Lecture 8

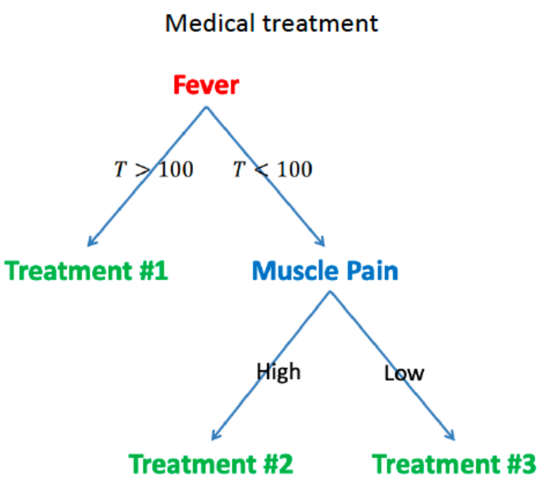
Decision Trees

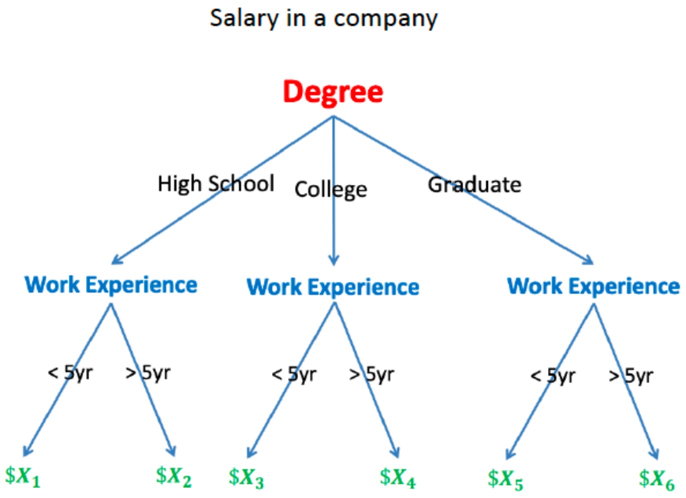
We have seen different ML models for classification/regression:

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

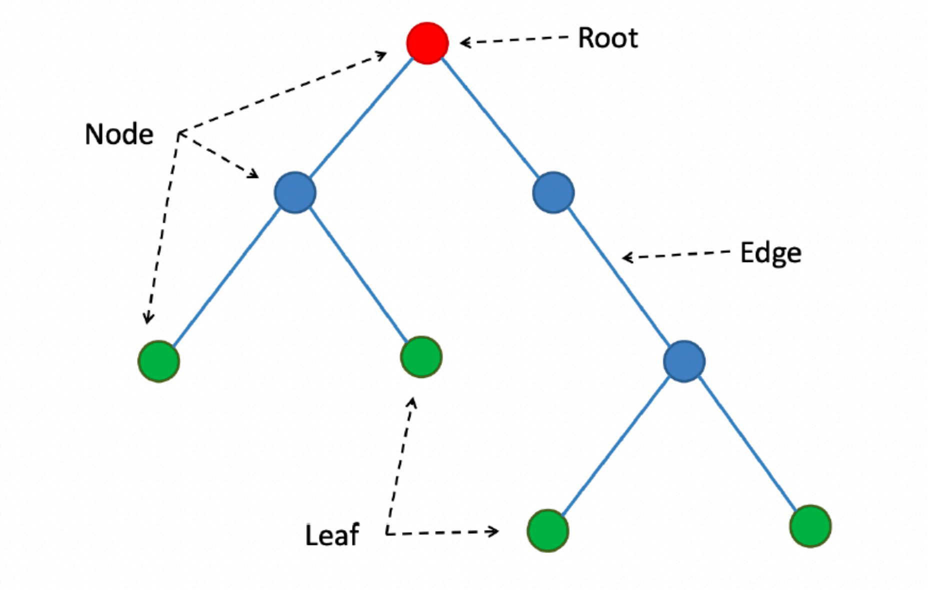
Decision tree is another popular one:

- 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.





