CSCI 567: Machine Learning

Vatsal Sharan

Fall 2022

Lecture 8, Oct 27

USC University of Southern California

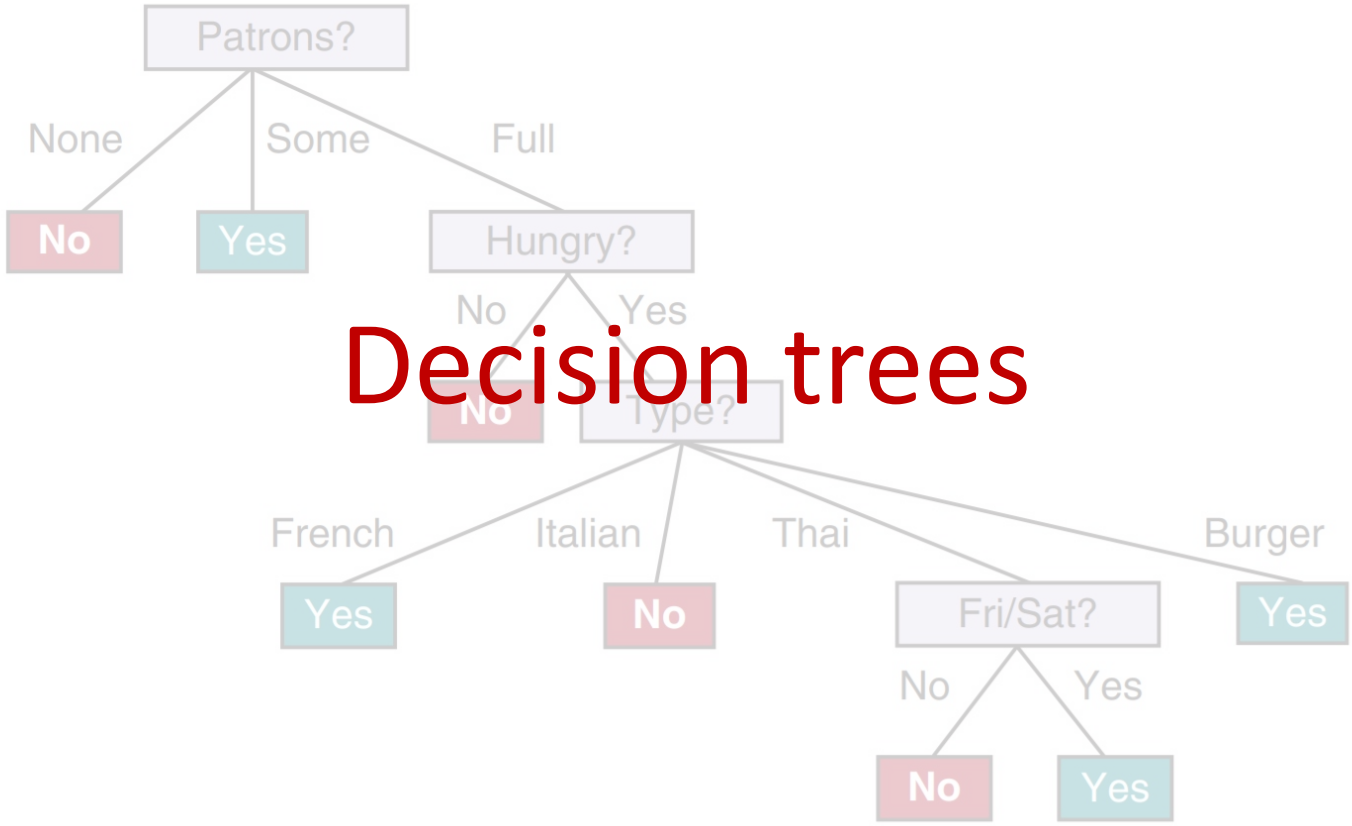
Administrivia

- HW4 will be released in parts. All of it is due together in about 3 weeks.
  - Part 1 on Decision trees and ensemble methods will be released tomorrow.

- Project details will be released early next week.
- Groups of 4 (start forming groups).

- Today's plan:
  - Decision trees
  - Ensemble methods

Decision trees



Decision trees

- Introduction & definition
- Learning the parameters
- Measures of uncertainty
- Recursively learning the tree & some variants

Decision trees

We have seen different ML models for classification/regression:

- linear models, nonlinear models induced by kernels, neural networks

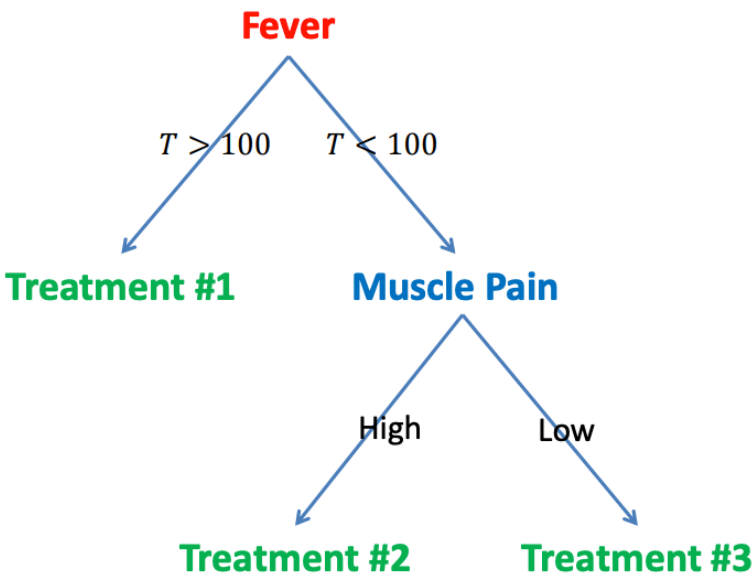
Decision tree is another popular one:

- nonlinear in general
- works for both classification and regression; we focus on classification
- one key advantage is good interpretability
- *ensembles* of trees are very effective

