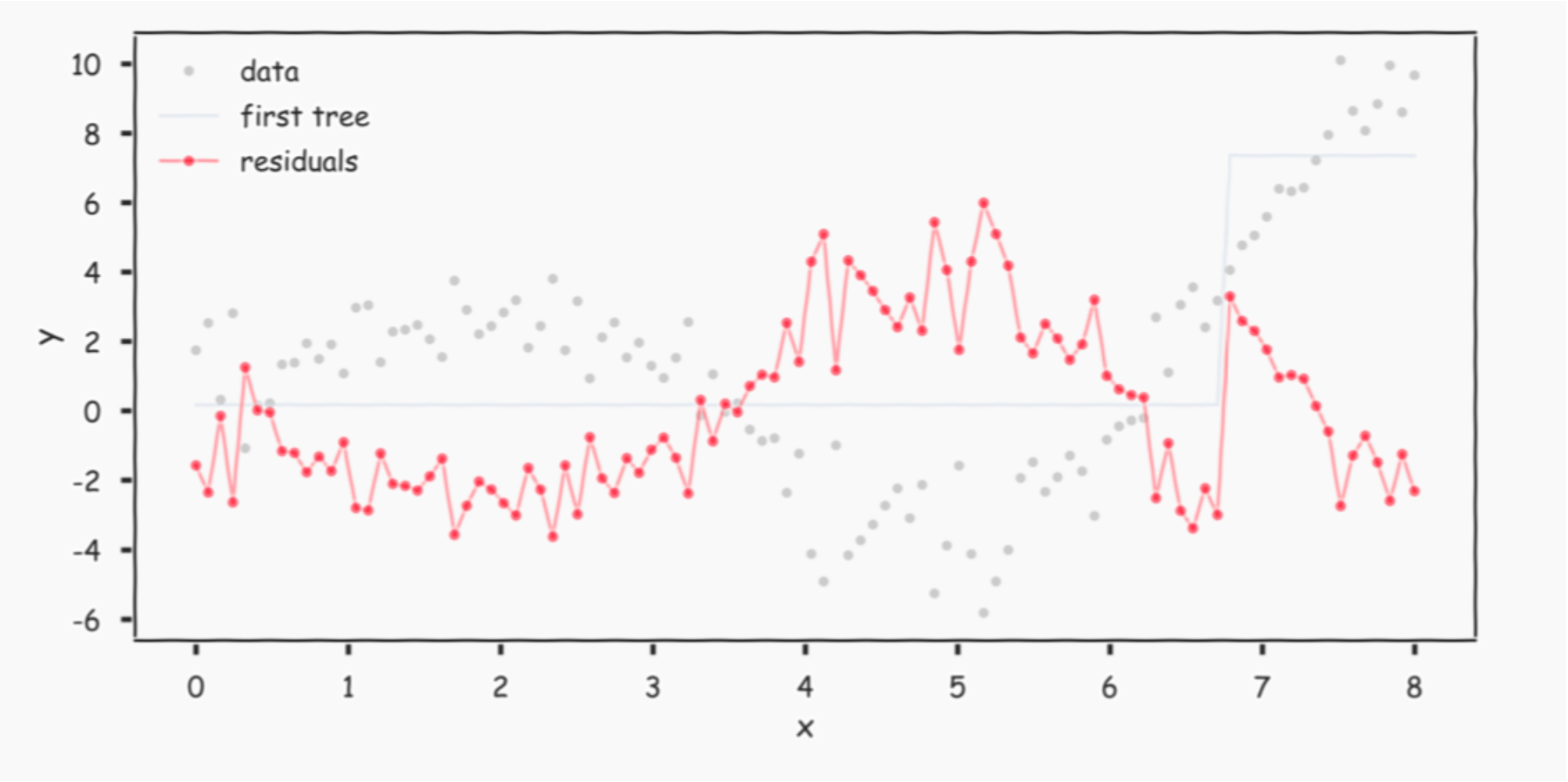
Example: Gradient Boosting



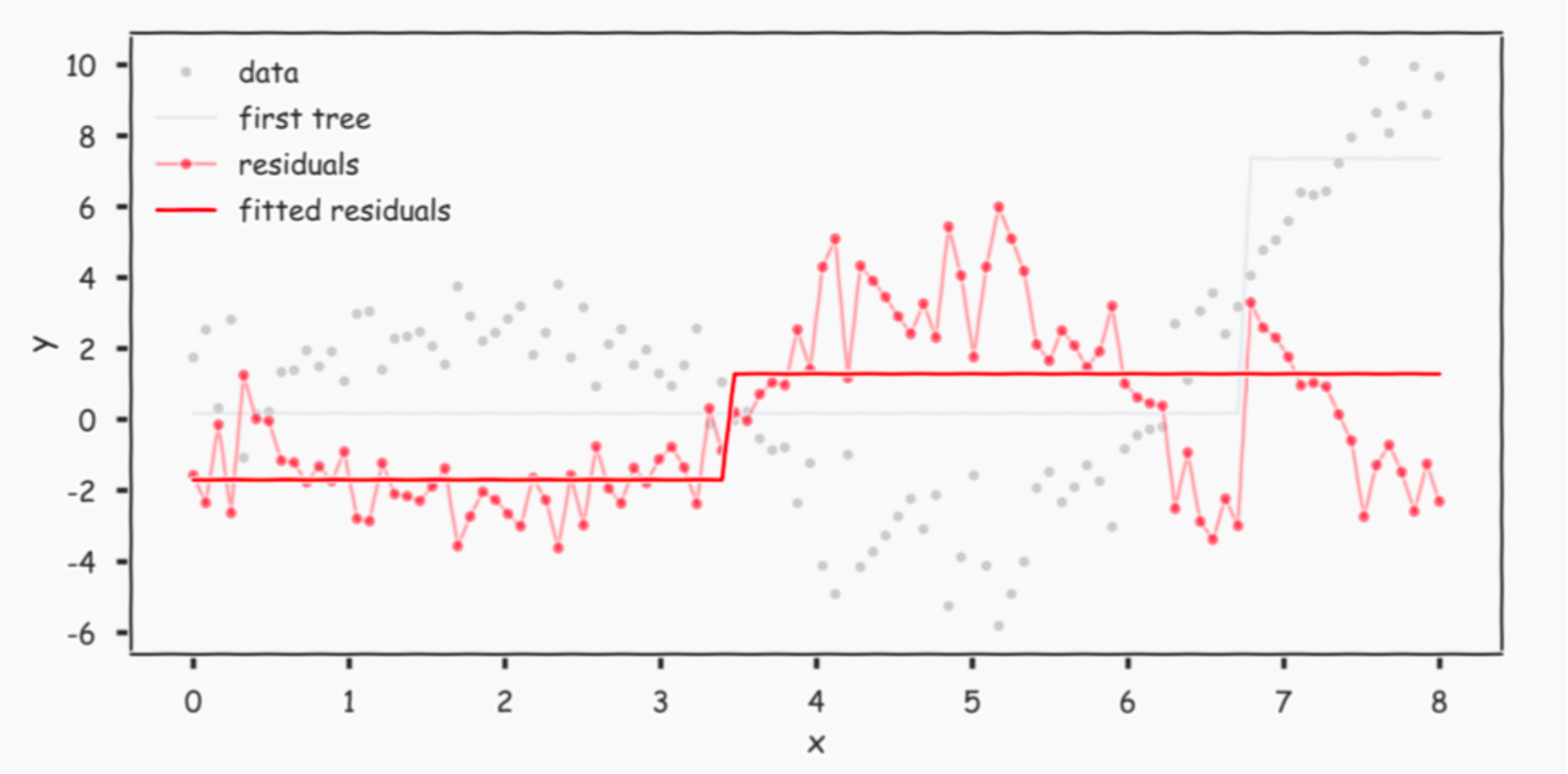
Example: Gradient Boosting



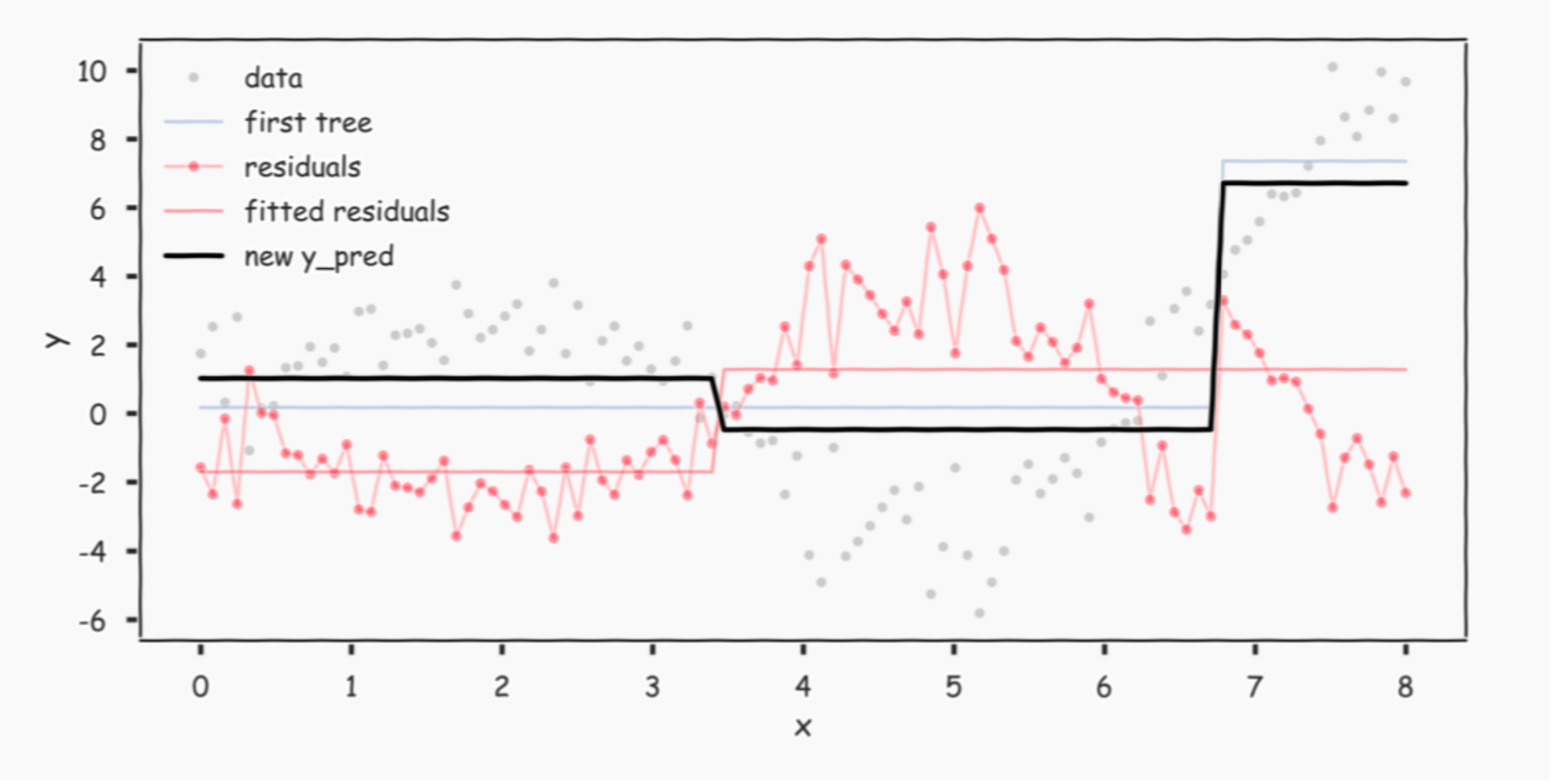
Example: Gradient Boosting



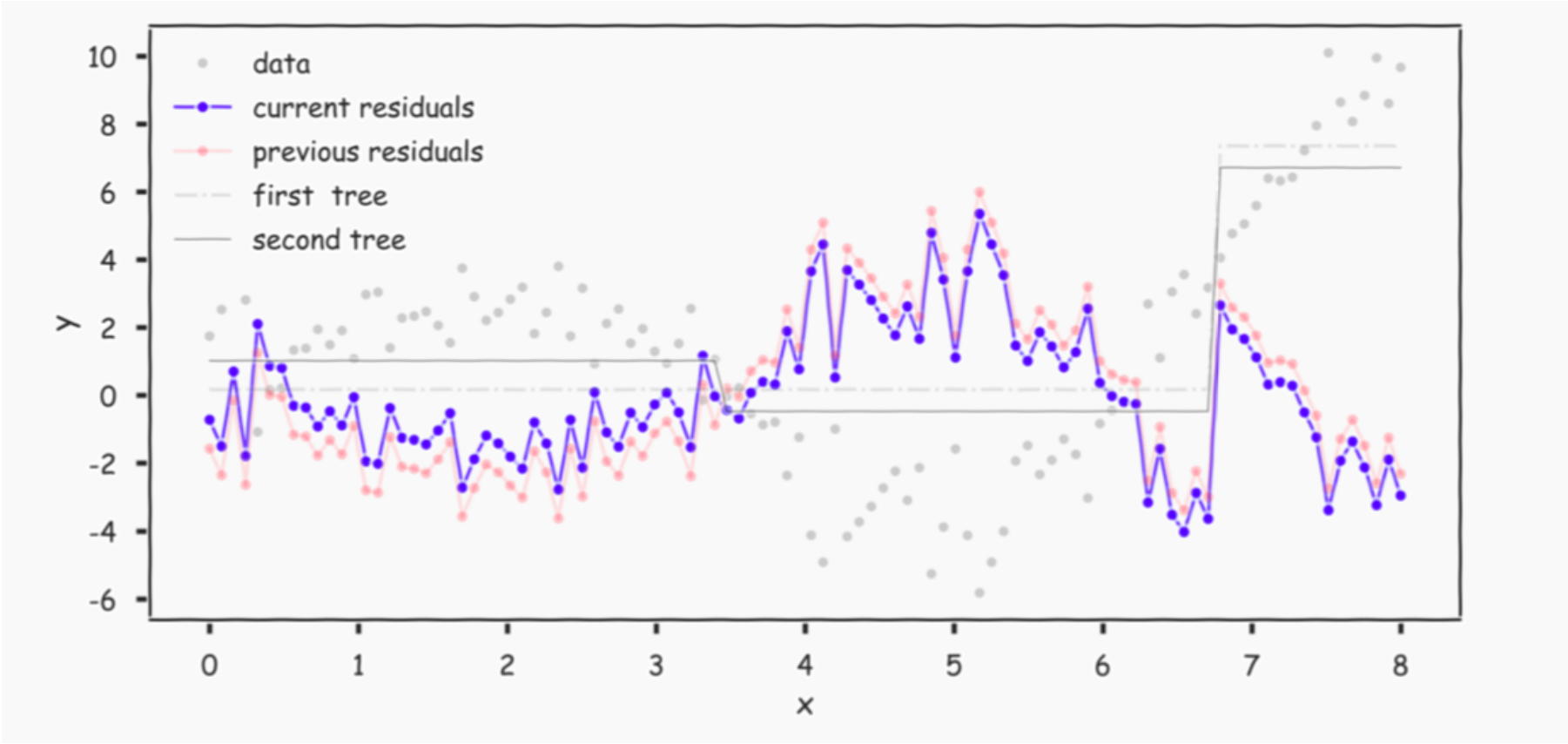
Example: Gradient Boosting



Example: Gradient Boosting



Example: Gradient Boosting



Brief Introduction to Gradient Descent

In optimization, when we wish to minimize a function, called the ***objective function***, over a set of variables, we compute the partial derivatives of this function with respect to the variables.

If the partial derivatives are sufficiently simple, one can analytically find a common root - i.e. a point at which all the partial derivatives vanish; this is called a ***stationary point.***

If the objective function has the property of being ***convex***, then the stationary point is precisely the min.

Framework of Gradient Descent

