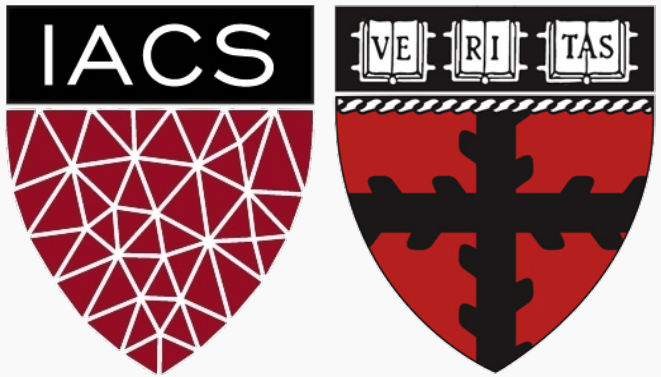
Lecture 17: Boosting

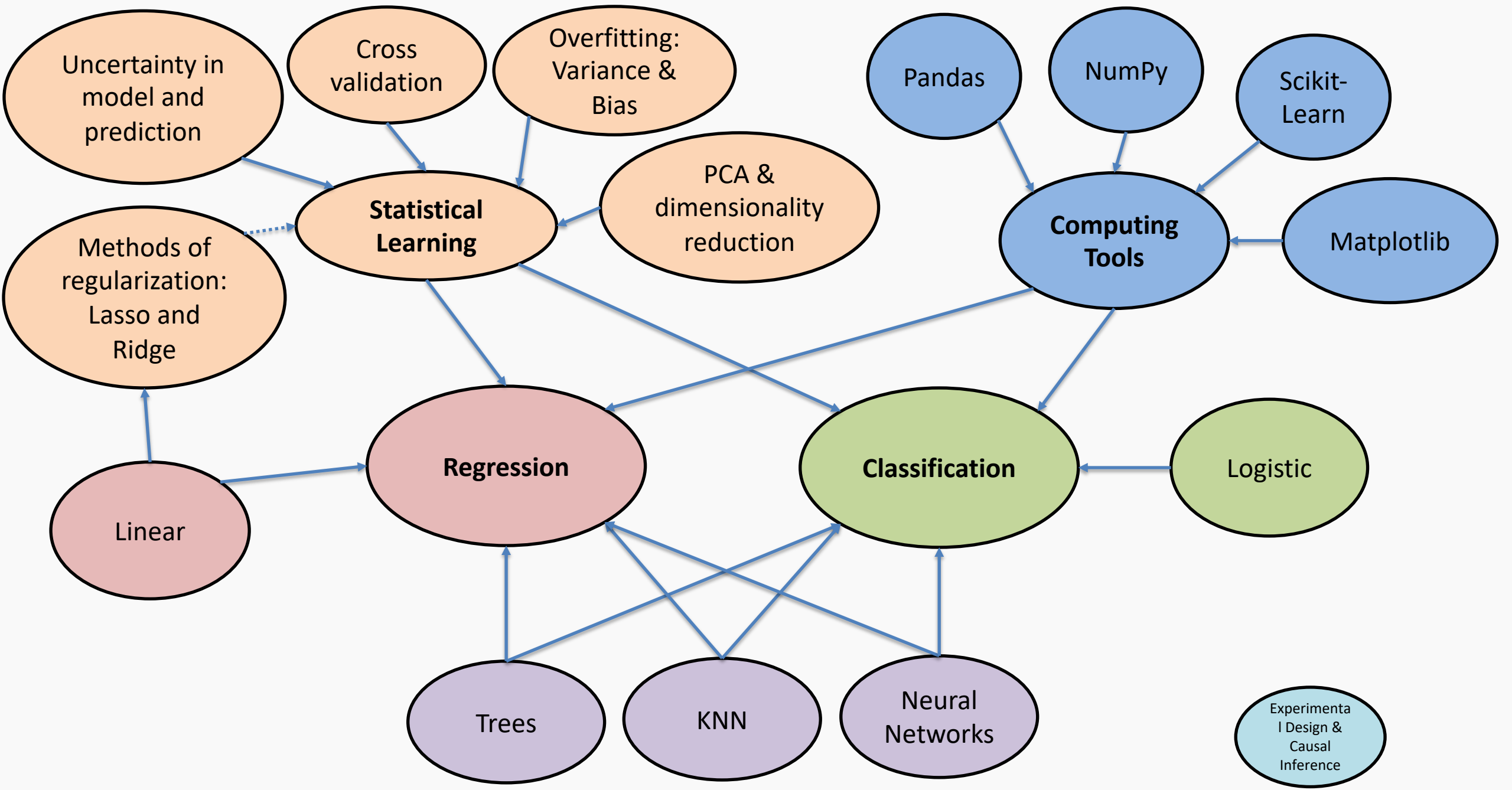
CS109A Introduction to Data Science

Pavlos Protopapas, Kevin Rader and Chris Tanner



ANNOUNCEMENTS

- Homework 5 (209) due on Wednesday 11:59 pm, Nov 6
- Homework 4 grades. They should out soon ☺


Uncertainty in model and prediction
Cross validation
Overfitting: Variance & Bias
Pandas
NumPy
Scikit-Learn
Statistical Learning
PCA & dimensionality reduction
Computing Tools
Matplotlib
Methods of regularization: Lasso and Ridge
Regression
Classification
Logistic
Linear
Trees
KNN
Neural Networks
Experimental Design & Causal Inference

Outline

- Review of Ensemble Methods
  - Finish Random Forest
- Boosting
  - Gradient Boosting
    - Set-up and intuition
    - Connection to Gradient Descent
    - The Algorithm
  - AdaBoost
- Other boosting algorithms

Bags and Forests of Trees

- Last time we examined how the short-comings of single decision tree models can be overcome by ensemble methods - making one model out of many trees.

- We focused on the problem of training large trees, these models have low bias but high variance.

- We compensated by training an ensemble of full decision trees and then averaging their predictions - thereby reducing the variance of our final model.

Bags and Forests of Trees (cont.)

Bagging:

- create an ensemble of trees, each trained on a bootstrap sample of the training set
- average the predictions.

Random forest:

- create an ensemble of trees, each trained on a bootstrap sample of the training set
- in each tree and each split, randomly select a subset of predictors, choose a predictor from this subset for splitting
- average the predictions

Note that the ensemble building aspects of both methods are embarrassingly parallel!

Tuning Random Forests

Random forest models have multiple hyper-parameters to tune:

1. the number of predictors to randomly select at each split
2. the total number of trees in the ensemble
3. the minimum leaf node size

In theory, each tree in the random forest is full, but in practice this can be computationally expensive (and added redundancies in the model), thus, imposing a minimum node size is not unusual.

Tuning Random Forests

There are standard (default) values for each of random forest hyper-parameters recommended by long time practitioners, but generally these parameters should be tuned through **OOB** (making them data and problem dependent).

e.g. number of predictors to randomly select at each split:

- $\sqrt{N_j}$ for classification
- $\frac{N}{3}$ for regression

sklearn = max_features

Using out-of-bag errors, training and cross validation can be done in a single sequence - we cease training once the out-of-bag error stabilizes