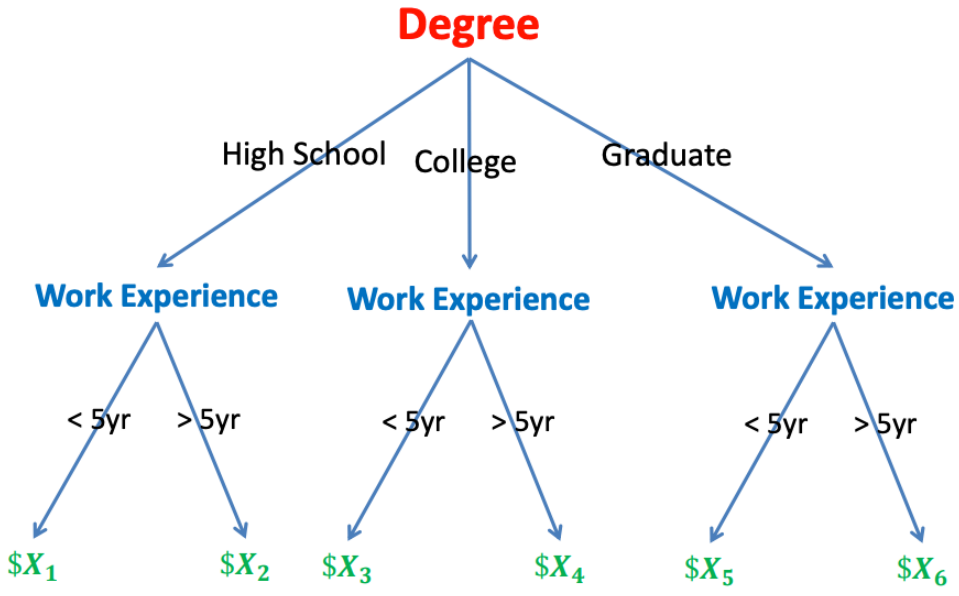
Example

Many decisions are made based on some tree structure

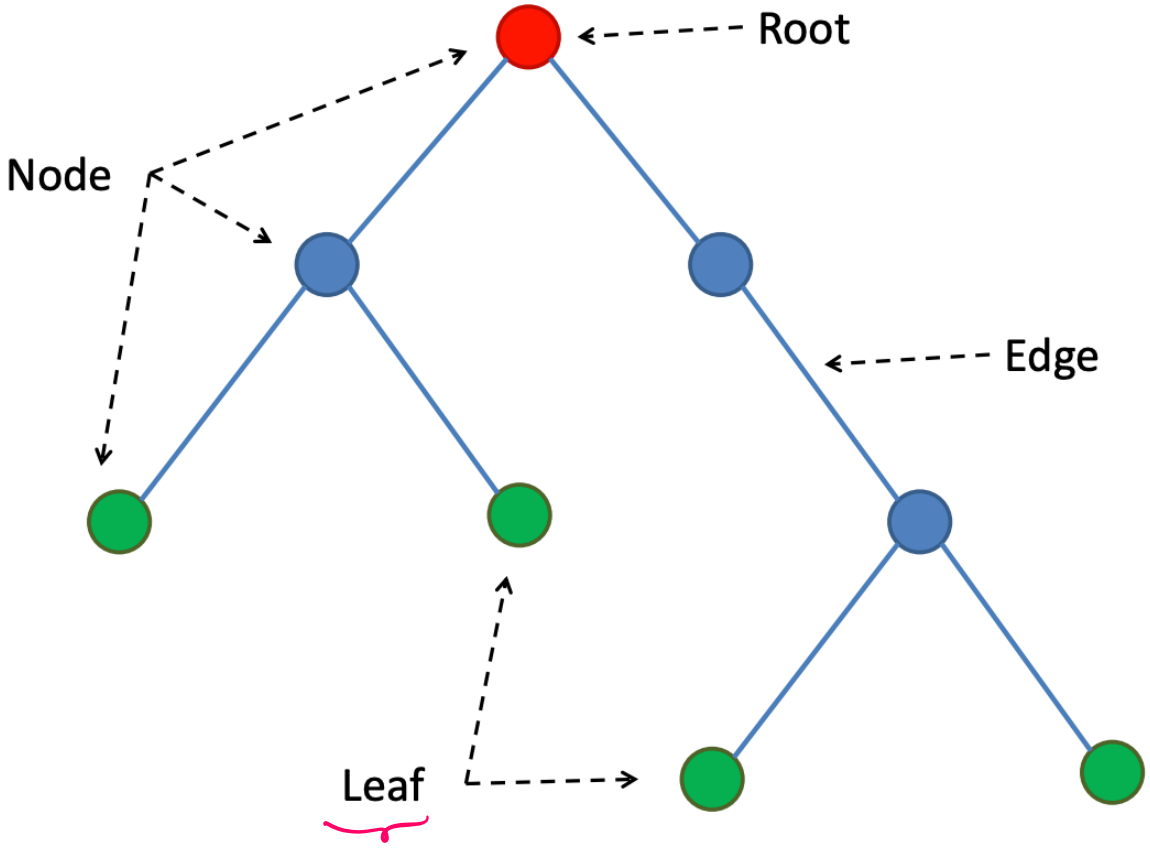
Medical treatment



Salary in a company



Tree terminology



nodes with no children

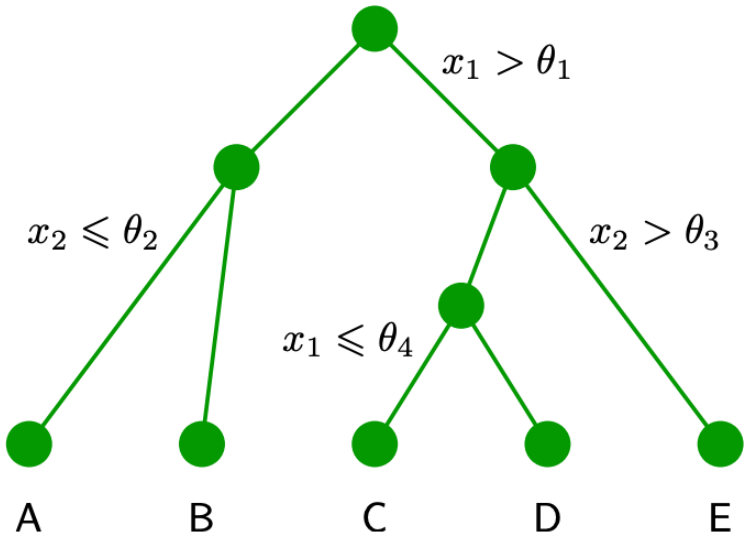
Tree terminology

Input: $\boldsymbol{x} = (x_1, x_2)$

decision tree

Output: $f(\boldsymbol{x})$ determined naturally by traversing the tree

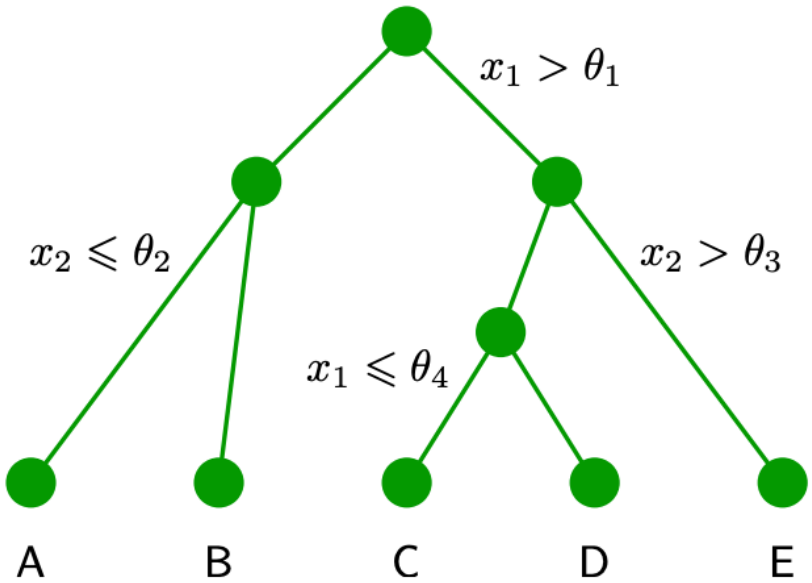
- start from the root
- test at each node to decide which child to visit next
- finally the leaf gives the prediction $f(\boldsymbol{x})$

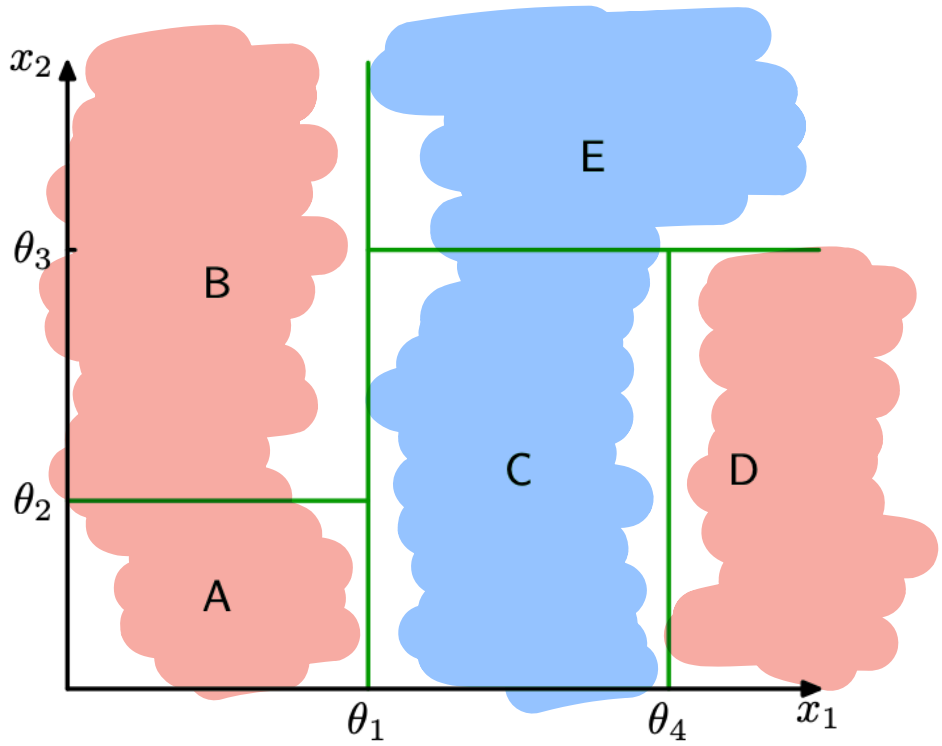


For example, $f((\theta_1 - 1, \theta_2 + 1)) = \mathsf{B}$

Complex to formally write down, but easy to represent pictorially or as code.

Decision boundary





if A, B, D : +ve label (red)
C, E : -ve label (blue)

Nonlinear decision boundary

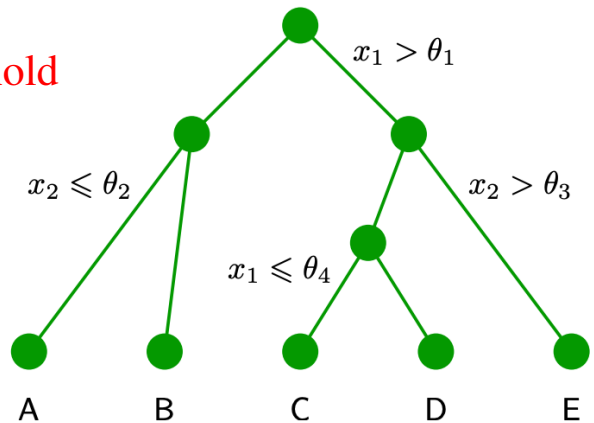
Parameters

Parameters to learn for a decision tree:

- The structure of the tree, such as the depth, #branches, #nodes, etc.
  Some of these are considered as hyperparameters. The structure of a tree is *not fixed in advance, but learned from data*.

- The test at each internal node:
  Which feature(s) to test on? If the feature is continuous, what threshold $(\theta_1, \theta_2, \ldots)$?

- The value/prediction of the leaves (A, B, ...)



Decision trees

- Introduction & definition
- Learning the parameters
- Measures of uncertainty
- Recursively learning the tree & some variants

Learning the parameters (optimization?)

So how do we *learn all these parameters?*

Empirical risk minimization (ERM): find the parameters that minimize some loss.