Tree terminology



Input: $x = (x_1, x_2)$

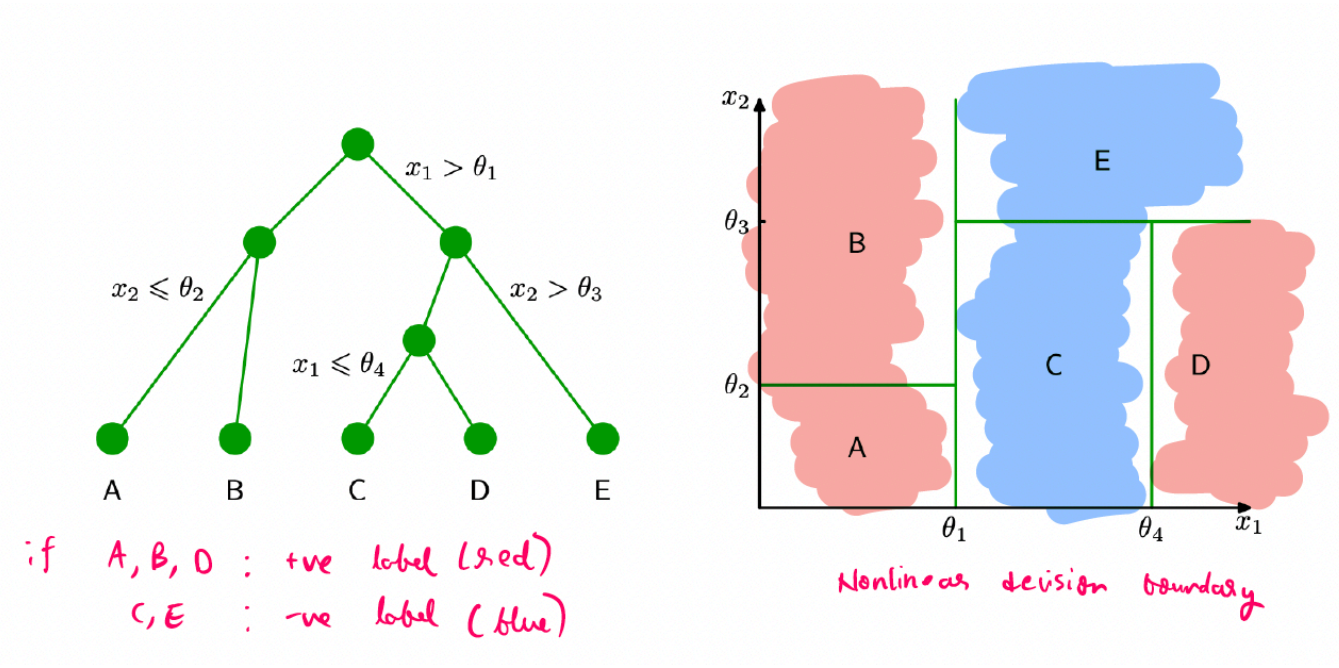
Output: $f(x)$ determined naturally by traversing the tree

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

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

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

Decision Bundary



Nonlinear decision boundary

Learning the 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 $(\theta_1, \theta_2, \cdots)$.

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

So how do we learn all these parameters?