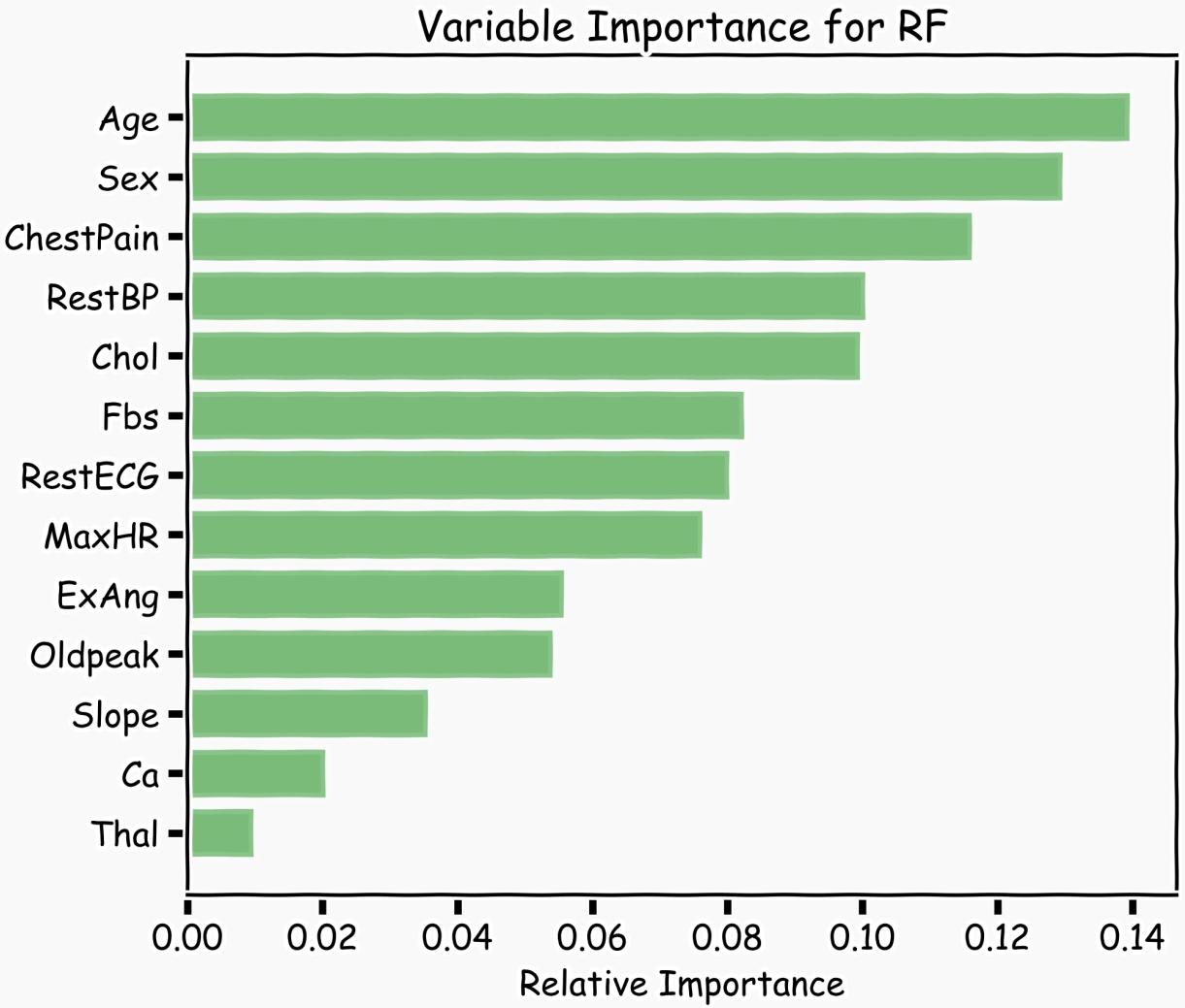
Variable Importance for RF

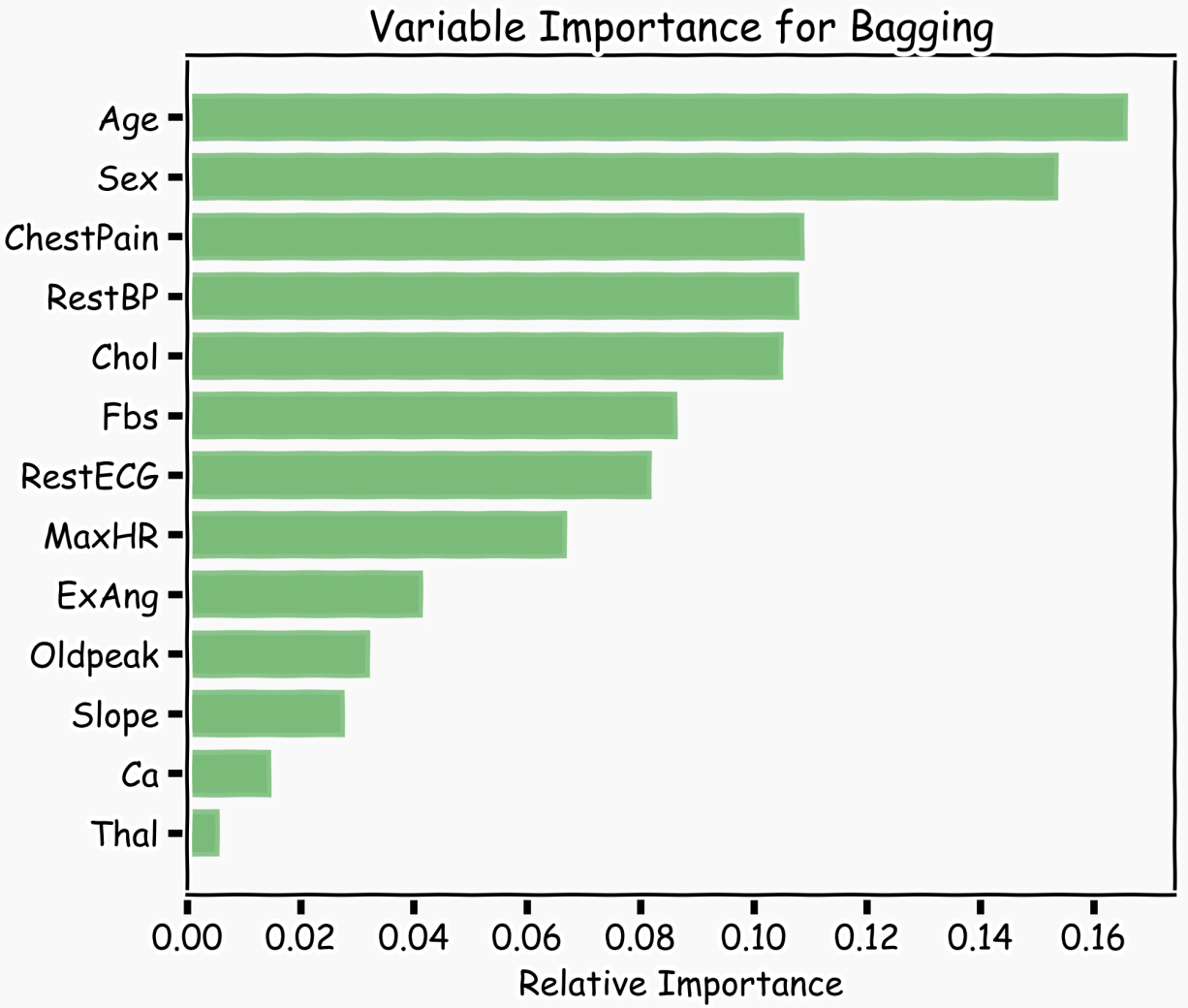
Same as with Bagging:

Calculate the total amount that the RSS (for regression) or Gini index (for classification) is decreased due to splits over a given predictor, averaged over all B trees.

Variable Importance for RF

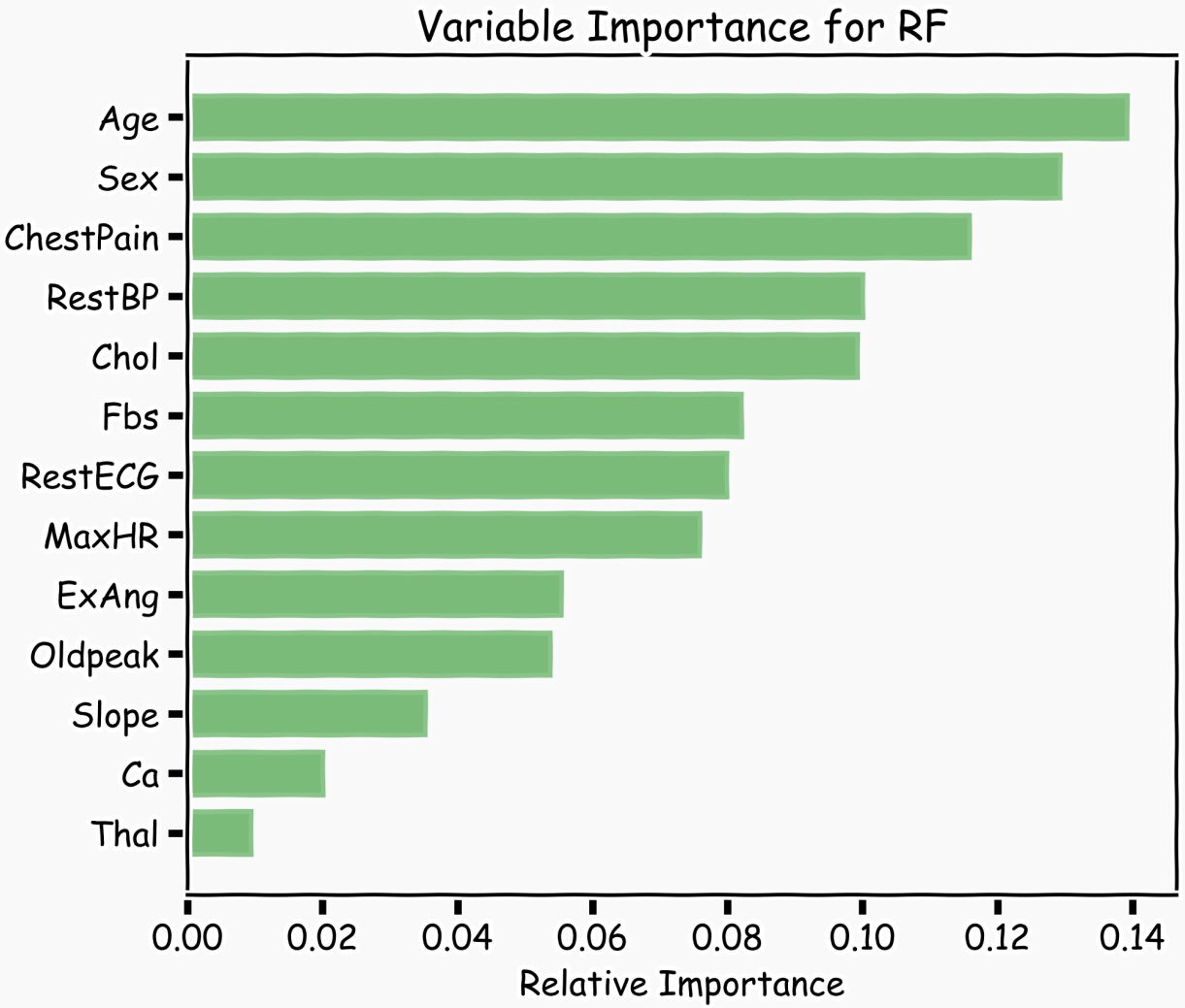
Random forest decreases VI for the most impt predictors, because of random choice of predictors - this is good because it decreases correlation of trees to each other

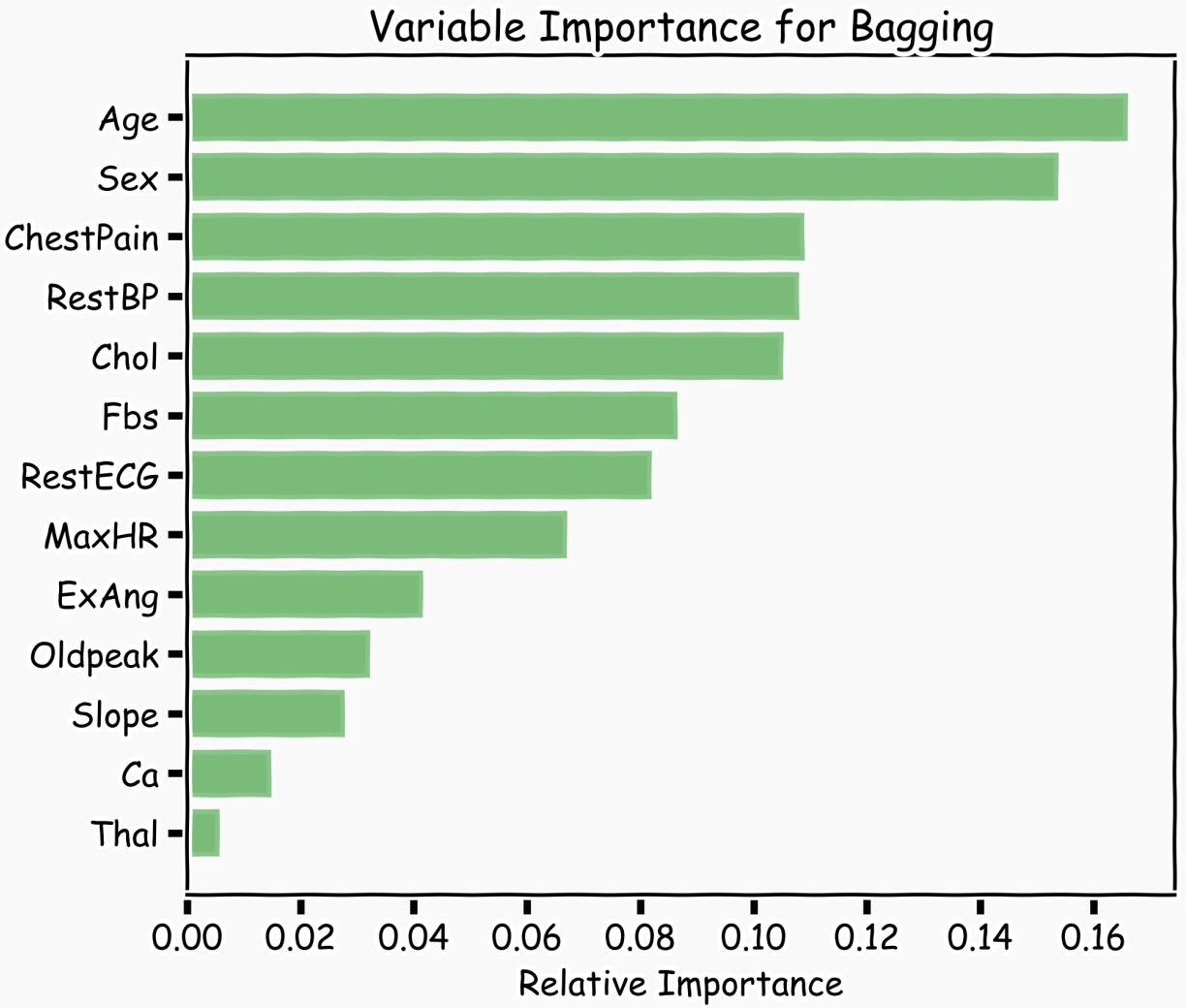




100 trees, max_depth=10

Variable Importance for RF





100 trees, max_depth=10

Final Thoughts on Random Forests

When the number of predictors is large, but the number of relevant predictors is small, random forests can perform poorly.

Question: Why?

In each split, the chances of selected a relevant predictor will be low and hence most trees in the ensemble will be weak models.

Have max features be very large at the beginning and choose optimal num of predictors with CV as needed

Final Thoughts on Random Forests (cont.)

Increasing the number of trees in the ensemble generally does not increase the risk of overfitting.

Again, by decomposing the generalization error in terms of bias and variance, we see that increasing the number of trees produces a model that is at least as robust as a single tree.

However, if the number of trees is too large, then the trees in the ensemble may become more correlated, and therefore increase the variance.

Bootstrap samples will become correlated e.g. take 2000 boostrap samples with 1000 points (you can get the same sample twice) —> this causes correlation between trees and overfitting to the data

Final Thoughts on Random Forests (cont.)