However, doing exact ERM *is too expensive for trees*.

this is size of the function class $|\mathcal{F}|$

- for T nodes, there are roughly $(\#\text{features})^T$ possible decision trees (need to decide which feature to use on each node).

- enumerating all these configurations to find the one that minimizes some loss is too computationally expensive.

- since most of the parameters are discrete (#branches, #nodes, feature at each node, etc.) *cannot really use gradient based approaches*.

e.g. can't really use gradient descent to decide which feature $(+1, \ldots, +d)$ to split on at any level...

Instead, we turn to some **greedy top-down approach**.

A running example

Russell & Norvig, AIMA

- predict whether a customer will wait to get a table at some restaurant
- 12 training examples
- 10 features (all discrete)

are there any alternate options nearby?

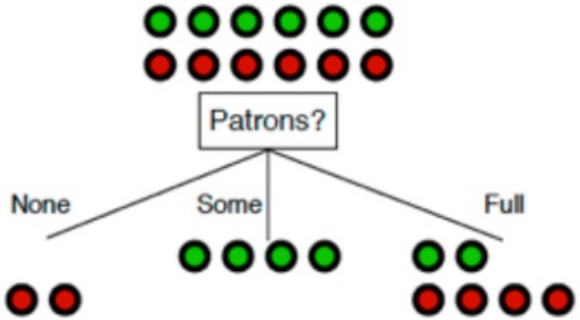
does it have a bar?

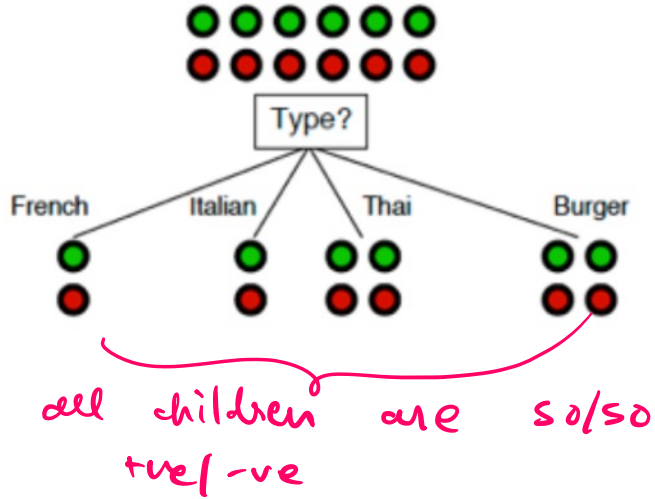
Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

"structured data": mix of categorical attributes, continuous attributes, all features have some meaning

First step: How to build the root?

Which feature should we test at the root? Examples:





Which split is better?

- intuitively "patrons" is a better feature since it leads to "more certain" children
- how to quantify this intuition?

Decision trees

- Introduction & definition
- Learning the parameters
- Measures of uncertainty
- Recursively learning the tree & some variants

Measure of uncertainty of a node

The uncertainity of a node should be **a function of the distribution of the classes within the node**.

Example: a node with 2 positive and 4 negative examples can be summarized by a distribution P with $P(Y = +1) = 1/3$ and $P(Y = -1) = 2/3$

One classic measure of the uncertainity of a distribution is its *(Shannon) entropy*:

$$H(P) = -\sum_{k=1}^{C} P(Y = k) \log P(Y = k)$$

C → # classes

Properties of entropy

$$H(P) = \mathbb{E}_{Y\sim P}\left[\log\left(\frac{1}{P(Y)}\right)\right]$$

this is a measure of how unlikely the outcome was

$$= \sum_{k=1}^{C} P(Y=k)\log\left(\frac{1}{P(Y=k)}\right)$$

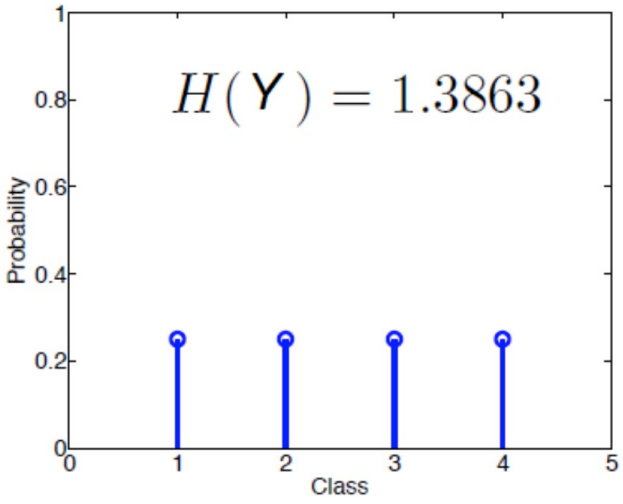
this is average "unlikeliness" when we sample outcomes from P.

$$= -\sum_{k=1}^{C} P(Y=k)\log P(Y=k)$$

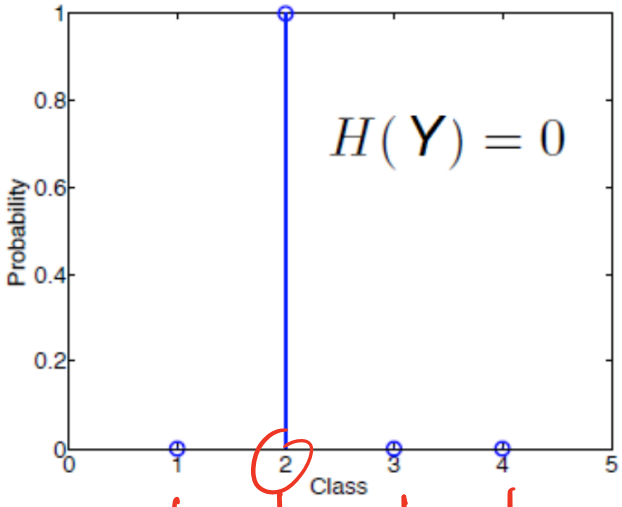
assign shorter codes to outcomes which are more likely

- the base of log can be 2, e or 10
- always non-negative
- it's the *smallest codeword length to encode symbols drawn from P*
- maximized if P is uniform (max = ln C): most uncertain case
- minimized if P focuses on one class (min = 0): most certain case
  - e.g. $P = (1, 0, \ldots, 0)$
  - $0 \log 0$ is defined naturally as $\lim_{z\to 0+} z \log z = 0$

Examples of computing entropy

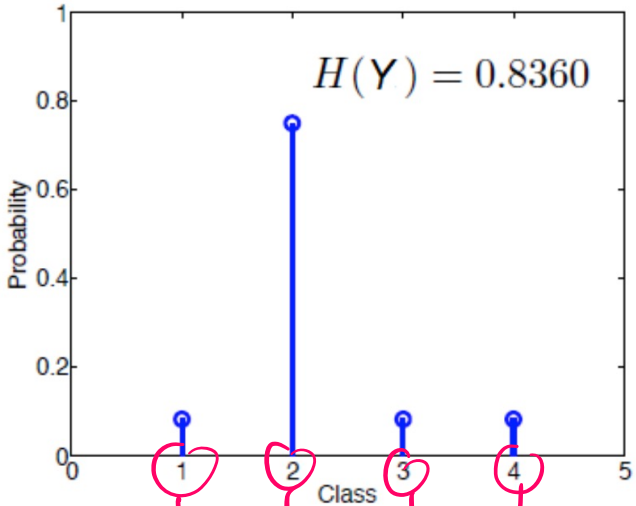


most uncertain



most certain

0



00 01 10 11

Examples of computing entropy

Entropy in each child if root tests on "patrons"

For "None" branch

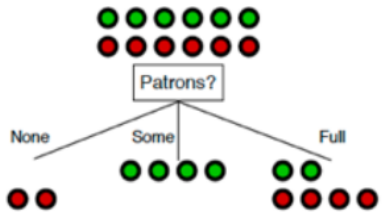
$$-\left(\frac{0}{0+2}\log\frac{0}{0+2}+\frac{2}{0+2}\log\frac{2}{0+2}\right)=0$$

For "Some" branch

$$-\left(\frac{4}{4+0}\log\frac{4}{4+0}+\frac{0}{4+0}\log\frac{0}{4+0}\right)=0$$

For "Full" branch

$$-\left(\frac{2}{2+4}\log\frac{2}{2+4}+\frac{4}{2+4}\log\frac{4}{2+4}\right)\approx 0.9$$



So how good is choosing "patrons" overall?

Very naturally, we take the weighted average of entropy:

$$\frac{2}{12}\times 0+\frac{4}{12}\times 0+\frac{6}{12}\times 0.9=0.45$$

Measure of uncertainty of a split

Suppose we split based on a discrete feature A, the uncertainty can be measured by the **conditional entropy**:

$$
\begin{aligned}
& H(Y \mid A) \\
&= \sum_a P(A=a) H(Y \mid A=a) \\
&= \sum_a P(A=a) \left(-\sum_{k=1}^{C} P(Y \mid A=a) \log P(Y \mid A=a) \right) \\
&= \sum_a \text{“fraction of examples at node } A=a\text{”} \times \text{“entropy at node } A=a\text{”}
\end{aligned}
$$

calculation from e.g.

Pick the feature that leads to the smallest conditional entropy.