In practice, our objective functions are complicated and analytically find the stationary point is intractable.

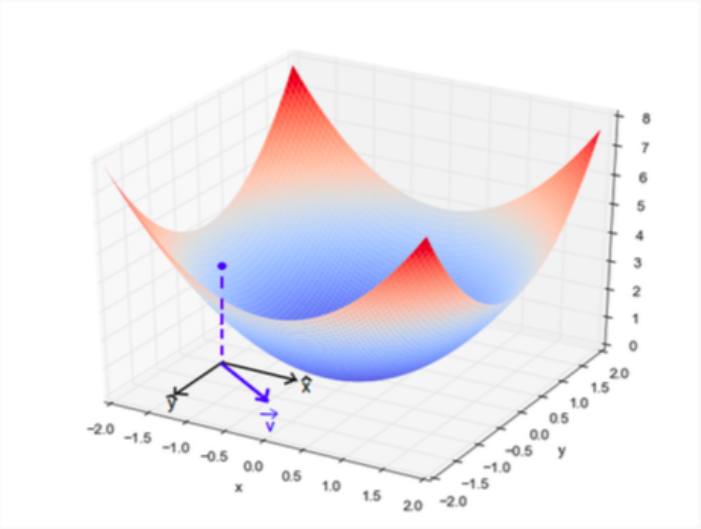
Instead, we use an iterative method called ***gradient descent***:

1. Initialize the variables at any value:
$$x = [x_1, ..., x_J]$$
2. Take the gradient of the objective function at the current variable values:
$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_J}(x)\right]$$
3. Adjust the variables values by some negative multiple of the gradient:
$$x \leftarrow x - \lambda \nabla f(x)$$
The factor λ is often called the learning rate.

Gradient Descent Algorithm

Claim: If the function is convex, this iterative methods will eventually move x close enough to the minimum, for an appropriate choice of λ.

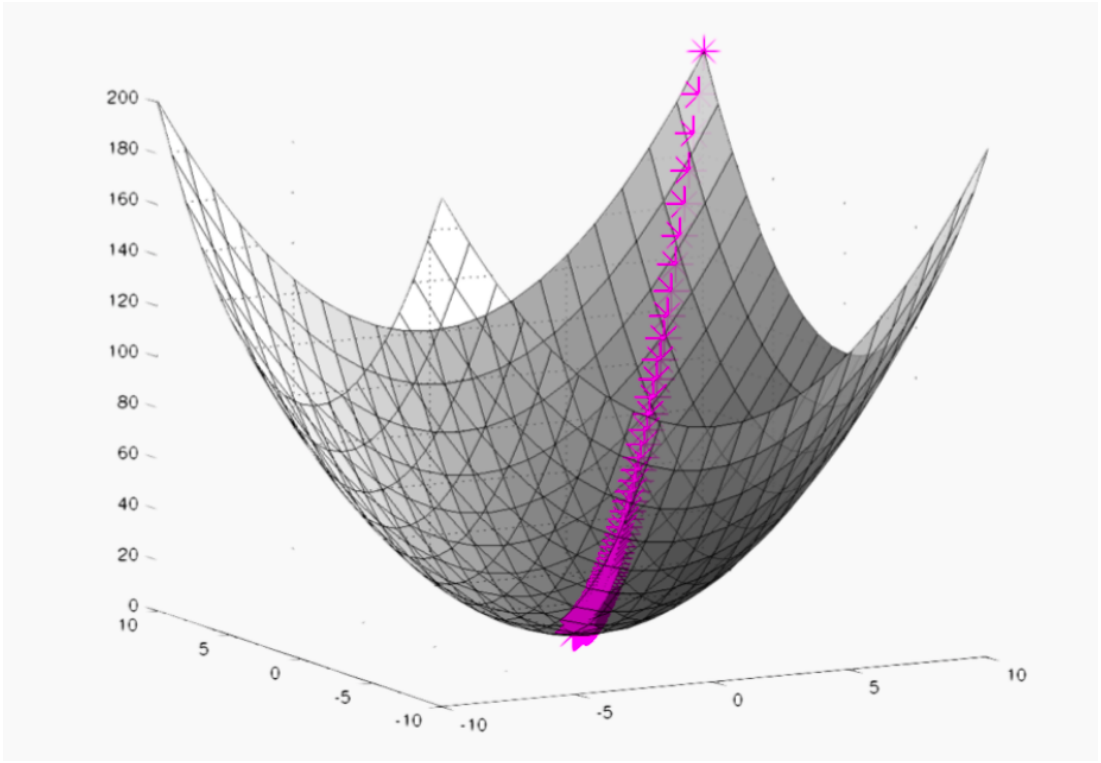
Why does this work? Recall, that as a vector, the gradient at at point gives the direction for the greatest possible rate of increase.