Probabilities:

- Random Forrest Classifier (and bagging) can return probabilities.
- **Question**: How?

  Instead of taking the class majority, you get probability of each class
  Average number of trees in your RF

More

- Unbalance dataset
- Weighted samples
- Categorical data
- Missing data
- Why did we reject misclassification error?

Unbalanced data sets: 99.5% red vs 0.5% blue
1. upsample blue or downsample red
2. weight samples using class weights; weight according to the ratio of the classes

Categorical data:
1. In practice, if you turn categories into ordinals (blue, red to 1, 2) that can be ok, but using ordinals you give things an order which isn't preferred
2. One-hot encoding - num predictors increase a lot, effect of a single one hot encoded column gets washed out (but this can work with small nums of categorical vars -1-2 categorical vars with 1-2 categorical types)

Missing data:
1. Imputation with means/medians, modeling/conditional probabilities
2. Trees have different ways of dealing with missing data - surrogate data approach (a-section topic)

Misclassification error:
1. Why don't we use it?
A. not differentiable (but we haven't used gradient descent to optimize so why does it matter?).
B. not convex - a lot of splits that give the same answer (a lot of local minima).
C. cannot do more than binary splits

Boosting

Motivation for Boosting

Question: Could we address the shortcomings of single decision trees models in some other way?

For example, rather than performing variance reduction on complex trees, can we decrease the bias of simple trees - make them more expressive?

Can we learn from our mistakes?

A solution to this problem, making an expressive model from simple trees, is another class of ensemble methods called ***boosting***. rather than doing random forests/bagging

Boosting Algorithms

Gradient Boosting

The key intuition behind boosting is that one can take an ensemble of simple models $\{T_h\}_{h \in H}$ and additively combine them into a single, more complex model.

Each model T_h might be a poor fit for the data, but a linear combination of the ensemble

$$T = \sum_h \lambda_h T_H$$

(coef: λ_h, tree: T_H)

can be expressive/flexible.

Question: But which models should we include in our ensemble? What should the coefficients or weights in the linear combination be?

Gradient Boosting: the algorithm

Gradient boosting is a method for iteratively building a complex regression model T by adding simple models. Each new simple model added to the ensemble compensates for the weaknesses of the current ensemble.

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. for regression in this example

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

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

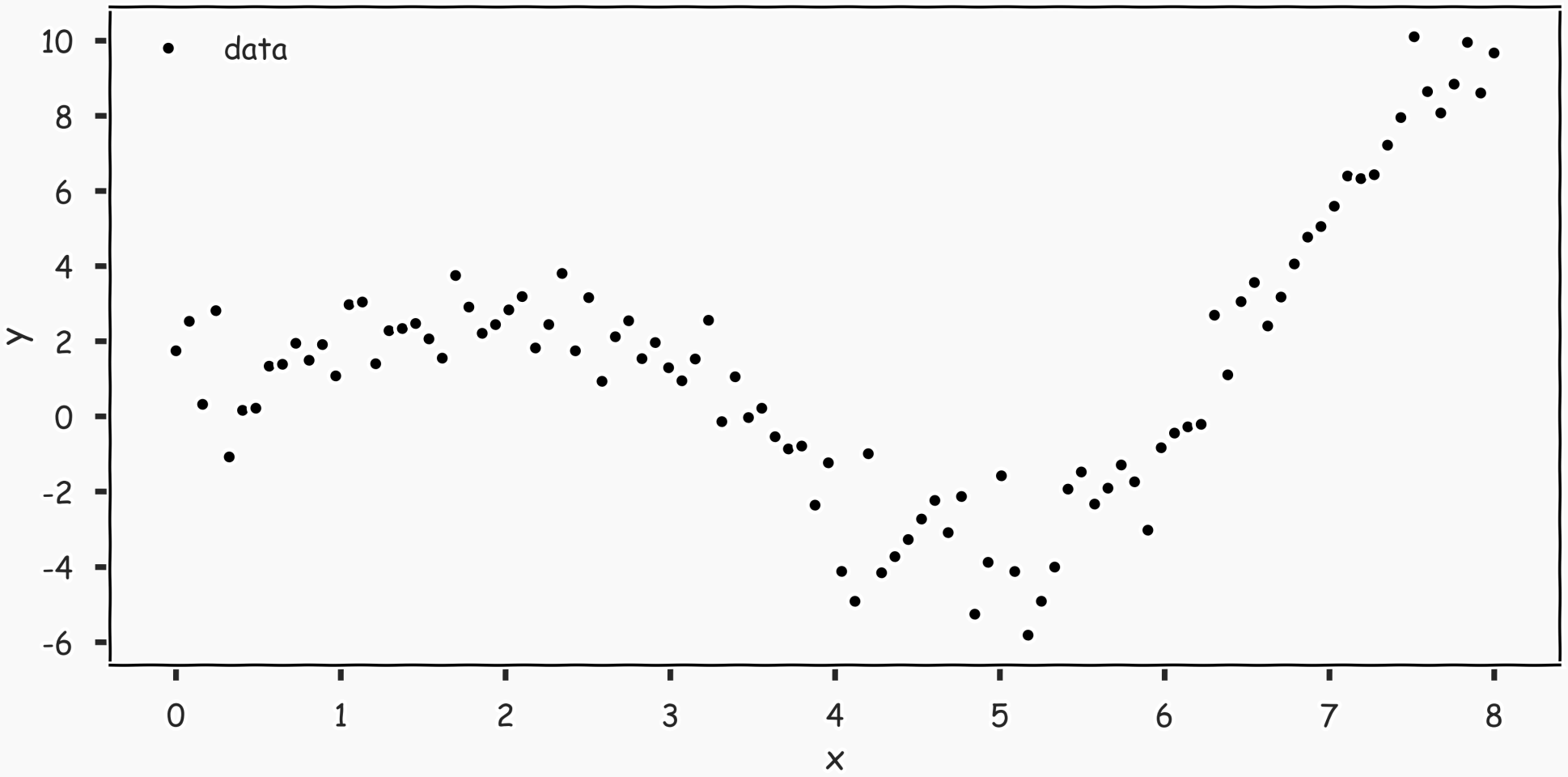
3. Set $T \leftarrow T + \lambda T^{(1)}$

4. Compute residuals, set $r_n \leftarrow r_n - \lambda T^i(x_n), \ n = 1, \ldots, N$

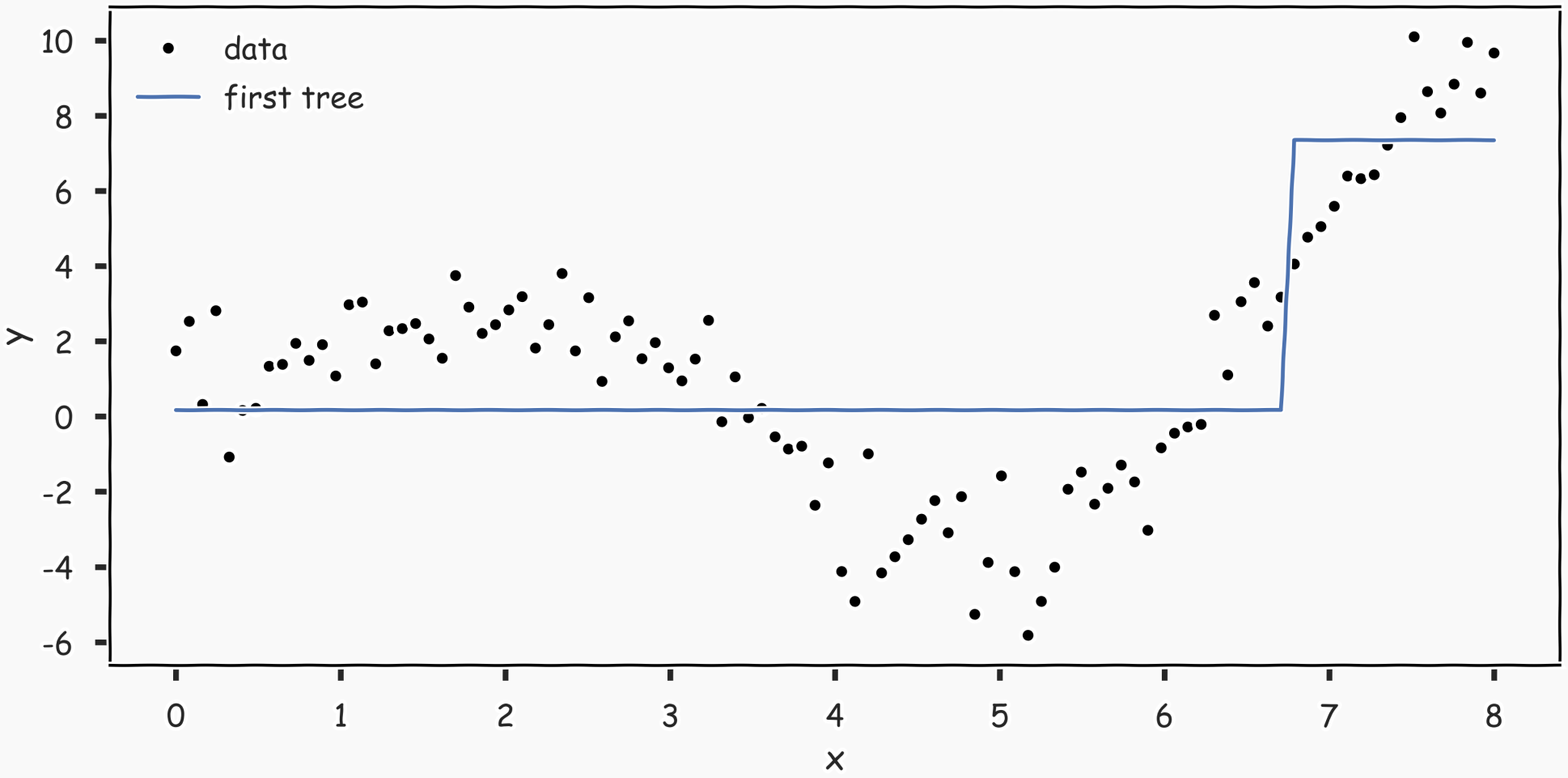
5. Repeat steps 2-4 until ***stopping*** condition met. not improving much anymore

where λ is a constant called the ***learning rate***.