Deciding the root

For "French" branch

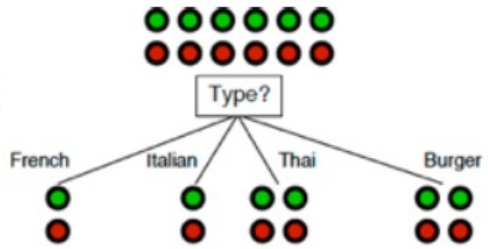
$$-\left(\frac{1}{1+1}\log\frac{1}{1+1}+\frac{1}{1+1}\log\frac{1}{1+1}\right)=1$$

For "Italian" branch

$$-\left(\frac{1}{1+1}\log\frac{1}{1+1}+\frac{1}{1+1}\log\frac{1}{1+1}\right)=1$$

For "Thai" and "Burger" branches

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



The conditional entropy is $\frac{2}{12}\times 1+\frac{2}{12}\times 1+\frac{4}{12}\times 1+\frac{4}{12}\times 1=1>0.45$

So splitting with "patrons" is better than splitting with "type".

In fact by similar calculation "patrons" is the best split among all features.

We are now done with building the root (this is also called a **stump**).

a decision tree with only a root

Decision trees

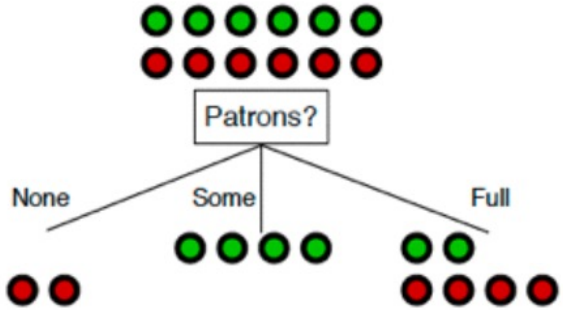
- Introduction & definition
- Learning the parameters
- Measures of uncertainty
- Recursively learning the tree & some variants

Repeat recursively

Split each child in the same way.

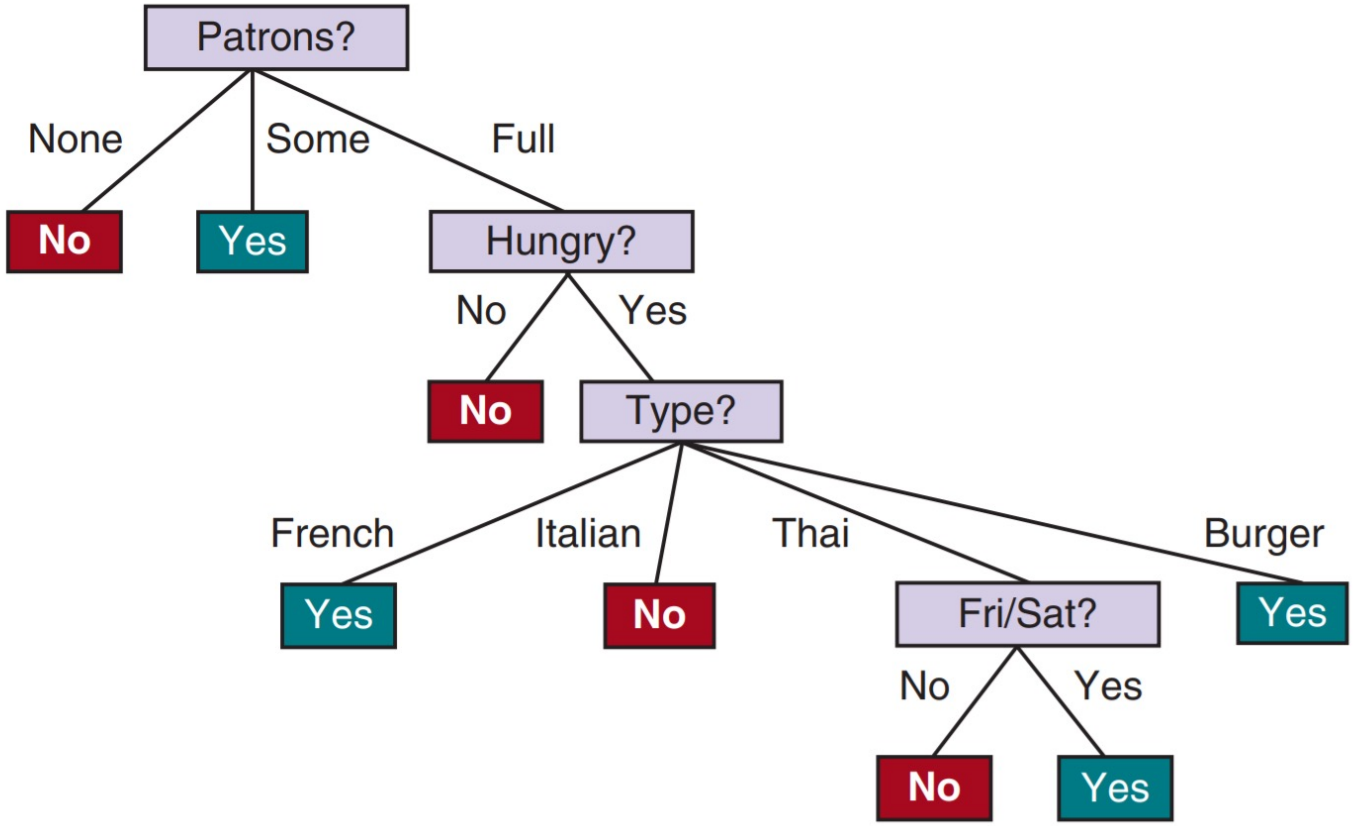
- but no need to split children "none" and "some": they are pure already and will be our leaves

- for "full", repeat, focusing on those 6 examples:



	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	WillWait
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

Repeat recursively



Putting it together

DecisionTreeLearning(Examples)

- if Examples have the same class, return a leaf with this class
- else if Examples is empty, return a leaf with majority class of parent
- else

    find the best feature A to split (e.g. based on conditional entropy)

    Tree $\leftarrow$ a root with test on A

    For each value a of A:

        Child $\leftarrow$ **DecisionTreeLearning**(Examples with $A = a$)
        add **Child** to **Tree** as a new branch

- return **Tree**

Variants

Popular decision tree algorithms (e.g. C4.5, CART, etc) are all based on this framework.

Variants:

- replace entropy by **Gini impurity**:

$$G(P) = \sum_{k=1}^{C} P(Y = k)(1 - P(Y = k))$$

pick example at random

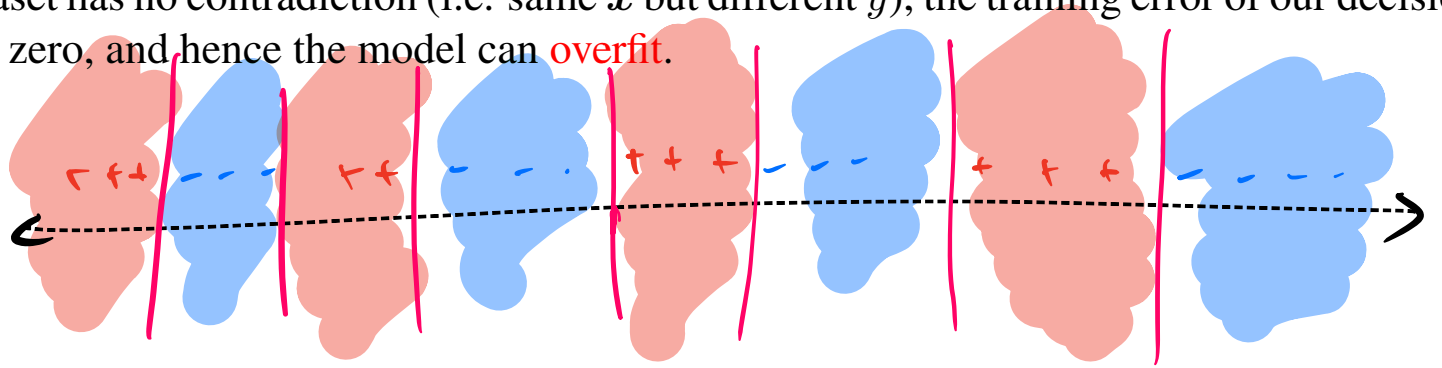
Prob of that sample being misclassified if we predict its label based on the label of another randomly sampled point.

meaning: *how often a randomly chosen example would be incorrectly classified if we predict according to another randomly picked example*

- if a feature is continuous, we need to find a threshold that leads to minimum conditional entropy or Gini impurity. *Think about how to do it efficiently.*