Gradient Descent Algorithm

Subtracting a λ multiple of the gradient from x, moves x in the ***opposite*** direction of the gradient (hence towards the steepest decline) by a step of size λ.

If f is convex, and we keep taking steps descending on the graph of f, we will eventually reach the minimum.



Interpreting Gradient Boosting from Gradient Descent

Often in regression, our objective is to minimize the MSE

$$\text{MSE}(\hat{y}_1, \ldots, \hat{y}_N) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Treating this as an optimization problem, we can try to directly minimize the MSE with respect to the predictions

$$\begin{aligned} \nabla \text{MSE} &= \left[\frac{\partial \text{MSE}}{\partial \hat{y}_1}, \ldots, \frac{\partial \text{MSE}}{\partial \hat{y}_N} \right] \\ &= -2 \left[y_1 - \hat{y}_1, \ldots, y_N - \hat{y}_N \right] \\ &= -2 \left[r_1, \ldots, r_N \right] \end{aligned}$$

The update step for gradient descent would look like

$$\hat{y}_n \leftarrow \hat{y}_n + \lambda r_n, \quad n = 1, \ldots, N$$

Gradient Boosting as Gradient Descent

There are two reasons why minimizing the MSE with respect to $\hat{y}_n$'s is not interesting:

- We know where the minimum MSE occurs: $\hat{y}_n = y_n$, for every n.
- Learning sequences of predictions $\hat{y}_n^1, \ldots, \hat{y}_n^i, \ldots$, does not produce a model. The predictions in the sequences do not depend on the predictors!

Gradient Boosting as Gradient Descent

The solution is to change the update step in gradient descent. Instead of using the gradient - the residuals - we use an ***approximation*** of the gradient that depends on the predictors:

$$\hat{y} \leftarrow \hat{y}_n + \lambda\, \hat{r}_n(x_n), \qquad n = 1, \ldots, N$$

In gradient boosting, we use a simple model to approximate the residuals, $\hat{r}_n(x_n)$, in each iteration.

Motto: gradient boosting is a form of gradient descent with the MSE as the objective function.

Technical note: note that gradient boosting is descending in a space of models or functions relating x_n to y_n!

Gradient Boosting as Gradient Descent

Under ideal conditions, gradient descent iteratively approximates and converges to the optimum.

When do we terminate gradient descent?

- We can limit the number of iterations in the descent. But for an arbitrary choice of maximum iterations, we cannot guarantee that we are sufficiently close to the optimum in the end.
- If the descent is stopped when the updates are sufficiently small (e.g. the residuals of T are small), we encounter a new problem: the algorithm may never terminate!

Both problems have to do with the magnitude of the learning rate, λ.

Choice of Learning Rate λ

Choosing λ:

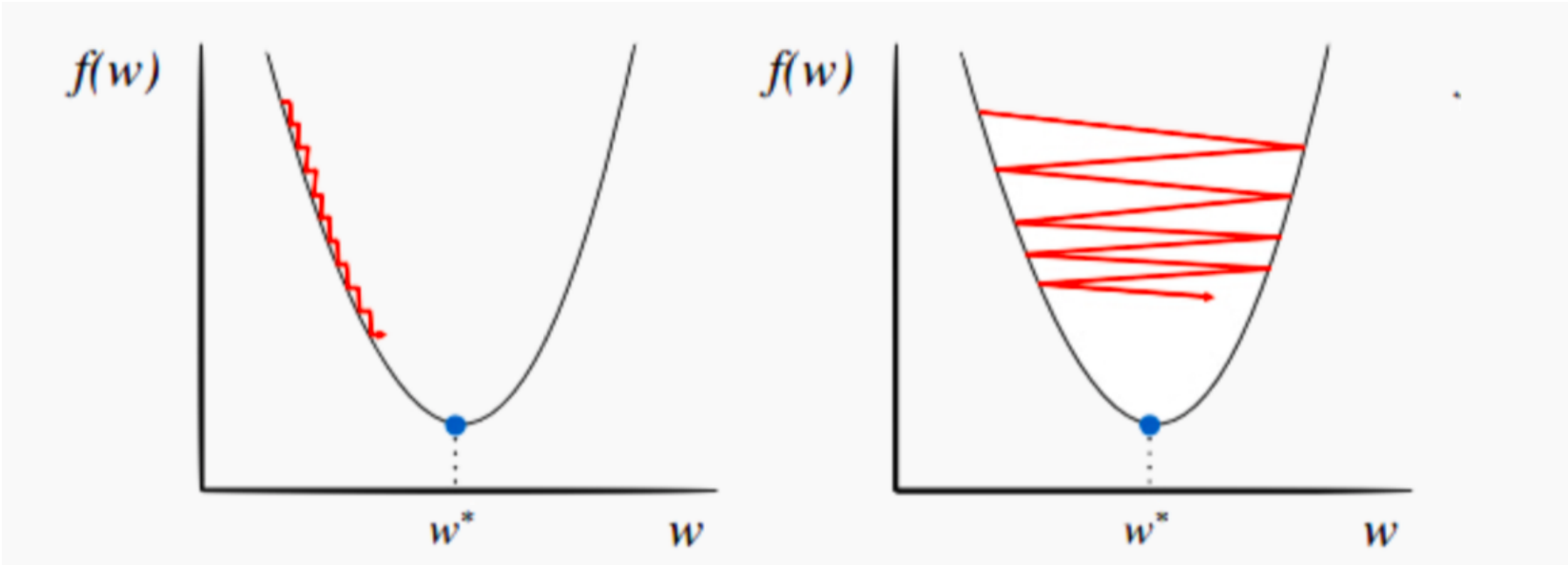
- If λ is a constant, then it should be tuned through cross validation.
- For better results, use a variable λ. That is, let the value of λ depend on the gradient

$$\lambda = h(\|\nabla f(x)\|),$$

where $\|\nabla f(x)\|$ is the magnitude of the gradient, $\nabla f(x)$. So

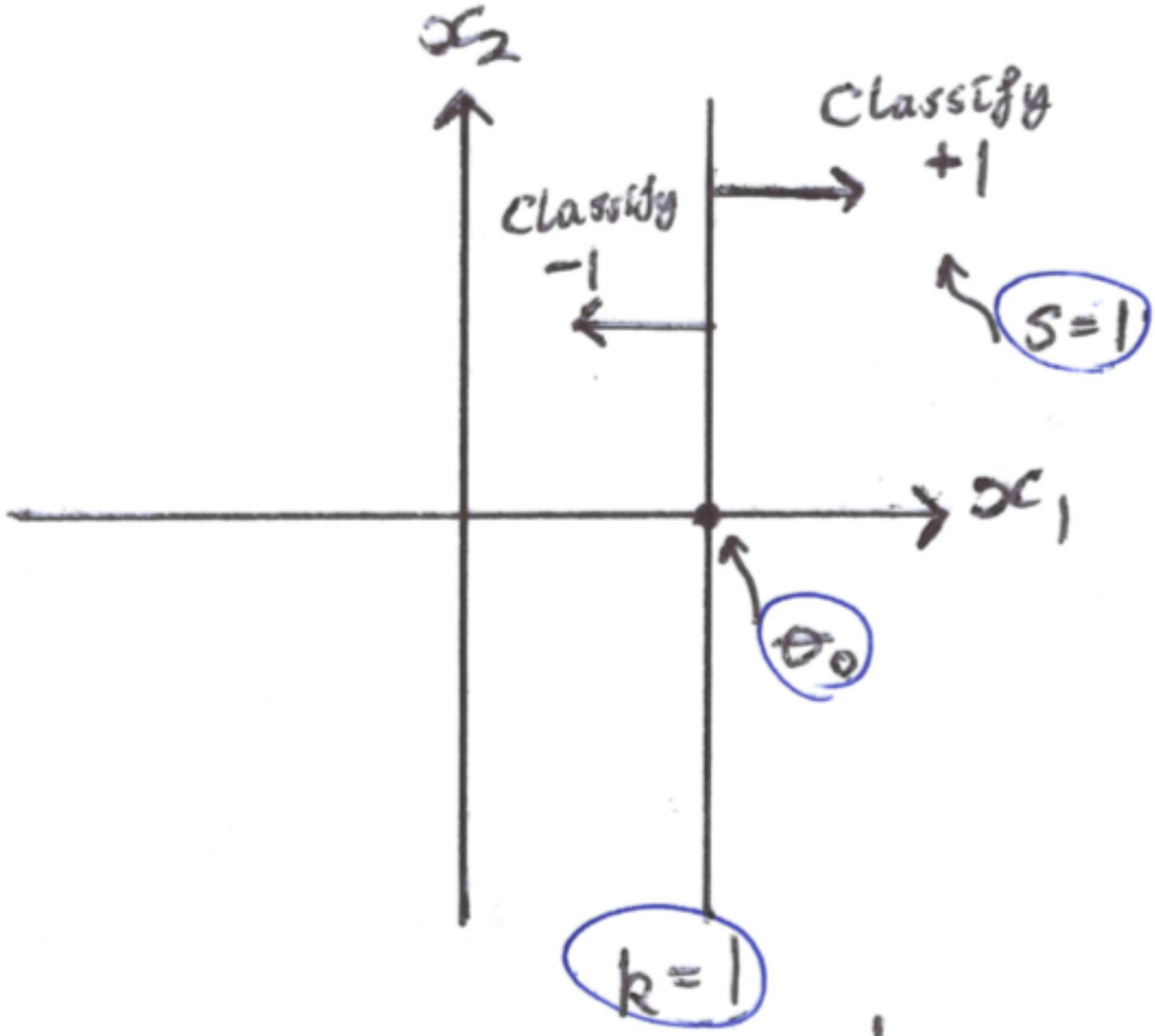
- around the optimum, when the gradient is small, λ should be small
- far from the optimum, when the gradient is large, λ should be larger