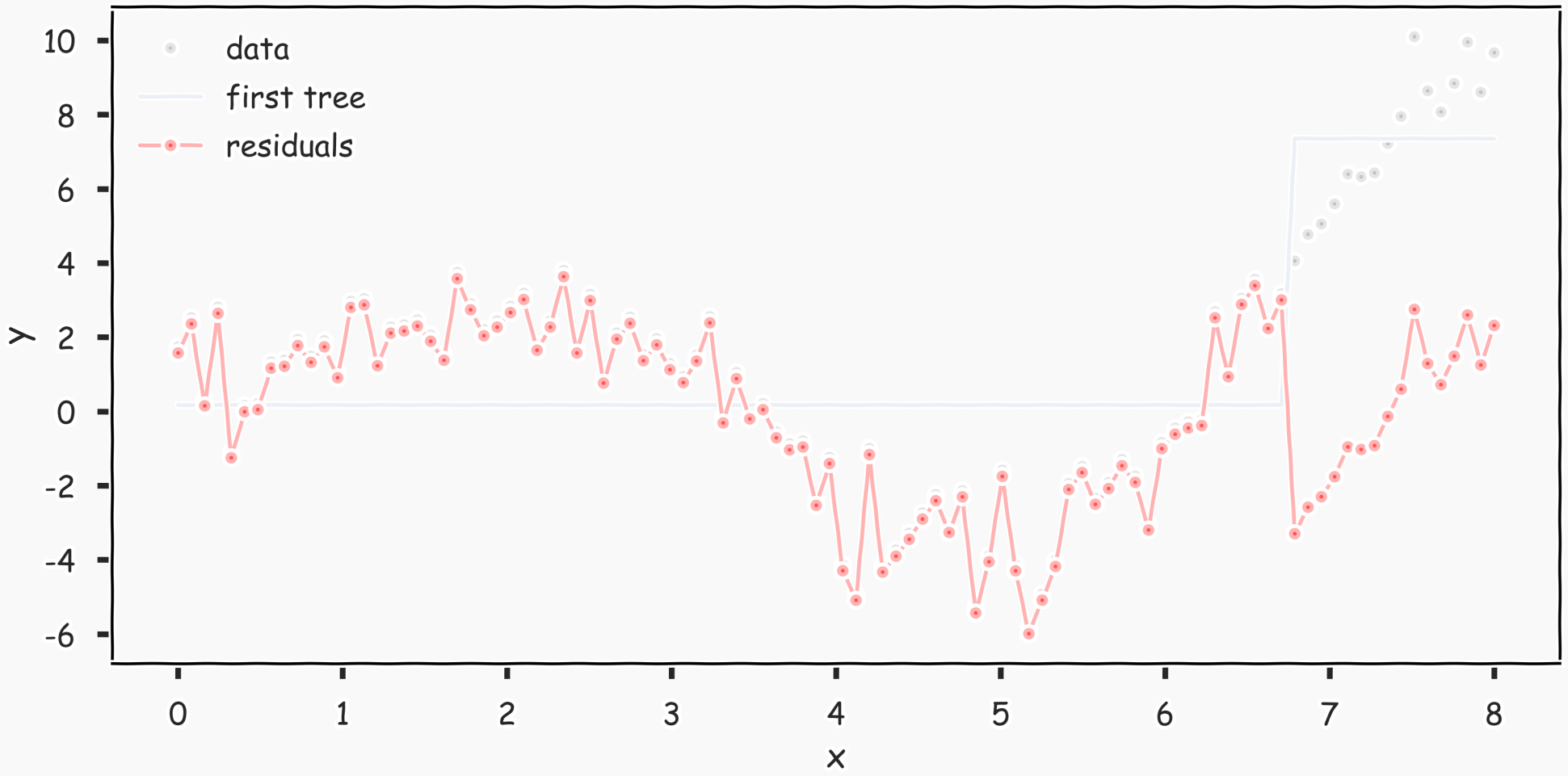
Gradient Boosting: illustration



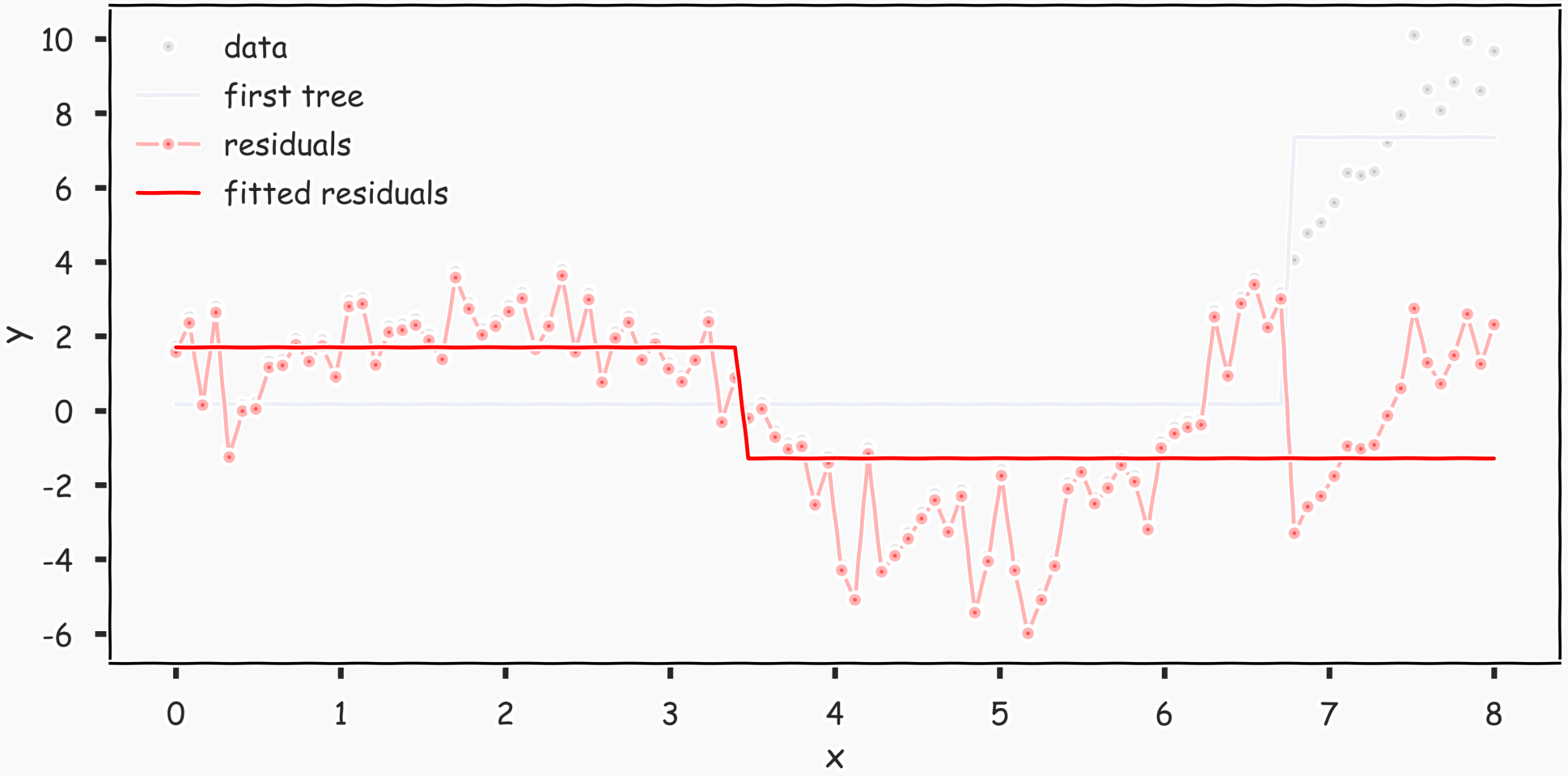
Gradient Boosting: illustration



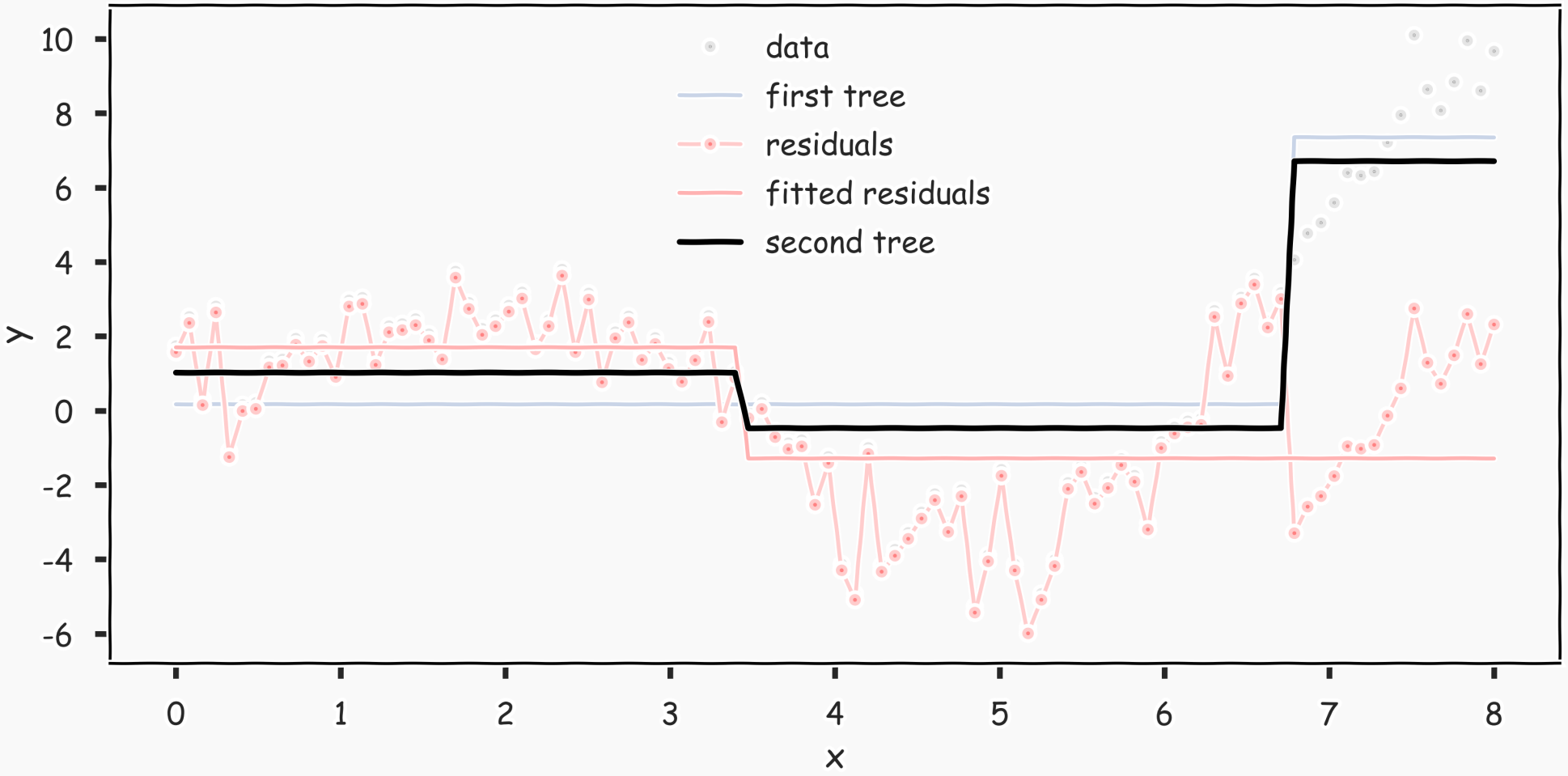
Gradient Boosting: illustration



Gradient Boosting: illustration

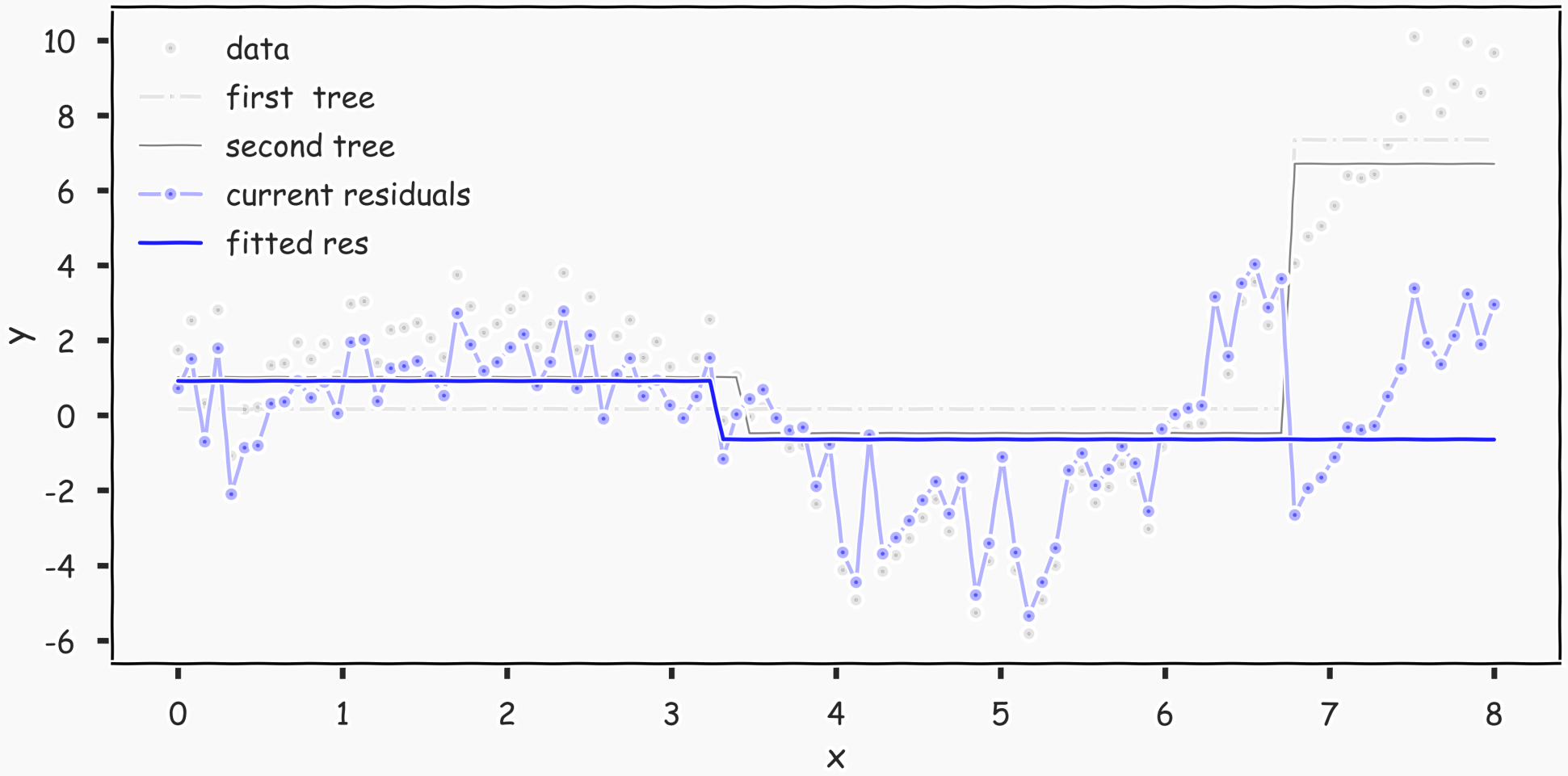


Gradient Boosting: illustration

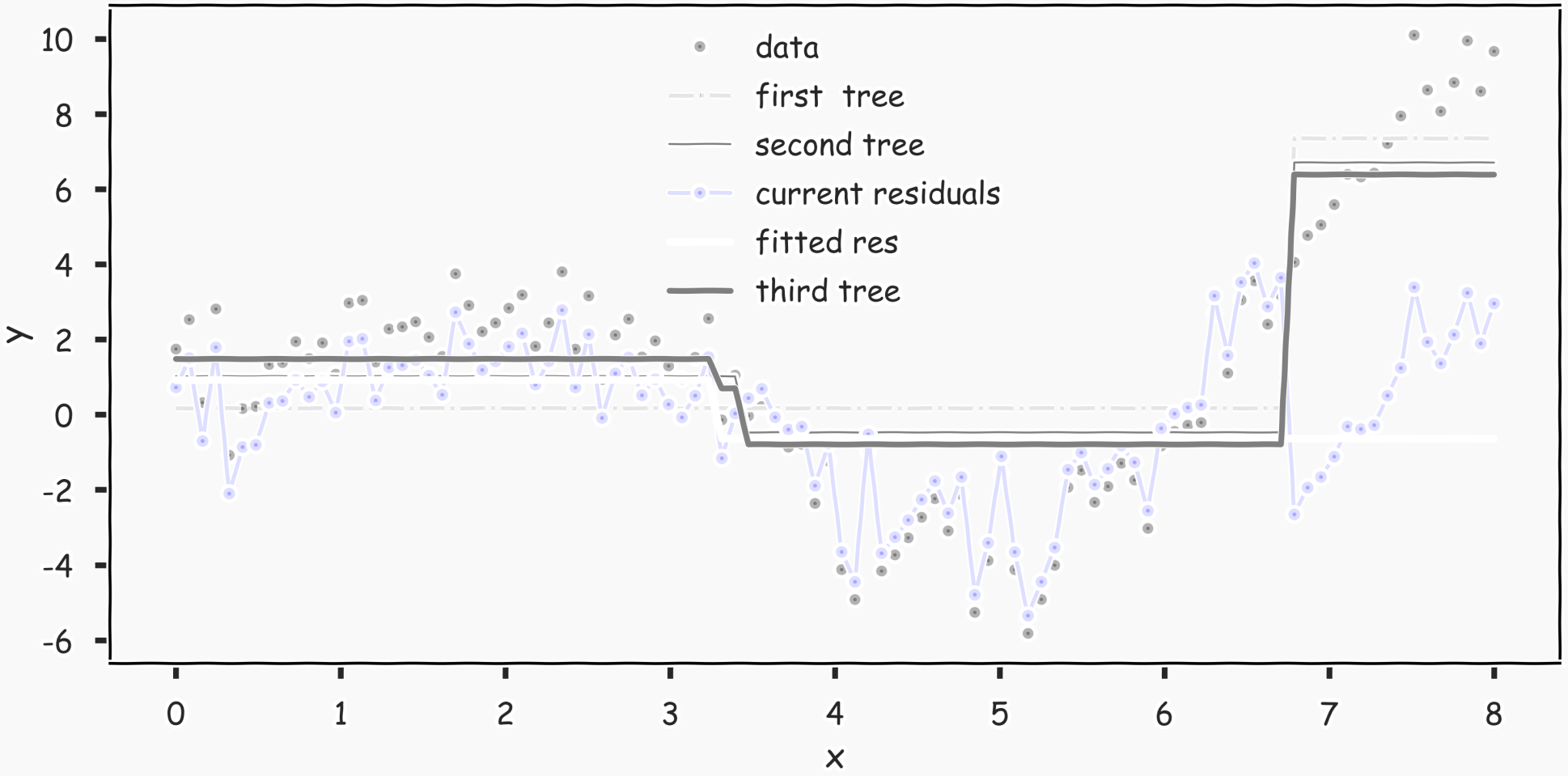


Gradient Boosting: illustration

Gradient Boosting: illustration



Gradient Boosting: illustration



Why Does Gradient Boosting Work?

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***.

Review: A Brief Sketch of Gradient Descent

loss

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.

Review: A Brief Sketch of Gradient Descent the Algorithm

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

Instead, we use an iterative method called ***gradient descent (as discussed in lecture 5)***:

1. Initialize the variables at any value:

$$x = [x_1, \ldots, x_J]$$

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 of the gradient:

The factor λ is often called the learning rate

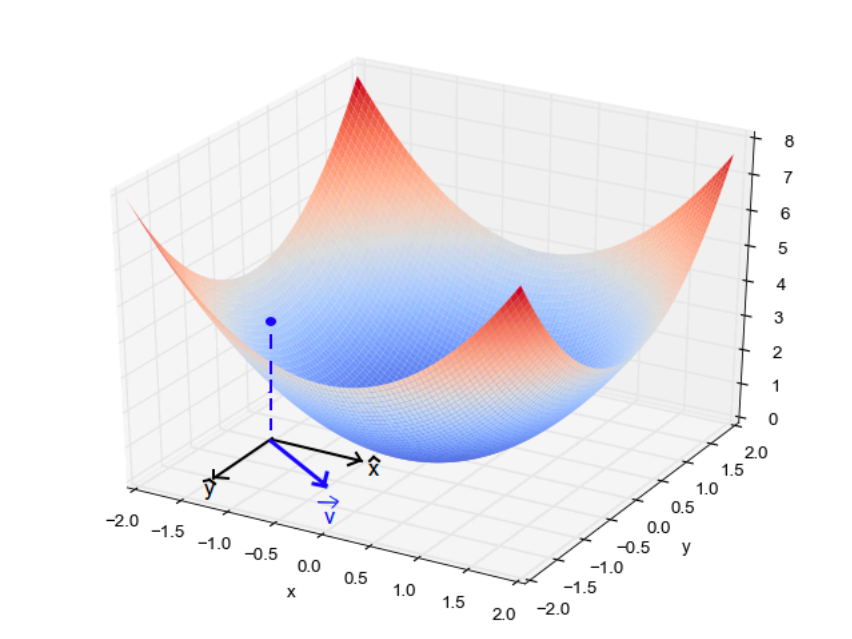