Regularization

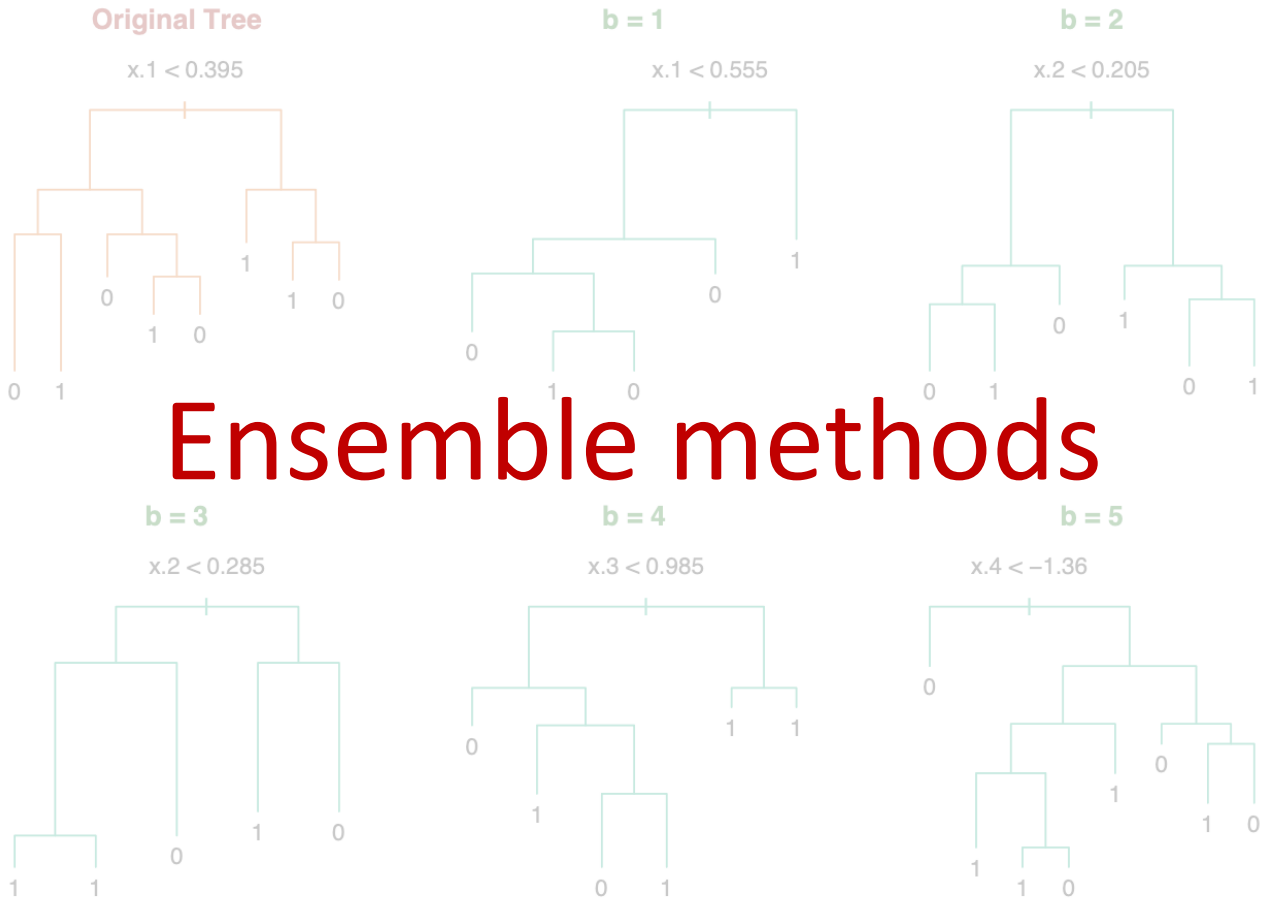
If the dataset has no contradiction (i.e. same $\boldsymbol{x}$ but different y), the training error of our decision tree algorithm is always zero, and hence the model can overfit.



To prevent overfitting:

- restrict the depth or #nodes (e.g. stop building the tree when the depth reaches some threshold).
- do not split a node if the #examples at the node is smaller than some threshold.
- other approaches as well, all make use of a validation set to tune these hyperparameters

You'll explore this in HW4.



Ensemble methods

Acknowledgement

We borrow some of the content from Stanford's CS229 slides on Ensemble Methods, by Nandita Bhaskhar:

https://cs229.stanford.edu/lectures-spring2022/cs229-boosting_slides.pdf

Ensemble methods

- Bagging
- Random forests
- Boosting: Basics
- Adaboost
- Gradient boosting

Decision Trees Recap

Pros

- Can handle large datasets
- Can handle mixed predictors (continuous, discrete, qualitative)
- Can ignore redundant variables
- Can easily handle missing data
- Easy to interpret if small

Cons

- Prediction performance is often poor (because it does not generalize well)
- Large trees are hard to interpret

Ensemble Methods for Decision Trees

Key idea: Combine multiple classifiers to form a learner with better performance than any of them individually ("wisdom of the crowd")

Issue: A single decision tree is very unstable, small variations in the data can lead to very different trees (since differences can propagate along the hierarchy).

They are *high variance models*, which can overfit.

↳ a model whose predictions can vary a lot based on randomness in data.

But they have many advantages (e.g. very fast, robust to data variations).

Q: How can we lower the variance?

A: Let's learn multiple trees!

How to ensure they don't all just learn the same thing??

Bagging

Bagging (Breiman, 1996)
Bootstrap **Agg**regat**ing**: lowers variance

Ingredients:
Bootstrap sampling: get different splits/subsets of the data
Aggregating: by averaging

Procedure:
→ Get multiple random splits/subsets of the data
→ Train a given procedure (e.g. decision tree) on each subset
→ Average the predictions of all trees to make predictions on test data

Leads to estimations with reduced variance.

Bagging

Collect T subsets each of some fixed size (say m) by sampling with replacement from training data. → Bootstrap sampling, put datapoint back after sampling it.

Let $f_t(\boldsymbol{x})$ be the classifier (such as a decision tree) obtained by training on the subset $t \in \{1, \ldots, T\}$. Then the aggregrated classifier $f_{agg}(\boldsymbol{x})$ is given by:

$$f_{agg}(\boldsymbol{x}) = \begin{cases} \frac{1}{T}\sum_{t=1}^{T} f_t(\boldsymbol{x}) & \text{for regression,} \\ \text{sign}\left(\frac{1}{T}\sum_{t=1}^{T} f_t(\boldsymbol{x})\right) = \text{Majority Vote}\{f_t(\boldsymbol{x})\}_{t=1}^{T} & \text{for classification.} \end{cases}$$

→ ±1 valued (binary classification)

Why majority vote? "Wisdom of the crowd"

Bagging

Why majority vote? "Wisdom of the crowd"

Suppose I ask each of you: "Will the stock market go up tomorrow?"
Suppose each of you has a 60% chance of being correct, and all of you make **independent predictions (probability of any 1 person being correct is independent of probability of any one else being correct).**

What is Probability(Majority vote of 100 people being correct)?

Let BinCDF(k, n, p) be the CDF at value k of the Binomial distribution corresponding to n trials and each trial having probability p of success

BinCDF(k, n, p) = if I flip n coins, each of which is "heads" w.p. p, what is probability that # heads ≤ k.

Probability(Majority vote of 100 people being correct) = 1 – BinCDF(50,100,0.6)
≈ 0.97