Results from Different Learning Rates

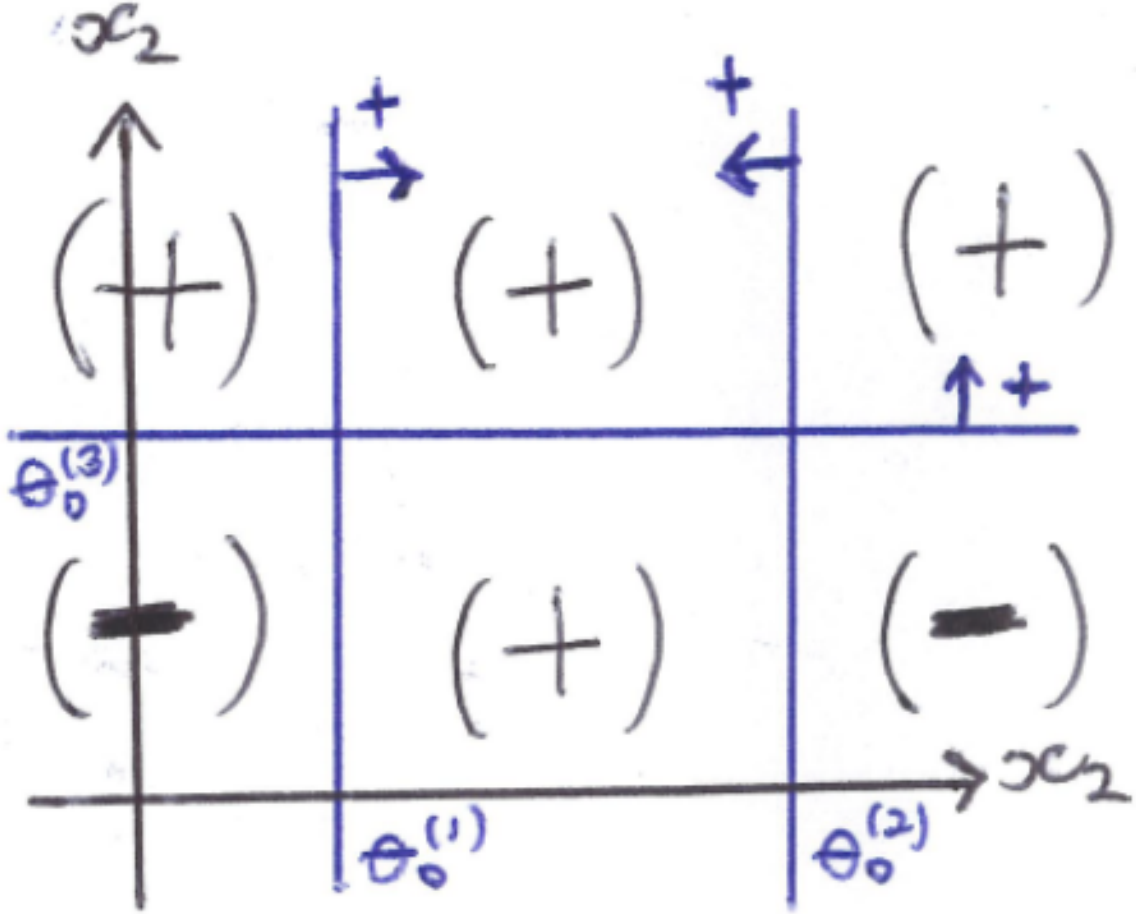


Adaboost

Example: Decision Stump



Example: Decision Tree



Motivation of AdaBoost: from Gradient Descent

Using the language of gradient descent also allow us to connect gradient boosting for regression to a boosting algorithm often used for classification, AdaBoost.

In classification, we *typically* want to minimize the classification error:

$$\text{Error} = \frac{1}{N}\sum_{n=1}^{N} \mathbb{1}(y_n \neq \hat{y}_n), \quad \mathbb{1}(y_n \neq \hat{y}_n) = \begin{cases} 0, & y_n = \hat{y}_n \\ 1, & y_n \neq \hat{y}_n \end{cases}$$

Naively, we can try to minimize Error via gradient descent, just like we did for MSE in gradient boosting.

The challenge is the loss function is not differentiable.

Motivation of AdaBoost

Our solution: we replace the Error function with a differentiable function that is a good indicator of classification error.

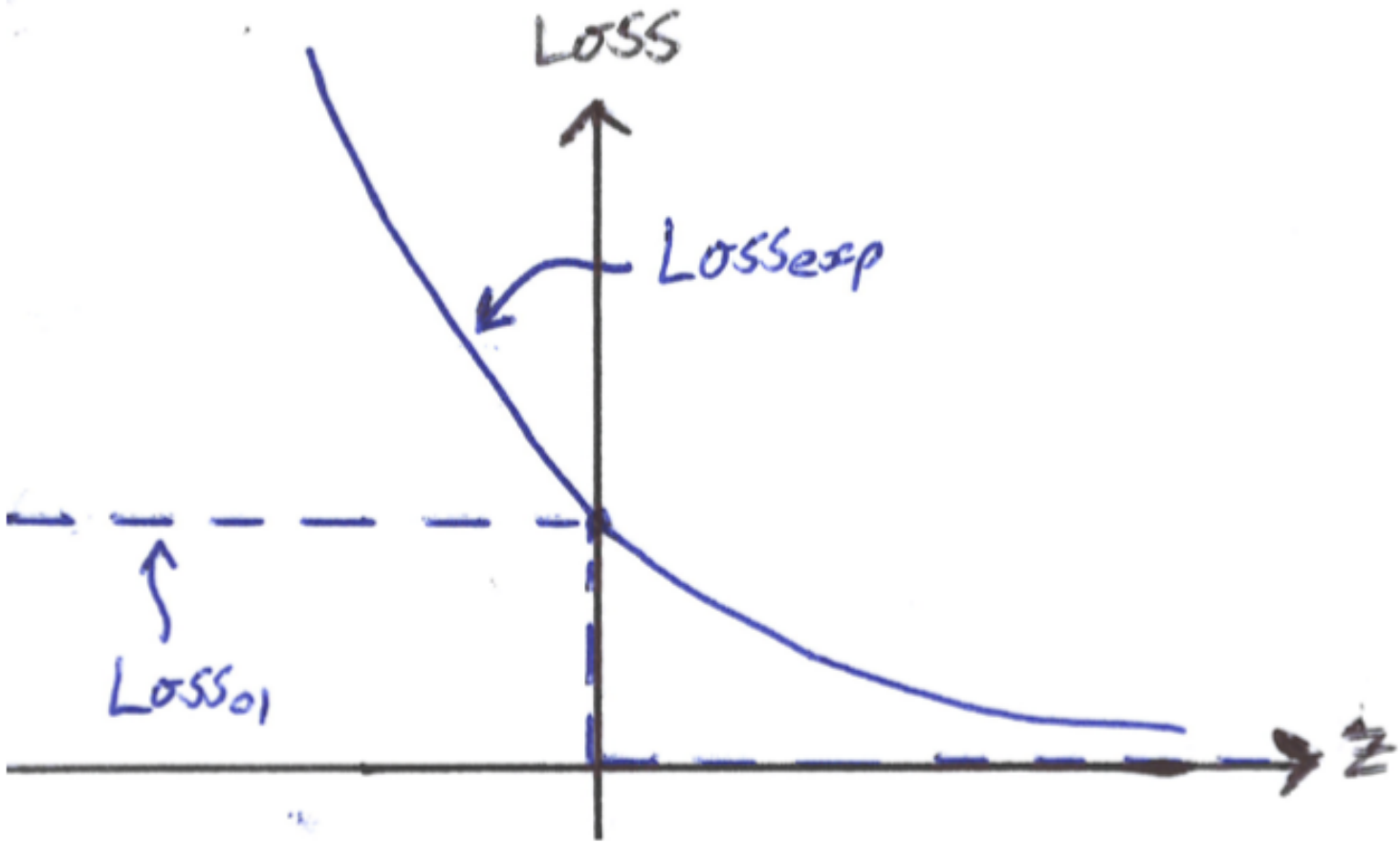
The function we choose is called ***exponential loss***

$$\text{ExpLoss} = \frac{1}{N}\sum_{n=1}^{N}\exp\left(-y_n\hat{y}_n\right),\ y_n \in \{-1, 1\}$$

Exponential loss is differentiable with respect to $\hat{y}_n$ and it is an upper bound of Error.

Analysis of Exponential Loss Function

This is an upper bound to the $0-1$ loss function $\frac{1}{N}\sum_{n=1}^{N} 1\{y_n \neq \hat{y}_n\}$.



Gradient Descent with Exponential Loss

We first compute the gradient for ExpLoss:

$$\nabla \text{Exp} = [-y_1 \exp(-y_1 \hat{y}_1), \ldots, -y_N \exp(-y_N \hat{y}_N)]$$

It's easier to decompose each $y_n \exp(-y_n \hat{y}_n)$ as $w_n y_n$, where $w_n = \exp(-y_n \hat{y}_n)$.

This way, we see that the gradient is just a re-weighting applied the target values

$$\nabla \text{Exp} = [-w_1 y_1, \ldots, -w_N y_N]$$

Notice that when $y_n = \hat{y}_n$, the weight w_n is small; when $y_n \neq \hat{y}_n$, the weight is larger.

Gradient Descent with Exponential Loss

The update step in the gradient descent is

$$\hat{y}_n \leftarrow \hat{y}_n + \lambda w_n y_n, \qquad n = 1, \ldots, N$$

Just like in gradient boosting, we approximate the gradient, $\lambda w_n y_n$ with a simple model, $T^{(i)}$, that depends on x_n.