Adjust the variable values by some negative multiple of the gradient

$$x \leftarrow x - \lambda \nabla f(x)$$

Why Does Gradient Descent Work?

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

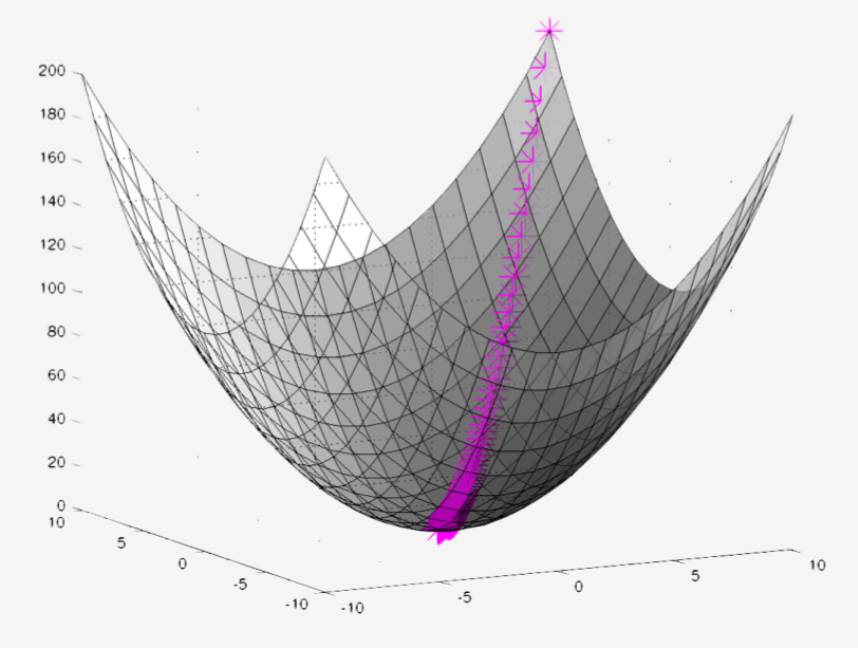
Why does this work? Recall, that as a vector, the gradient at at point gives the direction for the greatest possible rate of increase.



Why Does Gradient Descent Work?

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.



Gradient Boosting as 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 (cont.)

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 (cont.)

$$\hat{y}_n \leftarrow \hat{y}_n + \lambda r_n$$

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:

prediction

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

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 (cont.)

But why do we care that gradient boosting is gradient descent?

By making this connection, we can import the massive amount of techniques for studying gradient descent to analyze gradient boosting.

For example, we can easily reason about how to choose the learning rate λ in gradient boosting.