Bagging: example

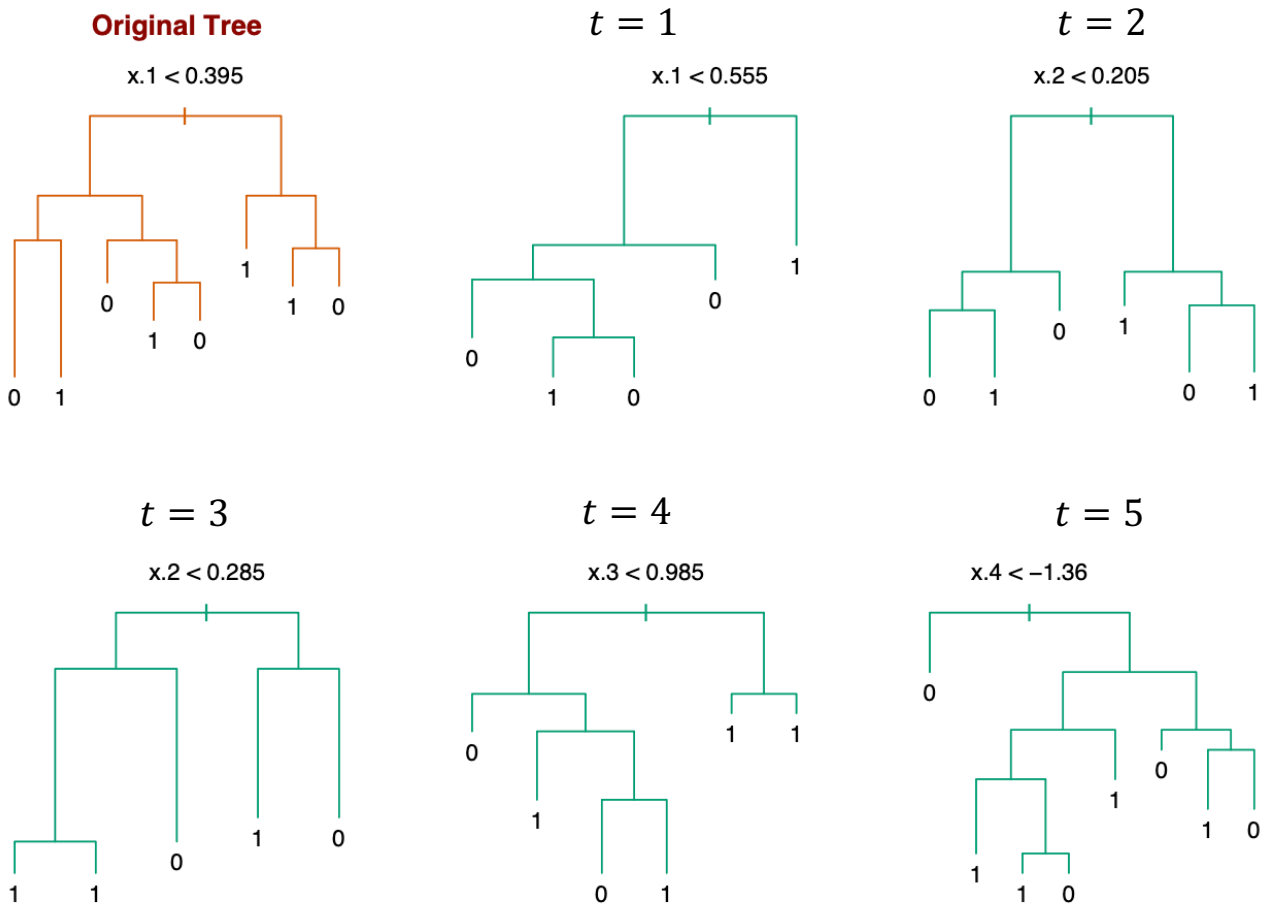


Figure from ESL

Bagging: summary

- Reduces overfitting (i.e., variance)
- Can work with any type of classifier (here focus on trees)
- Easy to parallelize (can train multiple trees in parallel)
- But loses on interpretability to single decision tree (true for all ensemble methods..)

training for one tree has nothing to do with training for others

Ensemble methods

- Bagging
- Random forests
- Boosting: Basics
- Adaboost
- Gradient boosting

Random forests

Issue with bagging: **Bagged trees are still too correlated**

Each is trained on large enough random sample of data and often end up not being sufficiently different.

How to decorrelate the trees further?

Simple technique: When growing a tree on a bootstrapped (i.e. subsampled) dataset, before each split select $k \leq d$ of the d input variables at random as candidates for splitting.

When $k = d \rightarrow$ same as Bagging
When $k < d \rightarrow$ Random forests

k is a hyperparameter, tuned via a validation set.

Random forests

Random forests are very popular!

Wikipedia: *Random forests are frequently used as "blackbox" models in businesses, as they generate reasonable predictions across a wide range of data while requiring little configuration.*

Issues:

- When you have large number of features, yet very small number of *relevant features*:
  Prob(selecting a relevant feature among k selected features) is very small

- Lacks expressive power compared to other ensemble methods we'll see next..

Ensemble methods

- Bagging
- Random forests
- Boosting: Basics
- Adaboost
- Gradient boosting

Boosting

Recall that the bagged/random forest classifier is given by

$$f_{agg}(\boldsymbol{x}) = \text{sign}\left(\frac{1}{T}\sum_{t=1}^{T} f_t(\boldsymbol{x})\right)$$

where each $\{f_t\}_{t=1}^T$ belongs to the function class $\mathcal{F}$ (such as a decision tree), and is trained in parallel.

Instead of training the $\{f_t\}_{t=1}^T$ in parallel, what if we sequentially learn which models to use from the function class $\mathcal{F}$ so that they are together as accurate as possible?

More formally, what is the best classifier $\text{sign}\,(h(\boldsymbol{x}))$, where

$$h(\boldsymbol{x}) = \sum_{t=1}^{T} \beta_t f_t(\boldsymbol{x}) \text{ for } \beta_t \geq 0 \text{ and } f_t \in \mathcal{F}.$$

Boosting is a way of doing this.

Boosting

- is a meta-algorithm, which takes a *base algorithm* (classification algorithm, regression algorithm, ranking algorithm, etc) as input and boosts its accuracy
- main idea: combine weak "rules of thumb" (e.g. 51% accuracy) to form a highly accurate predictor (e.g. 99% accuracy)
- works very well in practice (especially in combination with trees)
- has strong theoretical guarantees

We will continue to focus on binary classification.

Boosting: example

Email spam detection:

- given a training set like:
  - ("Want to make money fast? ...", **spam**)
  - ("Viterbi Research Gist ...", **not spam**)
- first obtain a classifier by applying a base algorithm, which can be a rather simple/weak one, like decision stumps:
  - e.g. contains the word "money" $\Rightarrow$ spam
- reweigh the examples so that "difficult" ones get more attention
  - e.g. spam that doesn't contain the word "money"
- obtain another classifier by applying the same base algorithm:
  - e.g. empty "to address" $\Rightarrow$ spam
- repeat ...
- final classifier is the (weighted) majority vote of all weak classifiers

Base algorithm

A **base algorithm** $\mathcal{A}$ (also called weak learning algorithm/oracle) takes a training set S weighted by D as input, and outputs classifier $f \leftarrow \mathcal{A}(S, D)$

- this can be any off-the-shelf classification algorithm (e.g. decision trees, logistic regression, neural nets, etc)
- many algorithms can deal with a weighted training set (e.g. for algorithm that minimizes some loss, we can simply replace "total loss" by "weighted total loss")
- even if it's not obvious how to deal with weight directly, we can always resample according to D to create a new unweighted dataset

For unweighted data, loss on training set S = $\frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i)$

↳ these weights are modified for weighted data

e.g. suppose I have a weighted training set as input to decision tree, then for any node calculate the conditional entropy by taking weights into account.

Boosting: Idea

The boosted predictor is of the form $f_{boost}(\boldsymbol{x}) = \text{sign}(h(\boldsymbol{x}))$, where,

$$h(\boldsymbol{x}) = \sum_{t=1}^{T} \beta_t f_t(\boldsymbol{x}) \text{ for } \beta_t \geq 0 \text{ and } f_t \in \mathcal{F}.$$

$h(x)$ lies in a larger function class corresponding to boosting algorithm

The goal is to minimize $\ell(h(\boldsymbol{x}), y)$ for some loss function ℓ.

Q: We know how to find the best predictor in $\mathcal{F}$ on some data, but how do we find the best weighted combination $h(\boldsymbol{x})$?

A: Minimize the loss by a *greedy approach*, i.e. find β_t, $f_t(\boldsymbol{x})$ one by one for $t = 1, \ldots, T$.

Specifically, let $h_t(\boldsymbol{x}) = \sum_{\tau=1}^{t} \beta_\tau f_\tau(\boldsymbol{x})$. Suppose we have found $h_{t-1}(\boldsymbol{x})$, how do we find β_t, $f_t(\boldsymbol{x})$?

Find the β_t, $f_t(\boldsymbol{x})$ which minimizes the loss $\ell(h_t(\boldsymbol{x}), y)$.

Different loss function ℓ give different boosting algorithms.

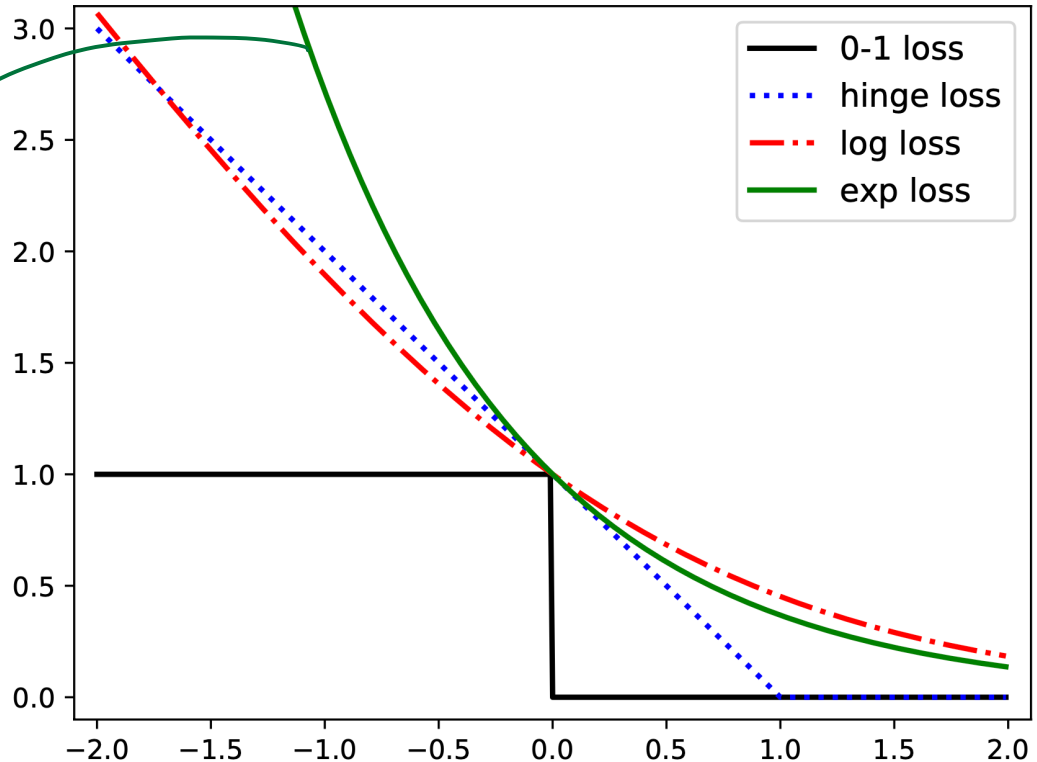
$$\ell(h(\boldsymbol{x}), y) = \begin{cases} (h(\boldsymbol{x}) - y)^2 & \rightarrow \text{ Least squares boosting,} \\ \exp(-h(\boldsymbol{x})y) & \rightarrow \textit{AdaBoost}. \end{cases}$$

Ensemble methods

- Bagging
- Random forests
- Boosting: Basics
- Adaboost
- Gradient boosting

AdaBoost

the exp loss penalizes being far away on the wrong side of decision boundary a lot more.



x-axis : $yh(x)$

Figure from PML

AdaBoost

AdaBoost minimizes exponential loss by a greedy approach, that is, find $\beta_t, f_t(\boldsymbol{x})$ one by one for $t = 1, \ldots, T$.