This means training $T^{(i)}$ on a re-weighted set of target values,

$$\{(x_1, w_1 y_1), \ldots, (x_N, w_N y_N)\}$$

That is, gradient descent with exponential loss means iteratively training simple models that ***focuses on the points misclassified by the previous model.***

AdaBoost Algorithm

With a minor adjustment to the exponential loss function, we have the algorithm for gradient descent:

1. Choose an initial distribution over the training data, $w_n = 1/N$.
2. At the i^{th} step, fit a simple classifier $T^{(i)}$ on weighted training data
$$\{(x_1, w_1 y_1), \ldots, (x_N, w_N y_N)\}$$
3. Update the weights:
$$w_n \leftarrow \frac{w_n \exp(-\lambda^{(i)} y_n T^{(i)}(x_n))}{Z}$$
where Z is the normalizing constant for the collection of updated weights
4. Update T: $T \leftarrow T + \lambda^{(i)} T^{(i)}$

where λ is the learning rate.

Choice of Learning Rate λ

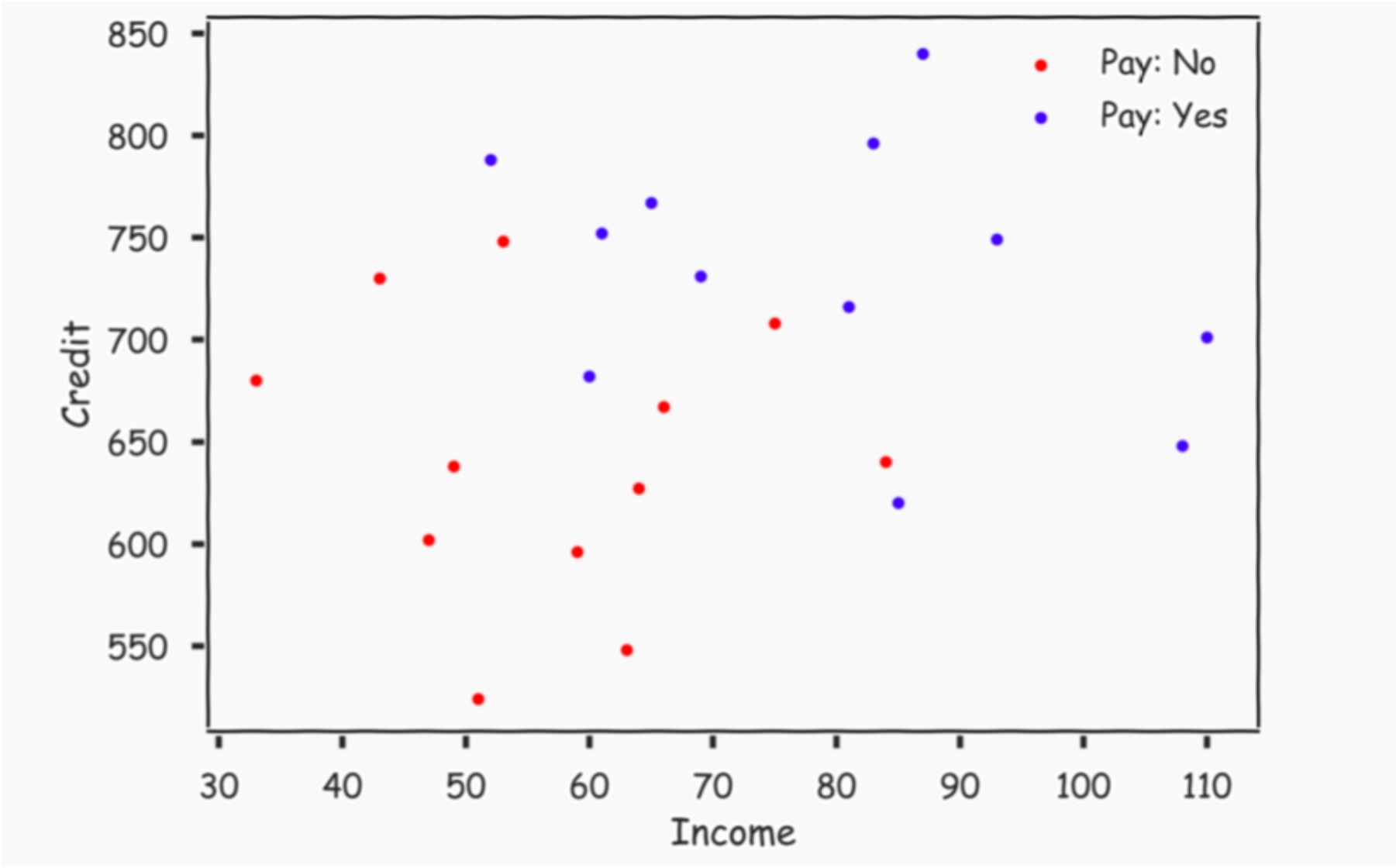
Unlike in the case of gradient boosting for regression, we can analytically solve for the optimal learning rate for AdaBoost, by optimizing:

$$\operatorname*{argmin}_{\lambda} \frac{1}{N} \sum_{n=1}^{N} \exp\left[-y_n(T + \lambda^{(i)} T^{(i)}(x_n))\right]$$

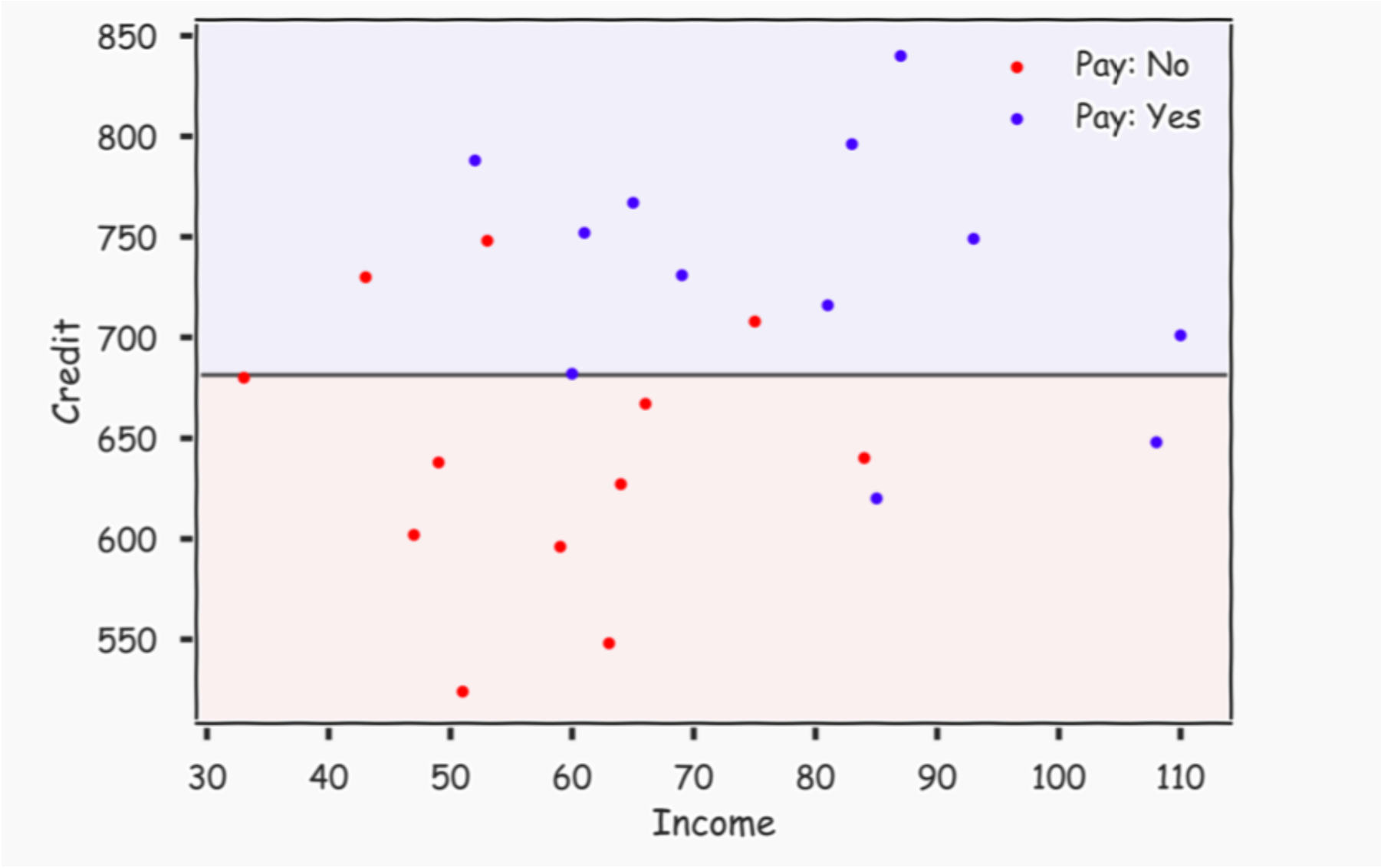
Doing so, we get that

$$\lambda^{(i)} = \frac{1}{2} \ln \frac{1-\epsilon}{\epsilon}, \quad \epsilon = \sum_{n=1}^{N} w_n \mathbb{1}(y_n \neq T^{(i)}(x_n))$$