Choosing a Learning Rate

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

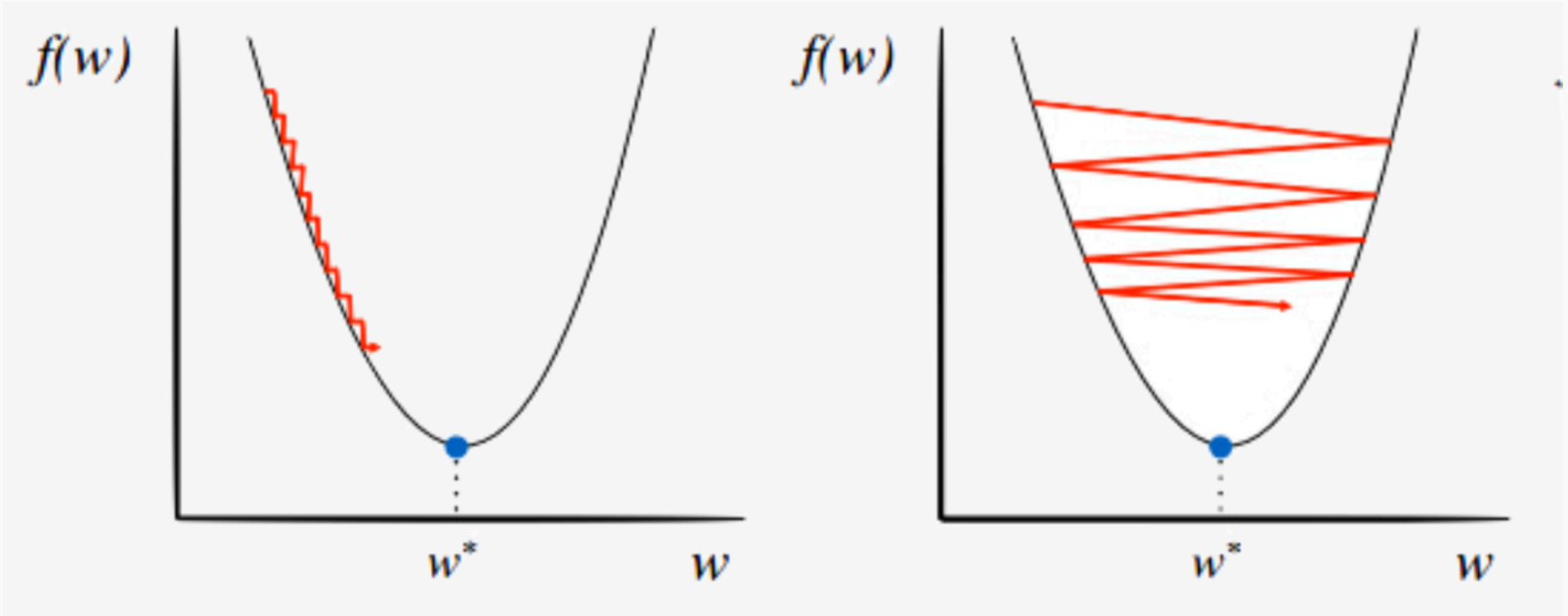
When do we terminate gradient descent?

- 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.

Choosing a Learning Rate

For a constant learning rate, λ, if λ is too small, it takes too many iterations to reach the optimum.



If λ is too large, the algorithm may 'bounce' around the optimum and never get sufficiently close.

Choosing a Learning Rate

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, $\nabla f(x)$

- 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

PPppBoost

AdaBoost

Motivation for AdaBoost

How to quantify error when I have classification

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.

Unfortunately, Error is not differentiable ☹

Motivation for AdaBoost (cont.)

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***

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

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

y = y_hat is exp^-1
y != y_hat is exp^1

Gradient Descent with Exponential Loss

We first compute the gradient for ExpLoss:

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

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

W expresses whether we are doing it right or wrong - W is high for wrong, low for right

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

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

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

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, $w_n = 1/N$. same weight for every point
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:

learn rate, actual val, current pred

$$w_n \leftarrow \frac{w_n \exp\left(-\lambda^{(i)} y_n T^{(i)}(x_n)\right)}{Z}$$

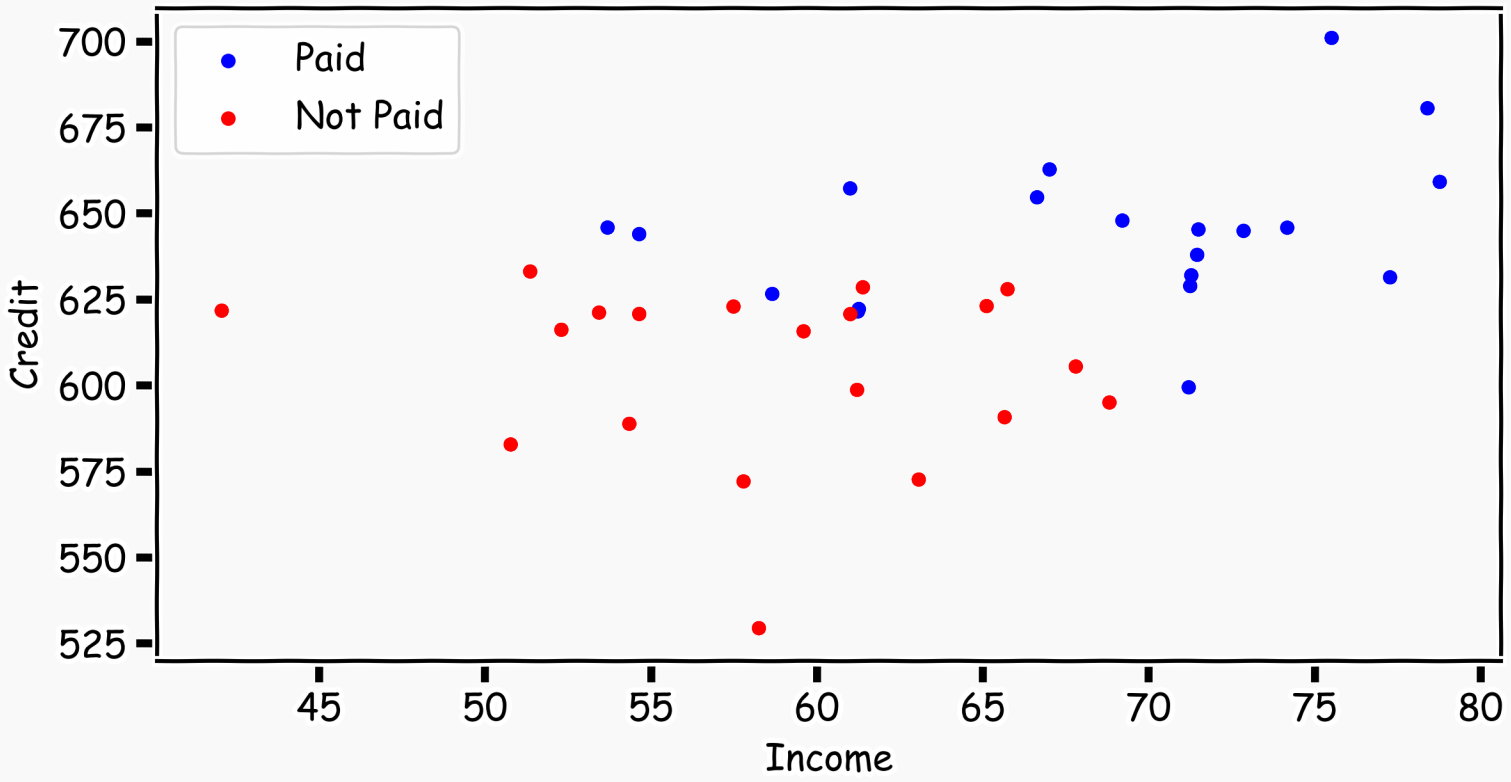
want some memory of previous iterations (Wn)
did you get things right or wrong previously?
want slow adjustment instead of flipping back and forth

where Z is the normalizing constant for the collection of updated weights

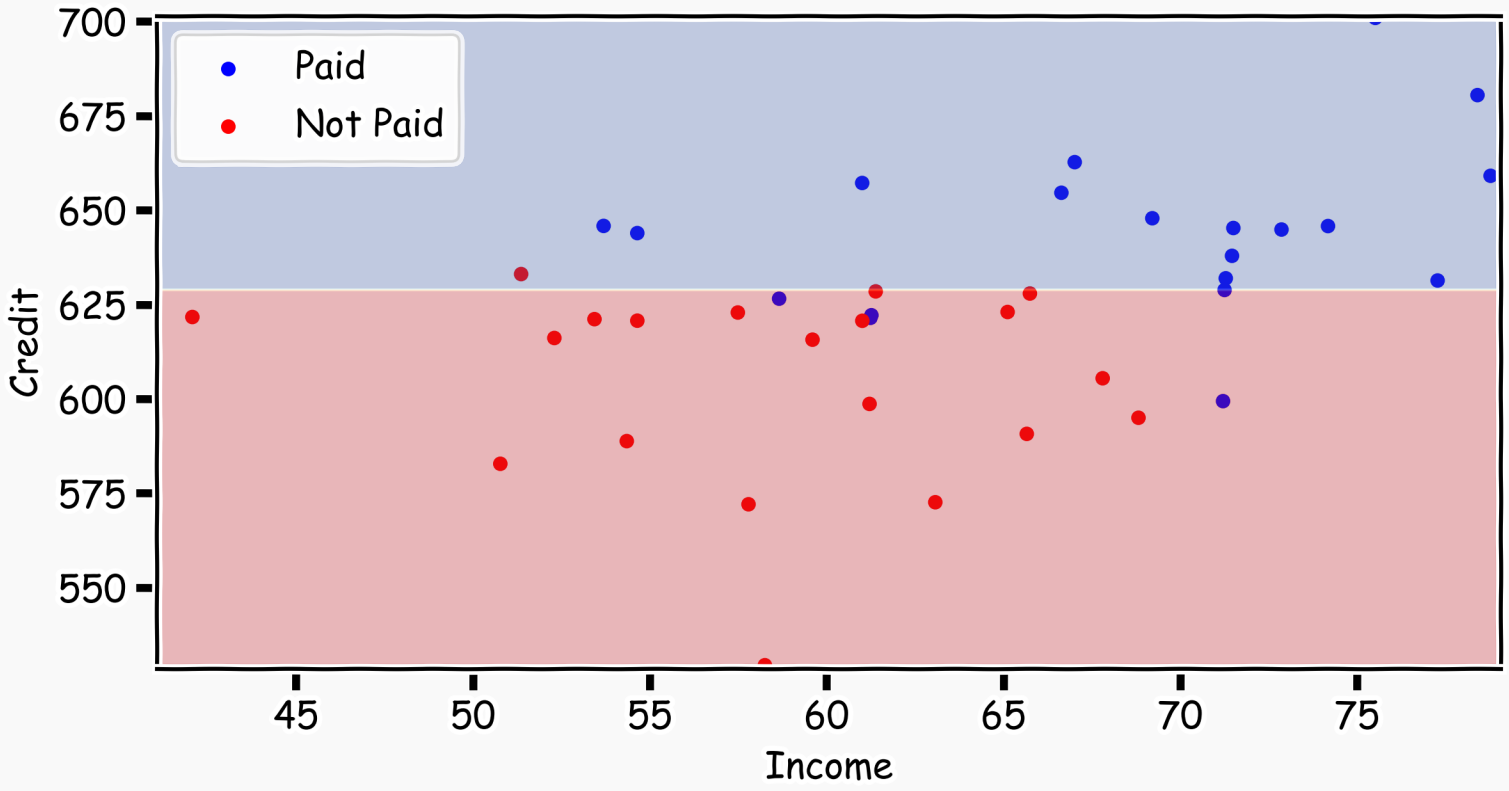
4. Update T: $T \leftarrow T + \lambda^{(i)} T^{(i)}$

where λ is the learning rate.