Recall $h_t(\boldsymbol{x}) = \sum_{\tau=1}^{t} \beta_\tau f_\tau(\boldsymbol{x})$. Suppose we have found h_{t-1}, *what should f_t be?* Greedily, we want to find $\beta_t, f_t(\boldsymbol{x})$ to minimize

using $h_t = h_{t-1} + \beta_t f_t$

$$\sum_{i=1}^{n} \exp\left(-y_i h_t(\boldsymbol{x}_i)\right) = \sum_{i=1}^{n} \exp\left(-y_i h_{t-1}(\boldsymbol{x}_i)\right) \exp\left(-y_i \beta_t f_t(\boldsymbol{x}_i)\right)$$

$$= \text{const}_t \cdot \sum_{i=1}^{n} D_t(i) \exp\left(-y_i \beta_t f_t(\boldsymbol{x}_i)\right)$$

where the last step is by defining the weights

plug in to verify

$$D_t(i) = \frac{\exp\left(-y_i h_{t-1}(\boldsymbol{x}_i)\right)}{\text{const}_t}$$

const_t is a normalizing constant to make $\sum_{i=1}^{n} D_t(i) = 1$. → to get a distribution

AdaBoost

So the goal becomes finding $\beta_t, f_t(\boldsymbol{x}) \in \mathcal{F}$ that minimize

$$\begin{aligned}
&\sum_{i=1}^{n} D_t(i) \exp\left(-y_i \beta_t f_t(\boldsymbol{x}_i)\right) \\
&= \sum_{i: y_i \neq f_t(\boldsymbol{x}_i)} D_t(i) e^{\beta_t} + \sum_{i: y_i = f_t(\boldsymbol{x}_i)} D_t(i) e^{-\beta_t} \\
&= \epsilon_t e^{\beta_t} + (1 - \epsilon_t) e^{-\beta_t} \qquad \text{(where } \epsilon_t = \textstyle\sum_{i: y_i \neq f_t(\boldsymbol{x}_i)} D_t(i) \text{ is } \textit{weighted error} \text{ of } f_t\text{)} \\
&= \epsilon_t (e^{\beta_t} - e^{-\beta_t}) + e^{-\beta_t}
\end{aligned}$$

Therefore, we should find $f_t(\boldsymbol{x})$ to minimize its the weighted classification error ϵ_t (what we expect the base algorithm to do intuitively).

AdaBoost

When $f_t(\boldsymbol{x})$ (and thus ϵ_t) is fixed, we then find β_t to minimize

$$\epsilon_t(e^{\beta_t} - e^{-\beta_t}) + e^{-\beta_t}$$

Exercise: verify that the solution is given by:

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

Hint: e^x is a convex function of x.

AdaBoost

How do we update the weights for the next step? The definition of $D_{t+1}(i)$ is actually recursive,

$h_t = h_{t-1} + \beta_t f_t$

$$
\begin{aligned}
D_{t+1}(i) &= \frac{\exp\left(-y_i h_t(\boldsymbol{x}_i)\right)}{\text{const}_{t+1}} \\
&= \frac{\exp\left(-y_i h_{t-1}(\boldsymbol{x}_i)\right)}{\text{const}_{t+1}} \cdot \exp\left(-y_i \beta_t f_t(\boldsymbol{x}_i)\right) \\
&= \left(D_t(i) \frac{\text{const}_t}{\text{const}_{t+1}}\right) \cdot \exp\left(-y_i \beta_t f_t(\boldsymbol{x}_i)\right)
\end{aligned}
$$

$D_t(i)$ ← const_t · const_t

$$
\implies D_{t+1}(i) \propto D_t(i) \exp(-\beta_t y_i f_t(\boldsymbol{x}_i)) = \begin{cases} D_t(i) e^{-\beta_t} & \text{if } f_t(\boldsymbol{x}_i) = y_i \\ D_t(i) e^{\beta_t} & \text{else} \end{cases}
$$

exponentially decrease / increase weights on correctly / incorrectly classified points

AdaBoost: Full algorithm

Given a training set S and a base algorithm $\mathcal{A}$, initialize D_1 to be uniform

For $t = 1, \ldots, T$

- obtain a weak classifier $f_t(\boldsymbol{x}) \leftarrow \mathcal{A}(S, D_t)$
- calculate the weight β_t of $f_t(\boldsymbol{x})$ as

$$\beta_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \qquad (\beta_t > 0 \Leftrightarrow \epsilon_t < 0.5)$$

  where $\epsilon_t = \sum_{i: f_t(\boldsymbol{x}_i) \neq y_i} D_t(i)$ is the weighted error of $f_t(\boldsymbol{x})$.

this condition says that the base algorithm should do something non-trivial

- update distributions

$$D_{t+1}(i) \propto D_t(i) e^{-\beta_t y_i f_t(\boldsymbol{x}_i)} = \begin{cases} D_t(i) e^{-\beta_t} & \text{if } f_t(\boldsymbol{x}_i) = y_i \\ D_t(i) e^{\beta_t} & \text{else} \end{cases}$$

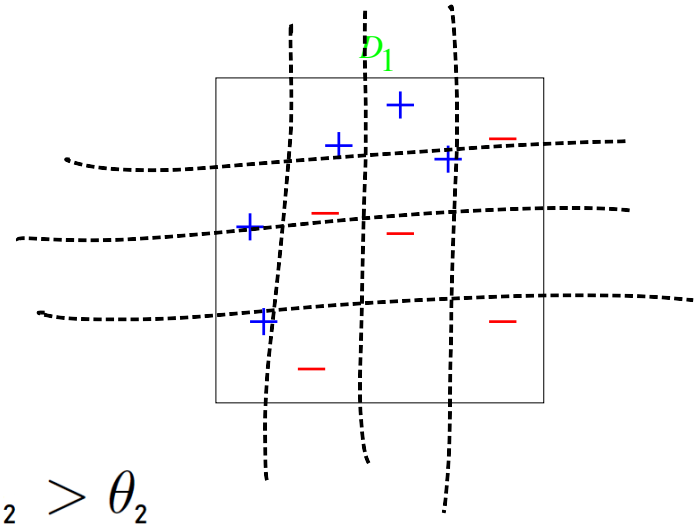
Output the final classifier $f_{boost} = \mathsf{sgn}\left(\sum_{t=1}^{T} \beta_t f_t(\boldsymbol{x}) \right)$

Adaboost: Example

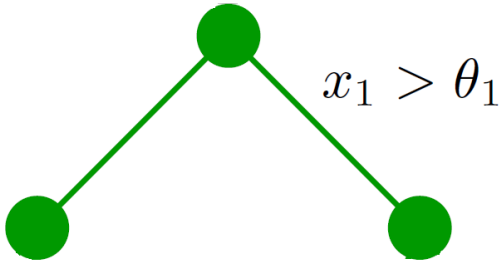
Put more weight on difficult to classify instances and less on those already handled well
New weak learners are added sequentially that focus their training on the more difficult patterns

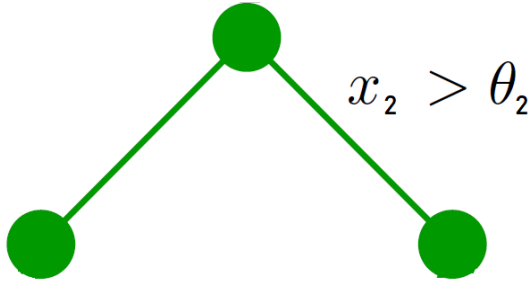
10 data points in $\mathbb{R}^2$

The size of + or - indicates the weight, which starts from uniform D_1



Base algorithm is decision stump:



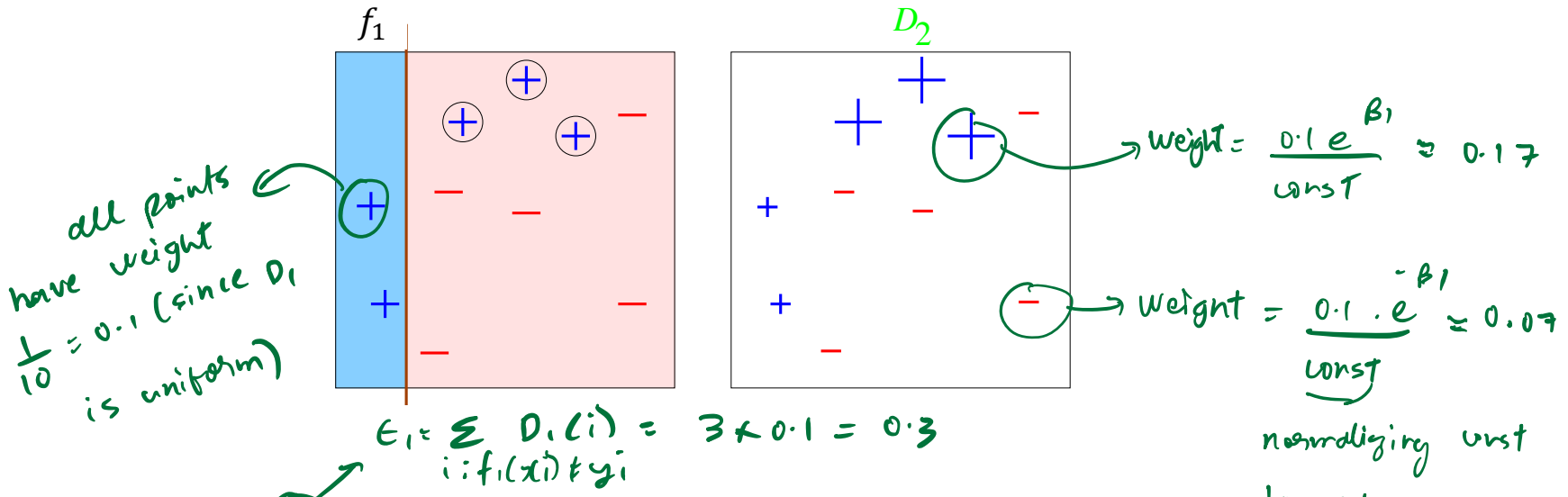


None of these stumps are individually good.

Observe that no stump can predict very accurately for this dataset.

Adaboost: Example

Put more weight on difficult to classify instances and less on those already handled well
New weak learners are added sequentially that focus their training on the more difficult patterns

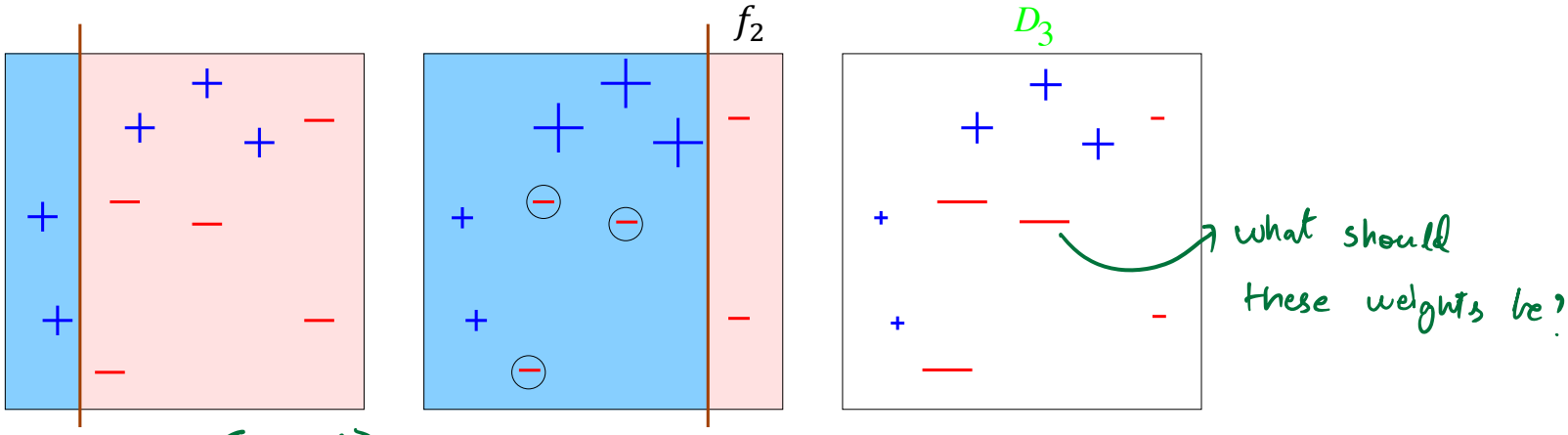


3 misclassified (circled): $\epsilon_1 = 0.3 \rightarrow \beta_1 = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) \approx 0.42.$

D_2 puts more weights on these misclassified points.

Adaboost: Example

Put more weight on difficult to classify instances and less on those already handled well
New weak learners are added sequentially that focus their training on the more difficult patterns



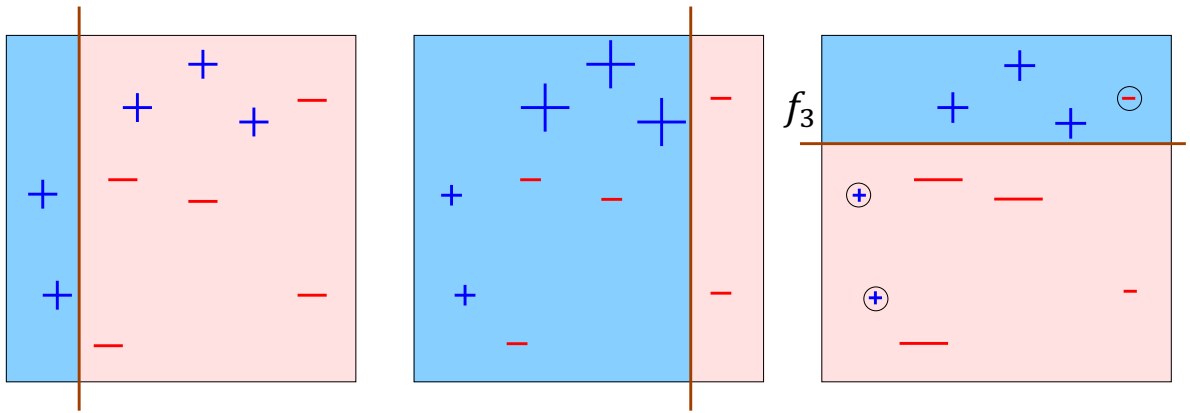
$\epsilon_2 = \sum_{i: f_2(i) \neq y_i} D_2(i) = 0.07 + 3 = 0.21$

3 misclassified (circled): $\epsilon_2 = 0.21 \rightarrow \beta_2 = 0.65$.

D_3 puts more weights on these misclassified points.

Adaboost: Example

Put more weight on difficult to classify instances and less on those already handled well
New weak learners are added sequentially that focus their training on the more difficult patterns

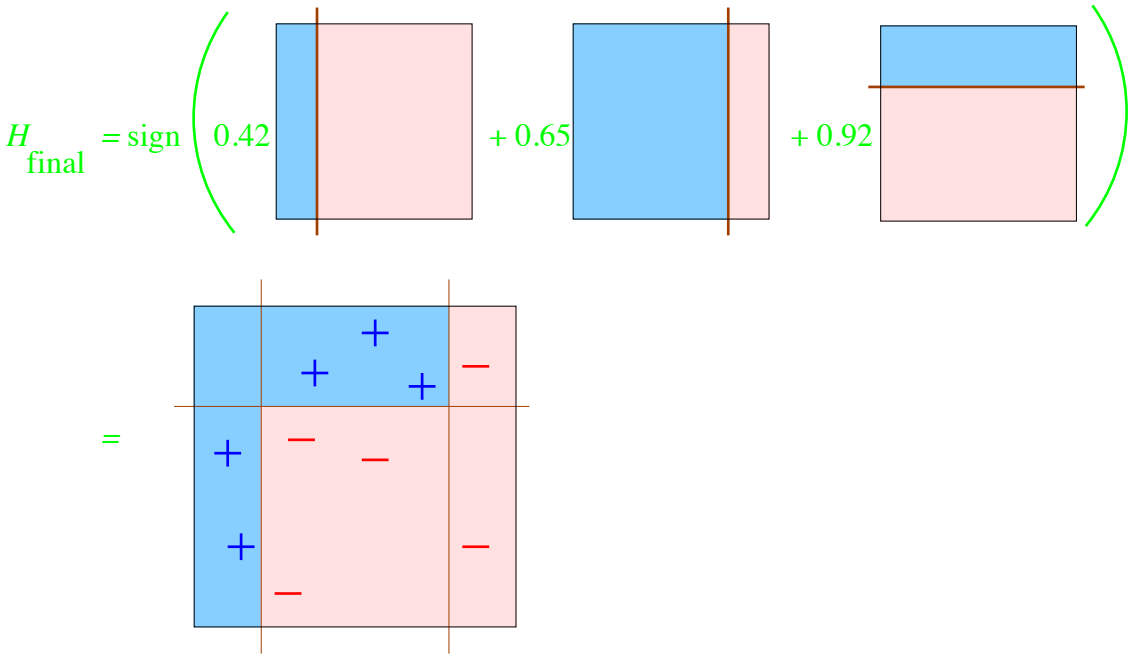


where is this coming from?

again 3 misclassified (circled): $\epsilon_3 = 0.14 \rightarrow \beta_3 = 0.92.$

Adaboost: Example

Put more weight on difficult to classify instances and less on those already handled well
New weak learners are added sequentially that focus their training on the more difficult patterns



All data points are now classified correctly, even though each weak classifier makes 3 mistakes.