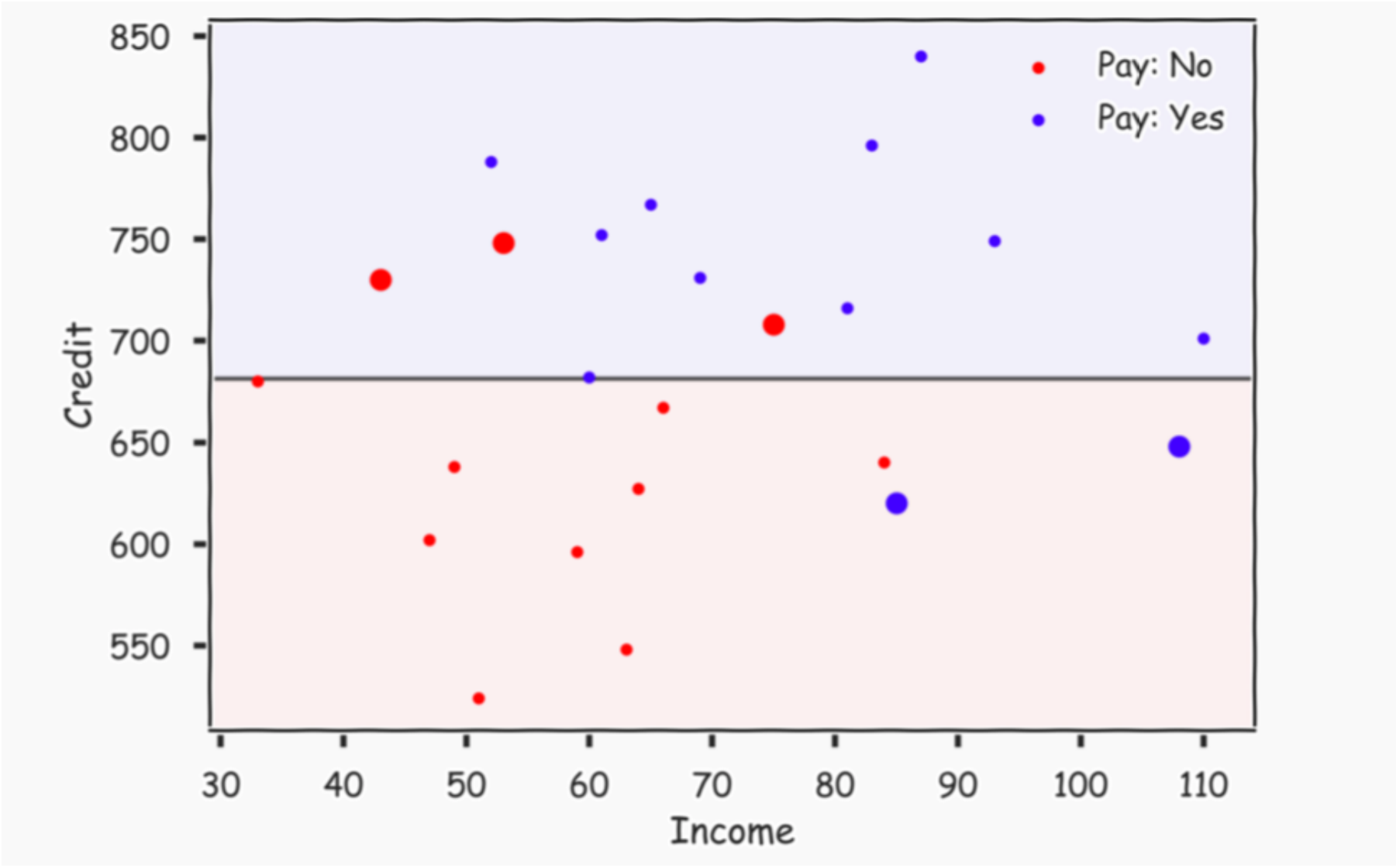
Example: AdaBoost, original data



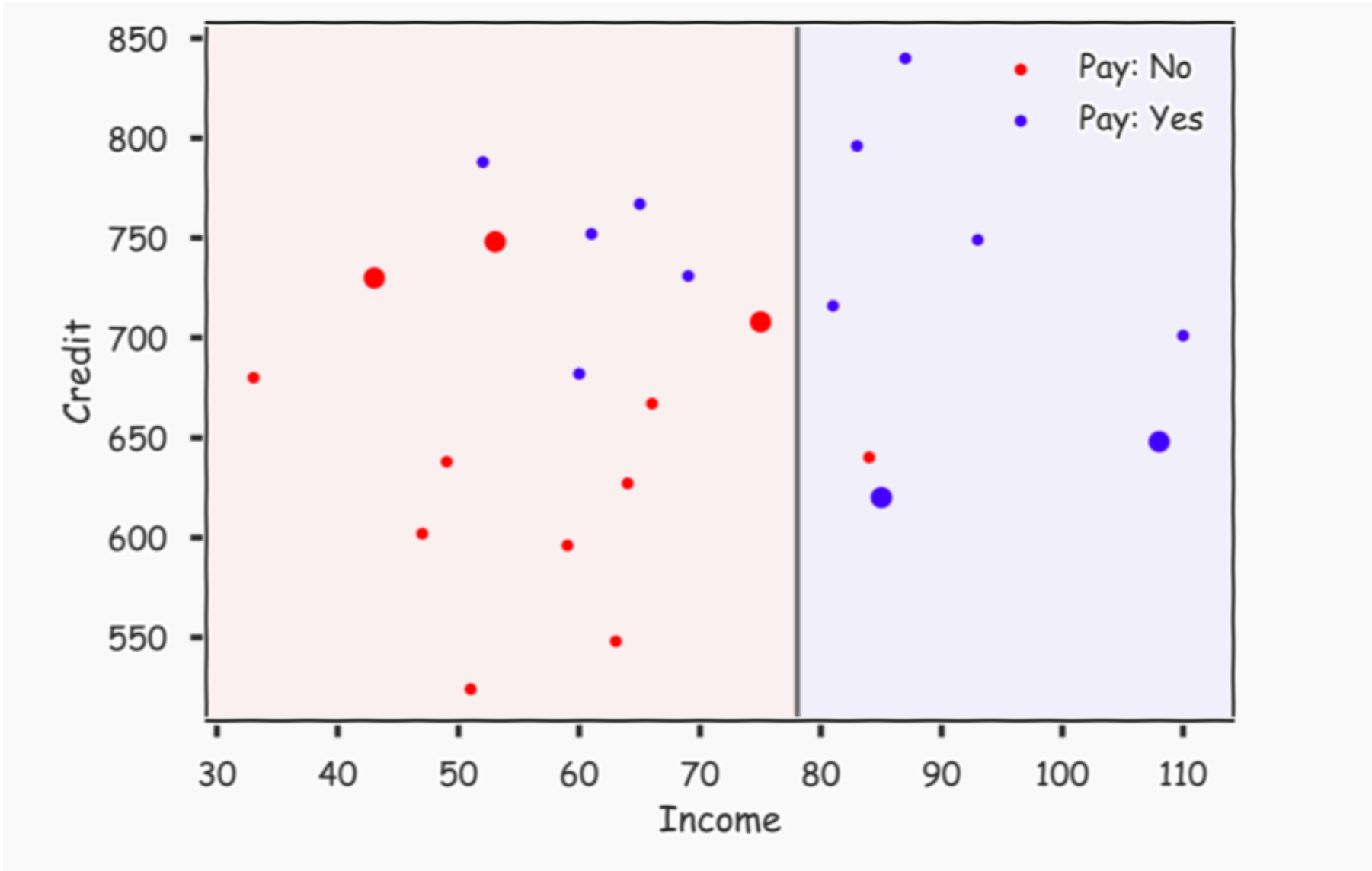
Example: AdaBoost, simple decision tree



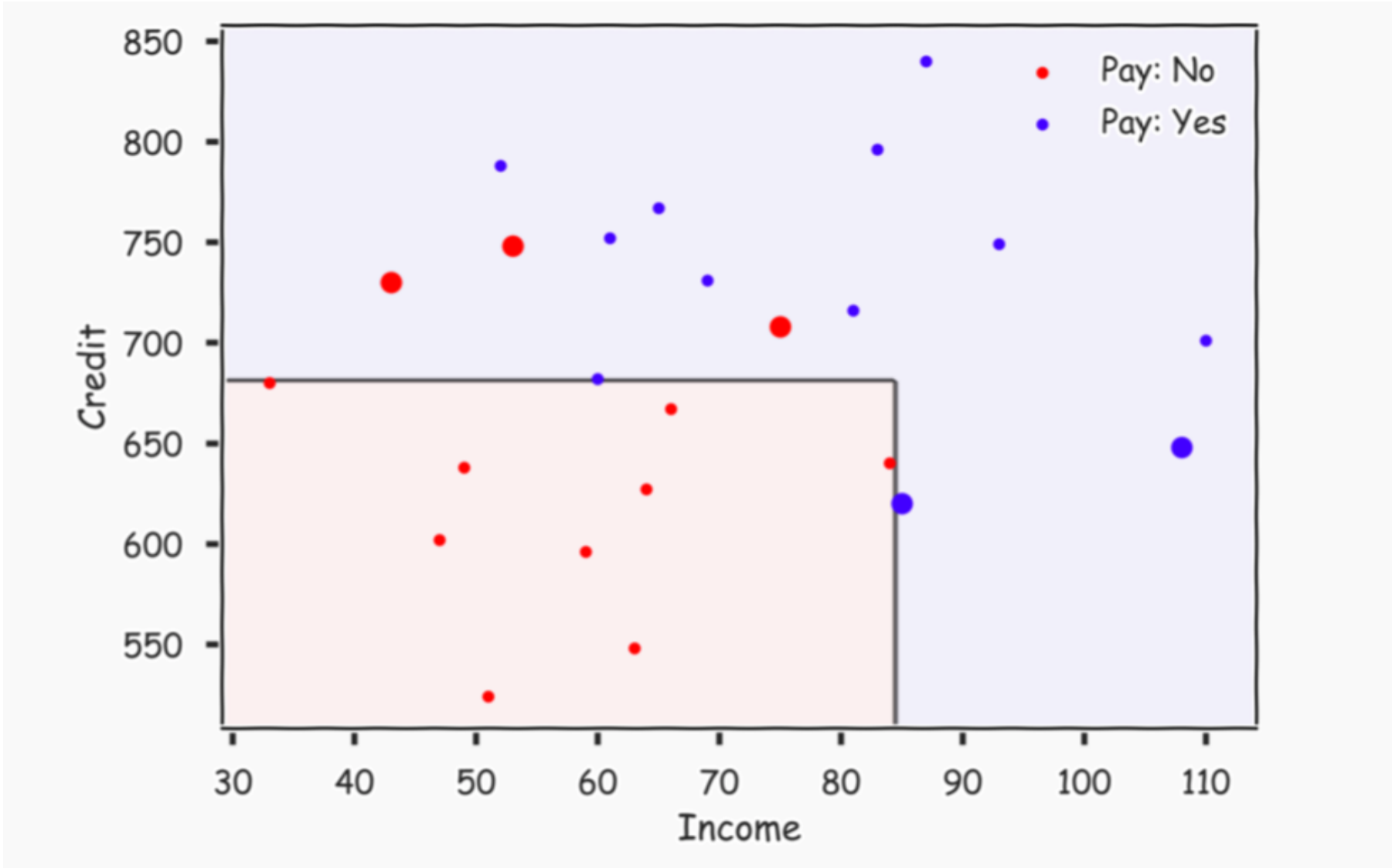
Example: AdaBoost, errors with higher weights