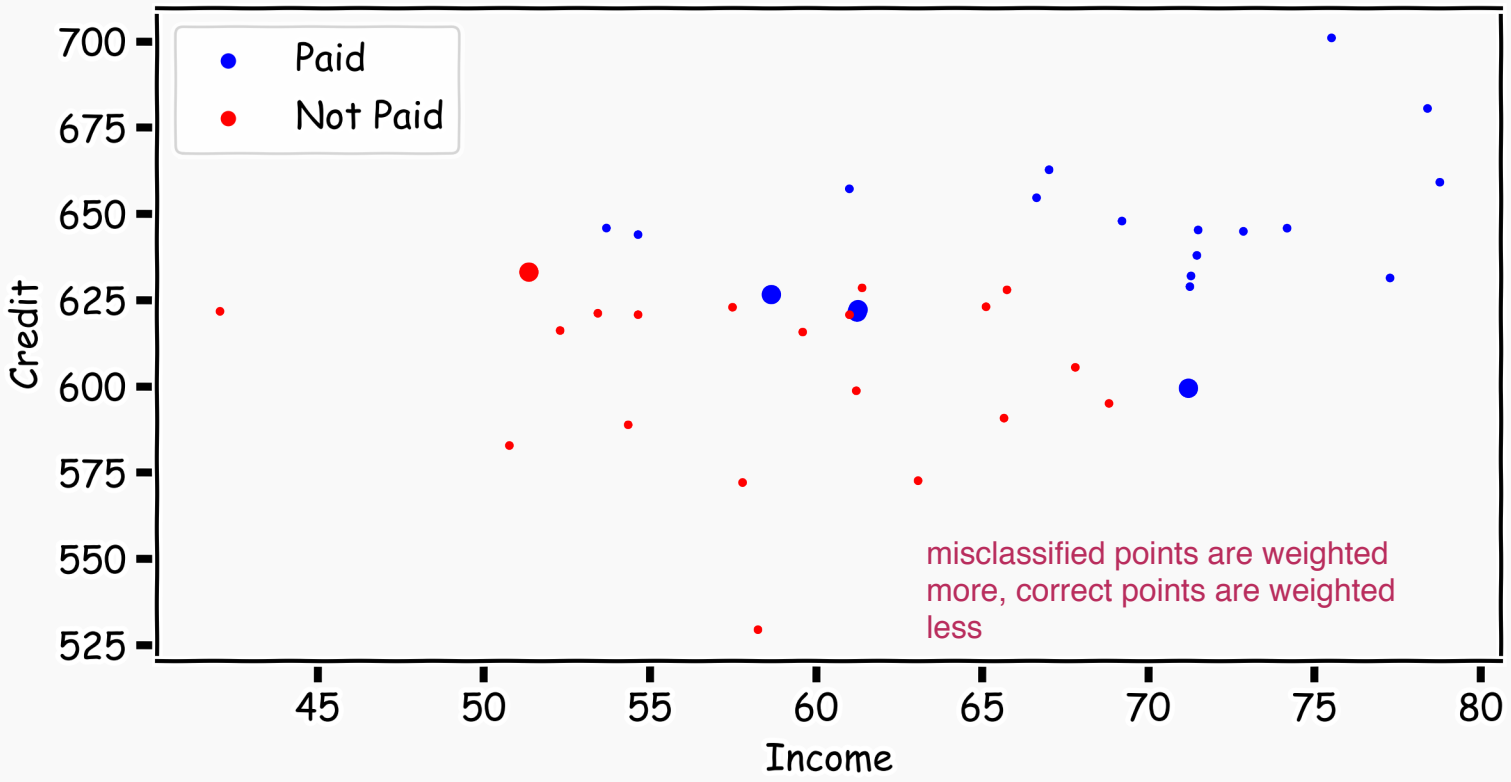
AdaBoost: start with equal weights



AdaBoost: fit a simple decision tree

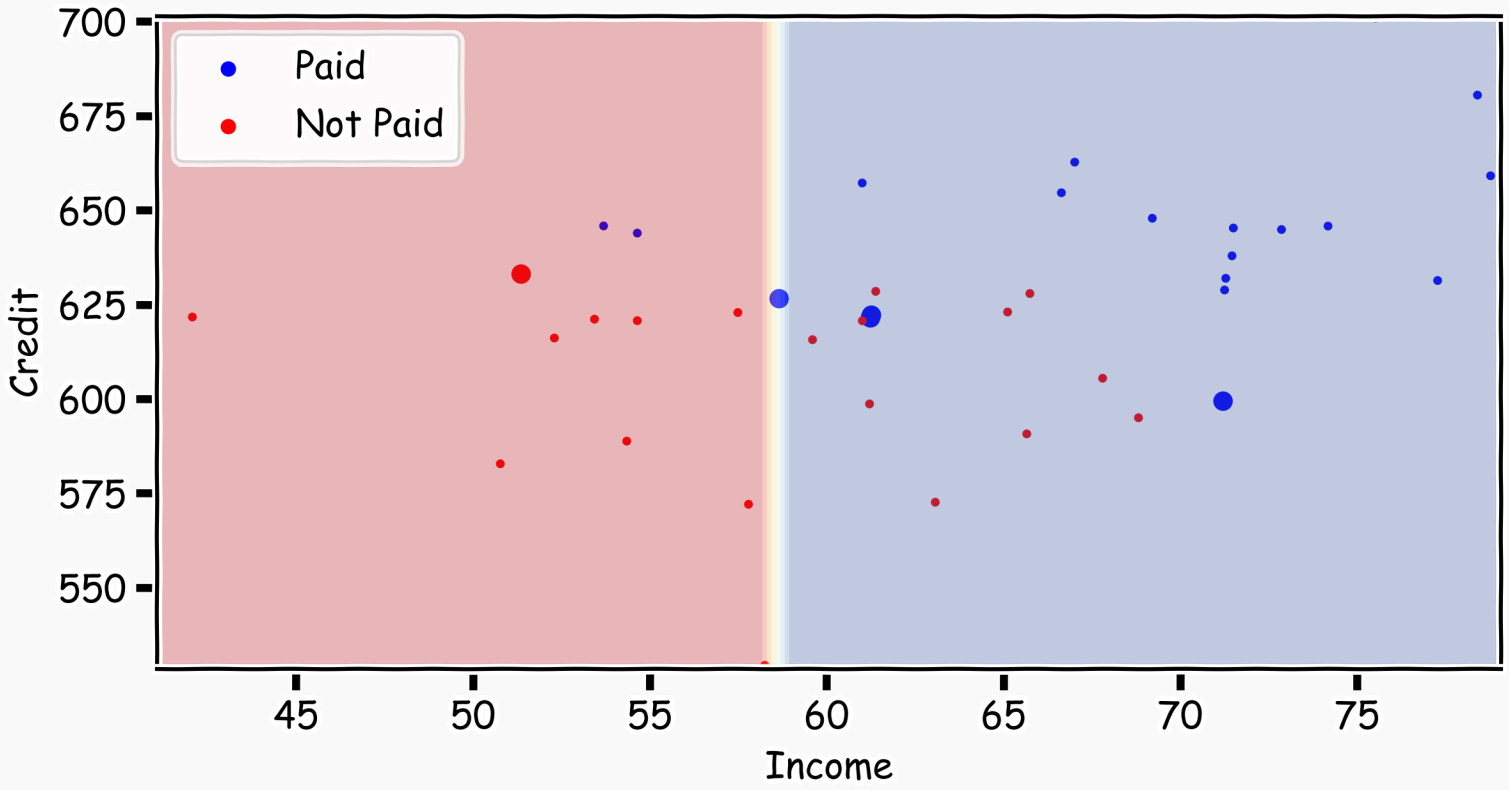


AdaBoost: update the weights

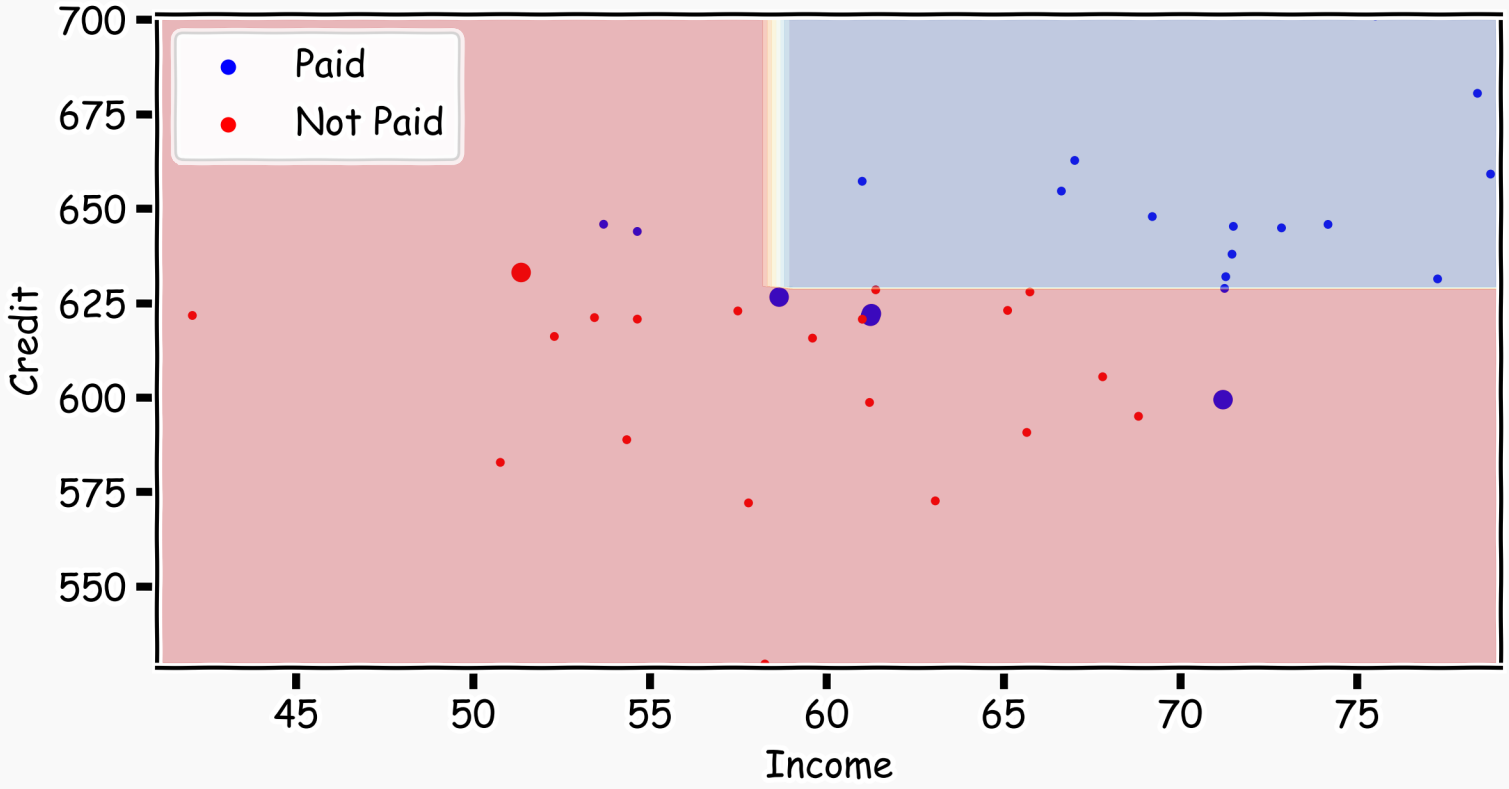


$$w_n \leftarrow \frac{w_n \exp(-\lambda^{(i)} y_n T^{(i)}(x_n))}{Z}$$

AdaBoost: fit another simple decision tree on re-weighted data

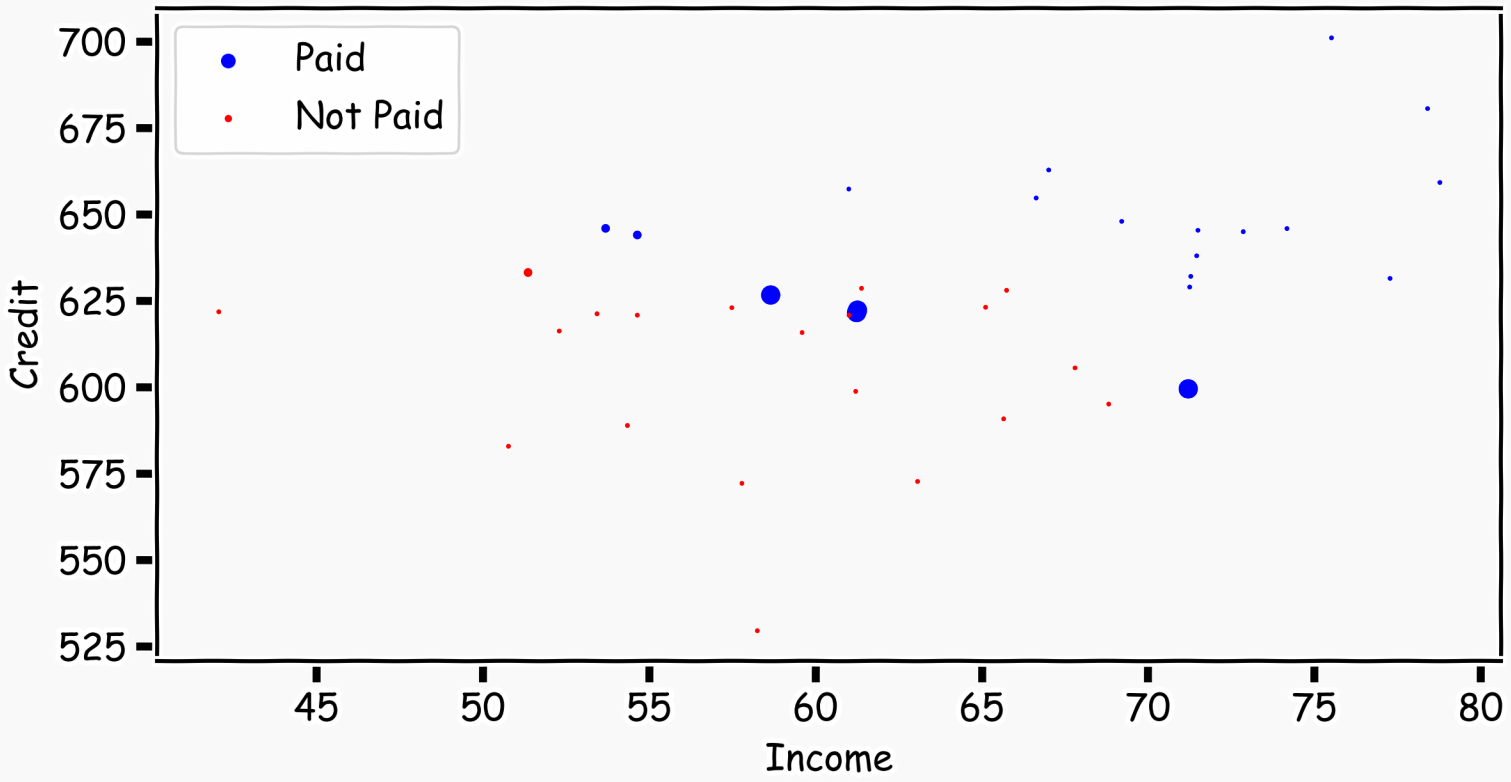


AdaBoost: add the new model to the ensemble



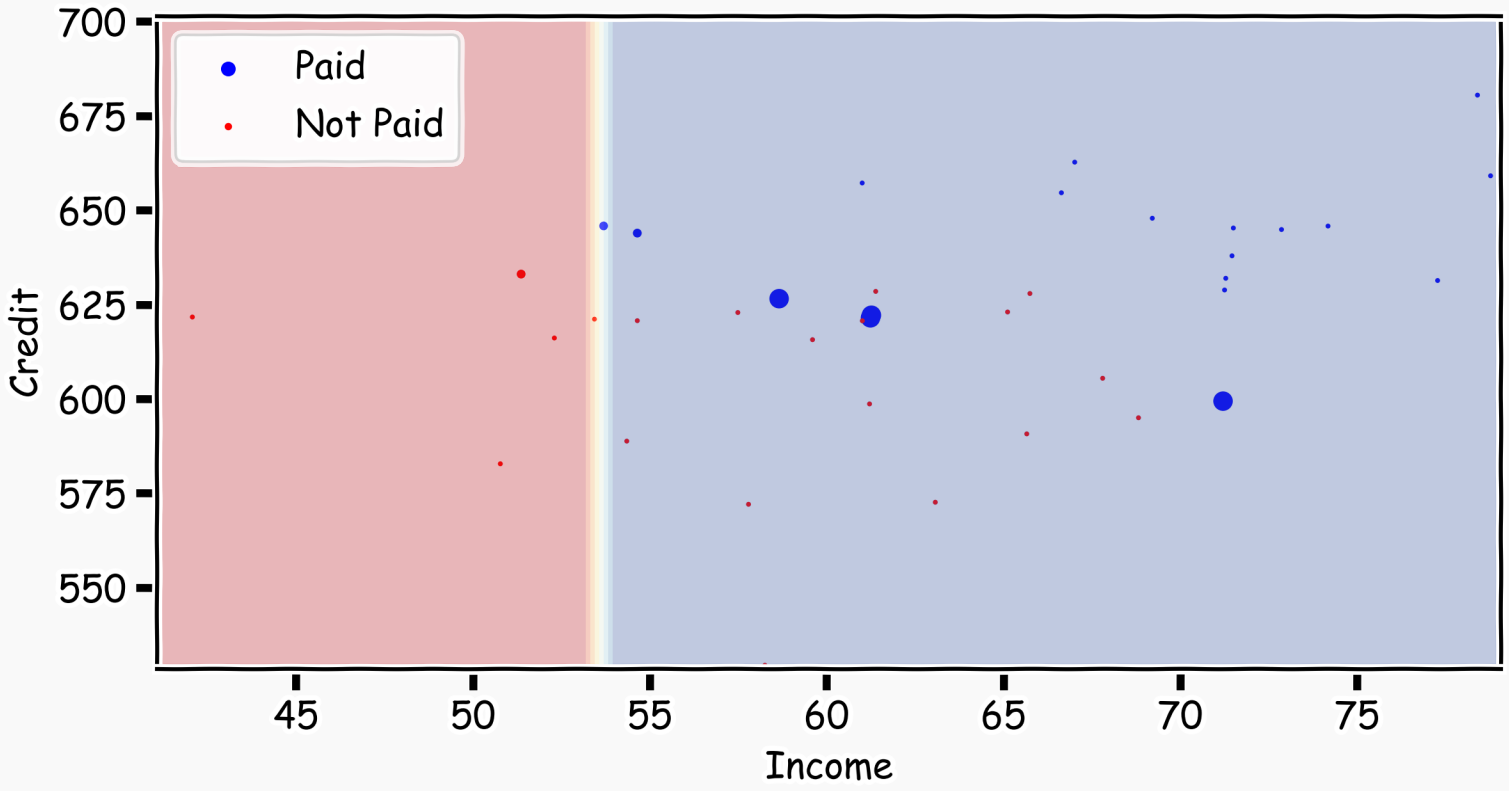
$$T \leftarrow T + \lambda^{(i)} T^{(i)}$$

AdaBoost: update the weights

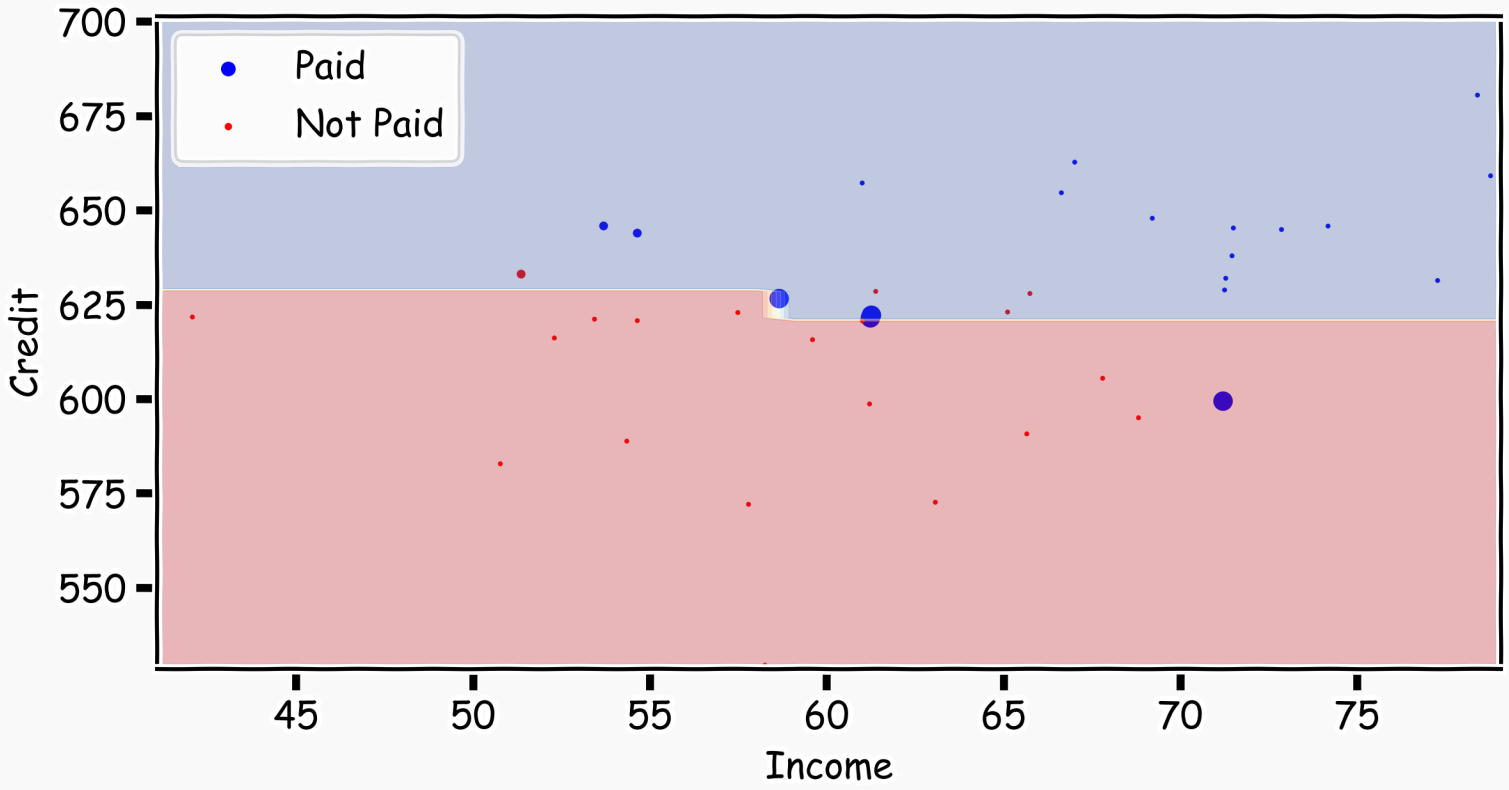


$$w_n \leftarrow \frac{w_n \exp(-\lambda^{(i)} y_n T^{(i)}(x_n))}{Z}$$

AdaBoost: fit a third, simple decision tree on re-weighted data



AdaBoost: add the new model to the ensemble, repeat…



$$T \leftarrow T + \lambda^{(i)} T^{(i)}$$

Choosing the Learning Rate

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

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

Doing so, we get that

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

Final thoughts on Boosting

There are few implementations on boosting:

- XGBoost: An efficient Gradient Boosting Decision
- LGBM: Light Gradient Boosted Machines. It is a library for training GBMs developed by Microsoft, and it competes with XGBoost
- CatBoost: A new library for Gradient Boosting Decision Trees, offering appropriate handling of categorical features

ADVANCED TOPICS

Final thoughts on Boosting

Increasing the number of ***trees*** can lead to overfitting.

Question: Why? You are focusing on the residuals - eventually you will fit perfectly to the training set. You need to stop before you overdo it!