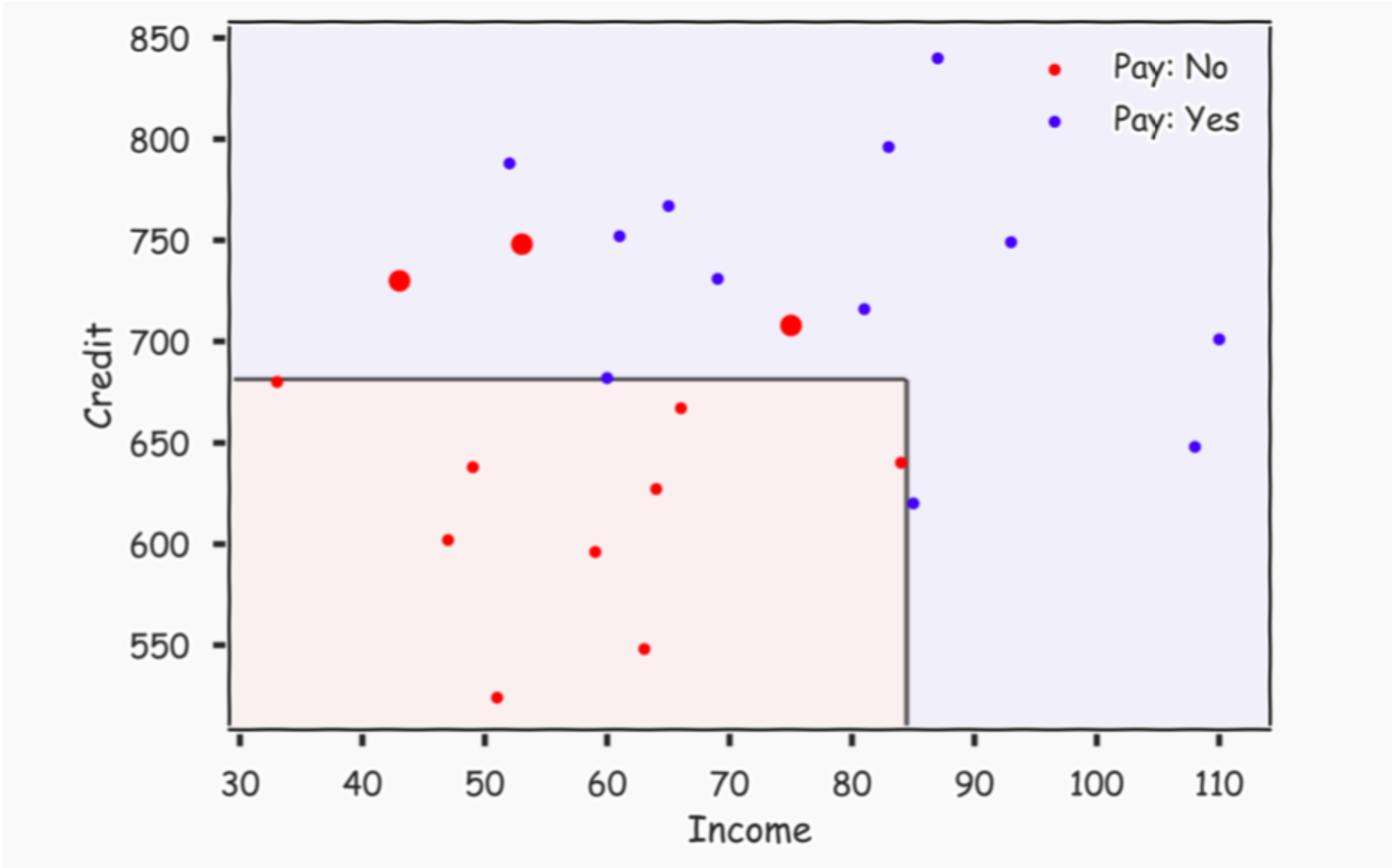
Example: AdaBoost, 2nd tree with weighted points



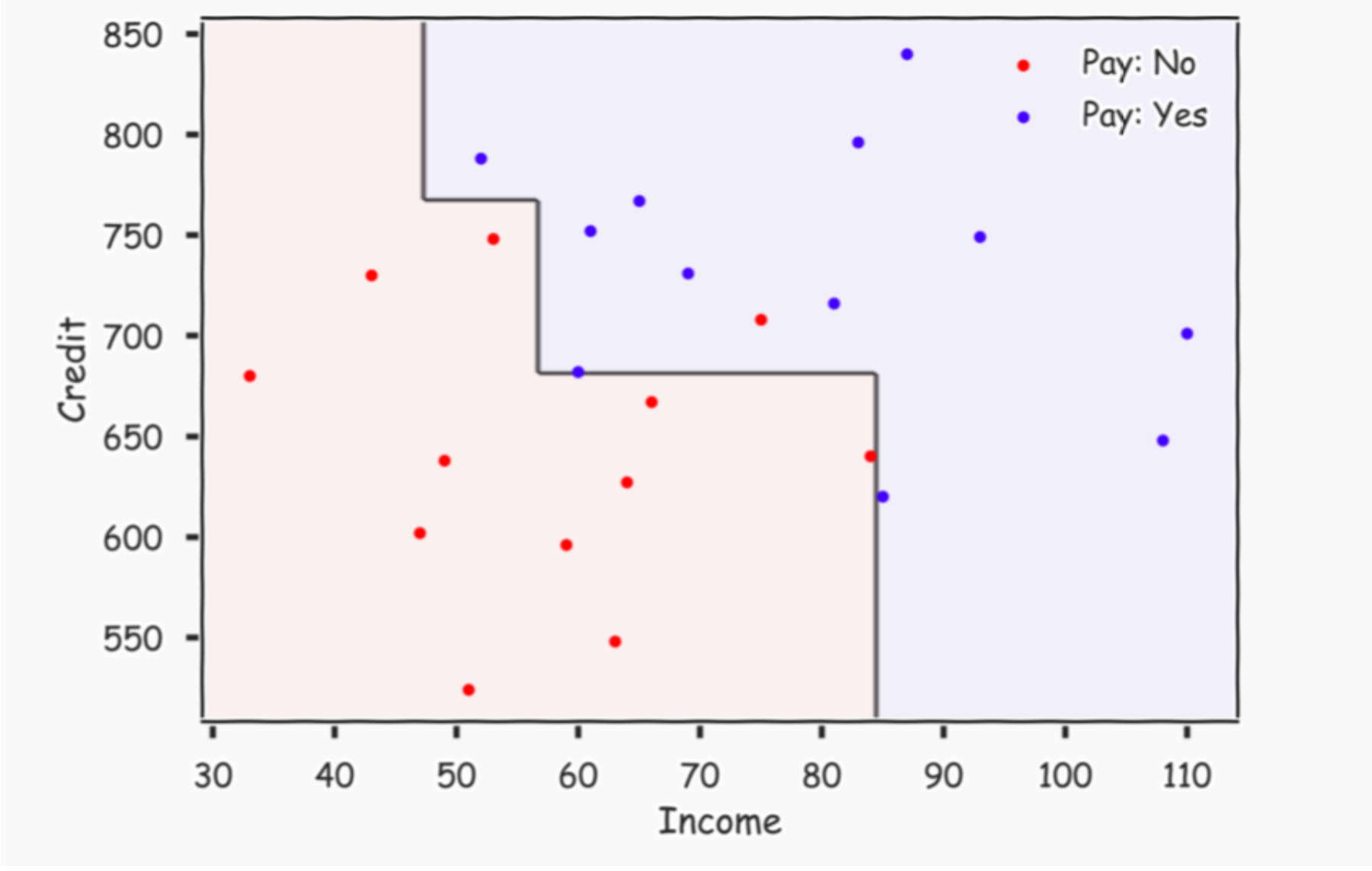
Example: AdaBoost, combine trees



Example: AdaBoost, reweight points



Example: AdaBoost



Re-study AdaBoost starting from Decision Tree

Simple Classifier: Decision Tree

- There are many possible choices for the "simple classifier" that we seek to take combinations of. We will focus on arguably the simplest class of all – decision stumps:

$$h(\mathbf{x}; \boldsymbol{\theta}) = \text{sign}(s(x_k - \theta_0)),$$

where $\boldsymbol{\theta} = \{s, k, \theta_0\}$. (*Note:* Don't confuse this with $\boldsymbol{\theta} \in \mathbb{R}^d$ from previous lectures. A consistent way of thinking about the two is "$\boldsymbol{\theta}$ is a vector of parameters", but here the parameters are quite different.)

  - k = index of the (only) feature the decision is made based on
  - s = sign (to allow both combinations of $+$ on one side and $-$ on the other)
  - θ_0 = offset/threshold (classify to $+$ if above this value and $-$ if below, or vice versa)

AdaBoost Algorithm

- Input: Data set $\mathcal{D} = \{(\mathbf{x}_t, y_t)\}_{t=1}^n$ with $\mathbf{x}_t \in \mathbb{R}^d$, $y_t \in \{-1, 1\}$, number of iterations[5] M
- Steps:

1. Initialize weights $w_0(t) = \frac{1}{n}$ for $t = 1, \ldots, n$
2. For $m = 1, \ldots, M$, do the following:
   (a) Choose the next base learner $h(\cdot; \hat{\boldsymbol{\theta}}_m)$ as follows:

$$\hat{\boldsymbol{\theta}}_m = \arg\min_{\boldsymbol{\theta}} \sum_{t:\, y_t \neq h(\mathbf{x}_t; \boldsymbol{\theta})} w_{m-1}(t) \tag{3}$$

   (b) Set $\hat{\alpha}_m = \frac{1}{2} \log \frac{1-\hat{\epsilon}_m}{\hat{\epsilon}_m}$, where $\hat{\epsilon}_m = \sum_{t=1}^n w_{m-1}(t) \mathbb{1}\{y_t \neq h(\mathbf{x}_t; \hat{\boldsymbol{\theta}}_m)\}$ is the minimal value attained in (3).
   (c) Update the weights:

$$w_m(t) = \frac{1}{Z_m} w_{m-1}(t) e^{-y_t h(\mathbf{x}_t; \hat{\boldsymbol{\theta}}_m) \hat{\alpha}_m} \tag{4}$$

   for each $t = 1, \ldots, n$, where Z_m is defined so that the weights sum to one:

$$Z_m = \sum_{t=1}^n w_{m-1}(t) e^{-y_t h(\mathbf{x}_t; \hat{\boldsymbol{\theta}}_m) \hat{\alpha}_m}. \tag{5}$$

3. Output: Classifier $f_M(\mathbf{x}) = \sum_{m=1}^M \hat{\alpha}_m h(\mathbf{x}; \hat{\boldsymbol{\theta}}_m)$

Analysis on Base Learners

Intuition behind the base learner (3):

- The quantity

$$\epsilon_m = \sum_{t\,:\,y_t \neq h(\mathbf{x}_t;\boldsymbol{\theta})} w_{m-1}(t)$$

  is called the *weighted training error*. Step (3) is choosing a base learner that "classifies best" when certain samples are treated as more important than others (as dictated by $w_{m-1}(\cdot)$).

- Sometimes you might see the selection rule (3) written as

$$\hat{\boldsymbol{\theta}}_m = \arg\min_{\boldsymbol{\theta}} \sum_{t=1}^{n} w_{m-1}(t) \cdot \big(- y_t h(\mathbf{x}_t;\boldsymbol{\theta})\big) \tag{6}$$

  To see that the two are equivalent, first note that since both y_t and $h(\cdot)$ take values in ± 1, we have

$$- y_t h(\mathbf{x}_t;\boldsymbol{\theta}) = 2\mathbb{1}\{y_t \neq h(\mathbf{x}_t;\boldsymbol{\theta})\} - 1 \tag{7}$$

  (just check the cases $y_t = h$ and $y_t \neq h$ separately). Summing over the samples $t = 1, \ldots, n$ gives

$$\sum_{t=1}^{n} w_{m-1}(t)\big(- y_t h(\mathbf{x}_t;\boldsymbol{\theta})\big) = 2\epsilon_m - 1,$$

Analysis of the Base Learners

- Claim. The updated weights $w_m(t)$ are such that

$$\sum_{t=1}^{n} w_m(t) \mathbb{1}\{y_t \neq h(\mathbf{x}_t; \hat{\boldsymbol{\theta}}_m)\} = \frac{1}{2}.$$

  Hence, the base learner chosen at iteration m is "useless" (i.e., does not outperform random guessing) on the next iteration.

  - Consequence: The same base learner will never be chosen on two consecutive rounds.

Analysis on Weights Updating

Intuition behind the update (4):

- Again exploiting the fact that $-y_t h(\mathbf{x}_t; \boldsymbol{\theta})$ only takes values $+1$ or -1, we can rewrite (4) as

$$w_m(t) = \frac{1}{Z_m} w_{m-1}(t) \times \begin{cases} e^{\hat{\alpha}_m} & y_t \neq h(\mathbf{x}_t; \hat{\boldsymbol{\theta}}_m) \\ e^{-\hat{\alpha}_m} & y_t = h(\mathbf{x}_t; \hat{\boldsymbol{\theta}}_m). \end{cases} \tag{8}$$

- Therefore we are doing the following:
  - If the newly chosen base learner classifies $\mathbf{x}_t$ incorrectly, increase the t-th weight
  - If the newly chosen base learner classifies $\mathbf{x}_t$ correctly, decrease the t-th weight.

  In other words, future decisions put more importance on inputs that were previously classified wrongly. (Analogy: Teacher places more teaching emphasis on topics that students scored poorly on)
- The update rule is a form of *multiplicative weights update*, which is a widespread tool that was developed independently in multiple research communities.

Training Error

- **Theorem.** After M iterations, the training error of AdaBoost satisfies

$$\frac{1}{n}\sum_{t=1}^{n}\mathbb{1}\{y_t f_M(\mathbf{x}_t) \le 0\} \le \exp\left(-2\sum_{m=1}^{M}\left(\frac{1}{2}-\hat{\epsilon}_m\right)^2\right).$$

  In particular, if $\hat{\epsilon}_m \le \frac{1}{2}-\gamma$ for all m and some $\gamma > 0$, then

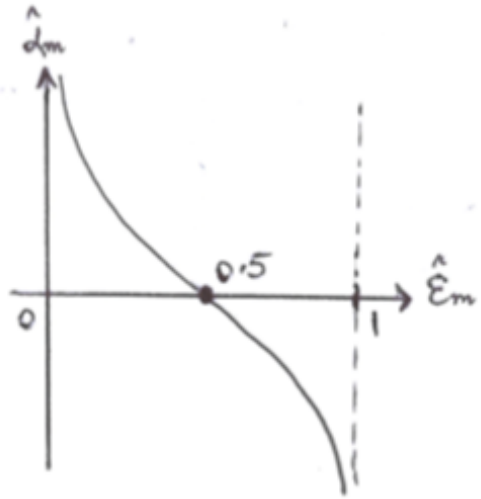
$$\frac{1}{n}\sum_{t=1}^{n}\mathbb{1}\{y_t f_M(\mathbf{x}_t) \le 0\} \le e^{-2M\gamma^2}.$$

- Since the left-hand side only takes values in $\{0, \frac{1}{n}, \dots\}$, the training error is zero for $M > \frac{\log n}{2\gamma^2}$.
  - The condition $\hat{\epsilon}_m \le \frac{1}{2}-\gamma$ can be interpreted as "we slightly beat random guessing" (since randomly guessing a label gives a 50% chance of success)

Analysis on the Learning Rate

Intuition behind the choice of $\hat{\alpha}_m$:

- The choice $\hat{\alpha}_m = \frac{1}{2}\log\frac{1-\hat{\epsilon}_m}{\hat{\epsilon}_m}$ as a function of the training error looks like the following:



- It satisfies some intuitive properties:
  - It is a decreasing function of $\hat{\epsilon}_m$ (the better the weighted training error this base learner gives, the more weight is given to it)
  - If $\hat{\epsilon}_m = 0$, then $\hat{\alpha}_m = \infty$ (if this base learner gives classifies perfectly, put all the weight on it)
  - If $\hat{\epsilon}_m = \frac{1}{2}$, then $\hat{\alpha}_m = 0$ (if this base learner gives only classifies 50/50, put no weight on it)

Example on AdaBoost Calculation

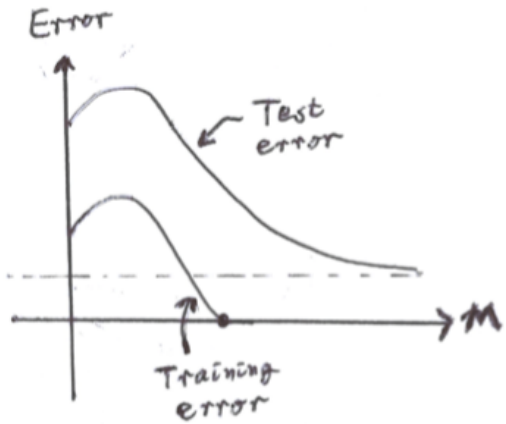
- A simple example with $n = 7$ samples:
- Initially $w_0 = \left(\frac{1}{7}, \ldots, \frac{1}{7}\right)$.
- Then $\hat{\epsilon}_1 = \frac{2}{7}$ and hence $\hat{\alpha}_1 = \frac{1}{2} \log \frac{5}{2}$
- The updated weight is

$$w_1(t) = \frac{1}{Z_1} \begin{cases} \frac{1}{7} e^{-\frac{1}{2} \log \frac{5}{2}} & \text{correct samples} \\ \frac{1}{7} e^{\frac{1}{2} \log \frac{5}{2}} & \text{incorrect samples,} \end{cases}$$

  and a bit of analysis shows that after choosing Z_1 to satisfy $\sum_t w_1(t) = 1$, the weights simplify to $w_1(t) \in \left\{\frac{1}{10}, \frac{1}{4}\right\}$ (five values of $\frac{1}{10}$, two values of $\frac{1}{4}$)

Test Error

- For typical learning algorithms, making the training error too small makes the *test error* (i.e., the error rate on *unseen* data) large.
  - This is known as *overfitting*, and will be explored more in the coming lectures.
- For boosting, we typically observe the following surprising phenomenon:



  - Test error continues decreasing even after getting zero training error!

Test Error

- Let's try to get an intuitive understanding of this. Define

$$\tilde{f}_M(\mathbf{x}) = \frac{\sum_{m=1}^M \hat{\alpha}_m h(\mathbf{x}; \hat{\boldsymbol{\theta}}_m)}{\sum_{m=1}^M \hat{\alpha}_m} \in [-1, 1], \qquad \text{margin}(t) = y_t \tilde{f}_M(\mathbf{x}_t) \in [-1, 1].$$

  We have $\text{margin}(t) > 0$ if and only if $\mathbf{x}_t$ is classified correctly. But the closer $\text{margin}(t)$ is to one, the "further away" the classifier is from incorrectly classifying it.

- Now define the following stricter notion of error:

$$\text{err}_n(f_M; \rho) = \frac{1}{n} \sum_{t=1}^n \mathbb{1}\{\text{margin}(t) \leq \rho\}.$$

  Note that the normal training error is simply $\text{err}_n(f_M; 0)$.

- *Implicitly, AdaBoost is minimizing* $e^{-\text{margin}(t)}$. This means that even after achieving $\text{err}_n(f_M; 0) = 0$, the algorithm continues to decrease $\text{err}_n(f_M; \rho)$ for $\rho > 0$:
  - A higher margin leads to better generalization, and boosting naturally increases the margin.