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

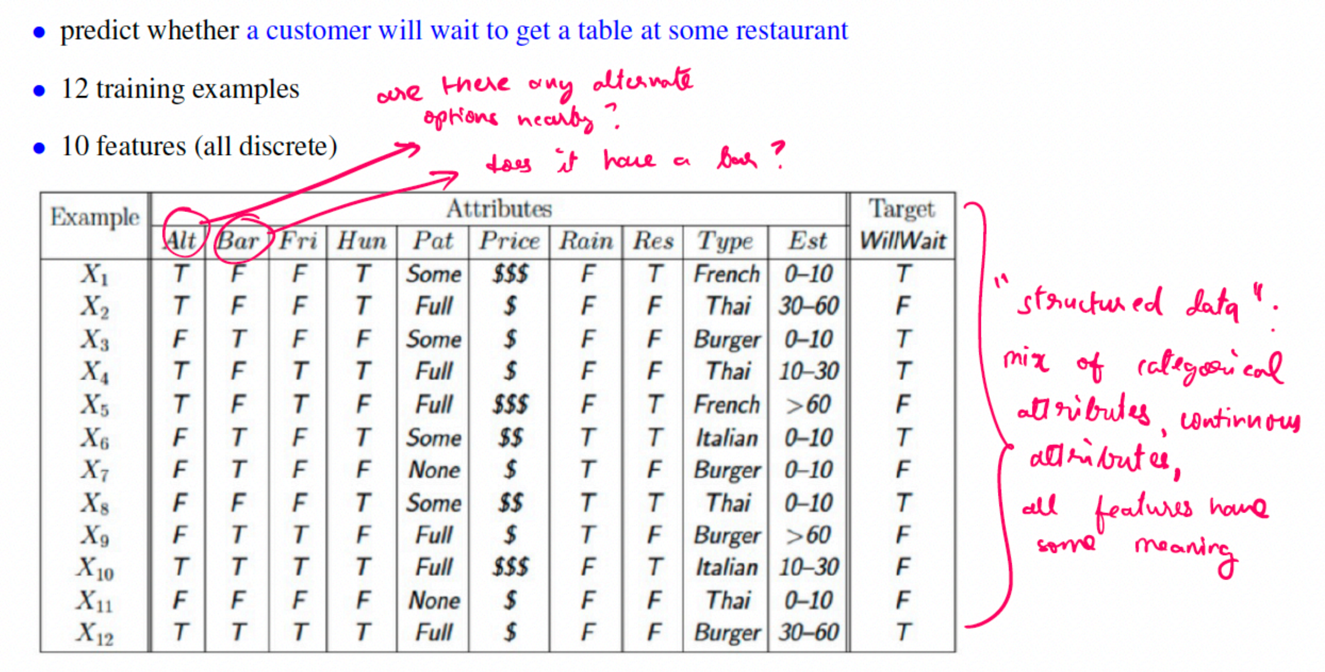
However, doing exact ERM is too expensive for trees.

- for T nodes, there are roughly $(\#features)^T$ (This is the size of function class $|\mathcal{F}|$) 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.

eg. cannot really use gradient descent to decide which feature to split on at any level!

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

An example: predict whether a customer will wait to get a table at some restaurant, 12 training examples, 10 features (all discrete).



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

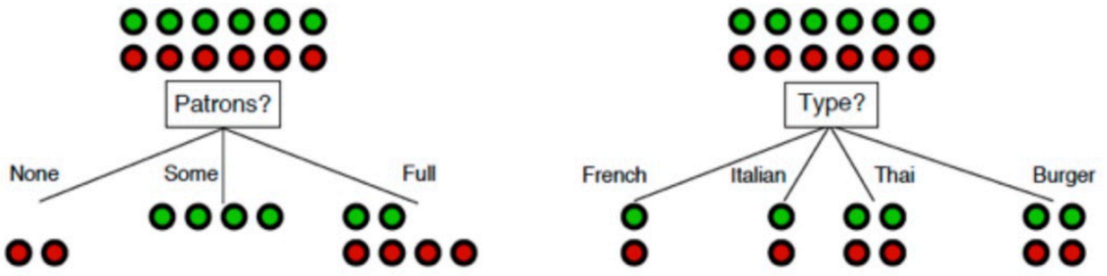
are there any alternate options nearby?

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

Which feature should we test at the root? Examples:



For type feature, all children are 50% / 50% pos/neg.

Intuitively "patrons" is a better feature since it leads to "more certain" children.

How to quantify?

Measure of uncertainty of a node (Entropy)

The uncertainty 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 uncertainty of a distribution is its (Shannon) entropy:

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

C is the classes.

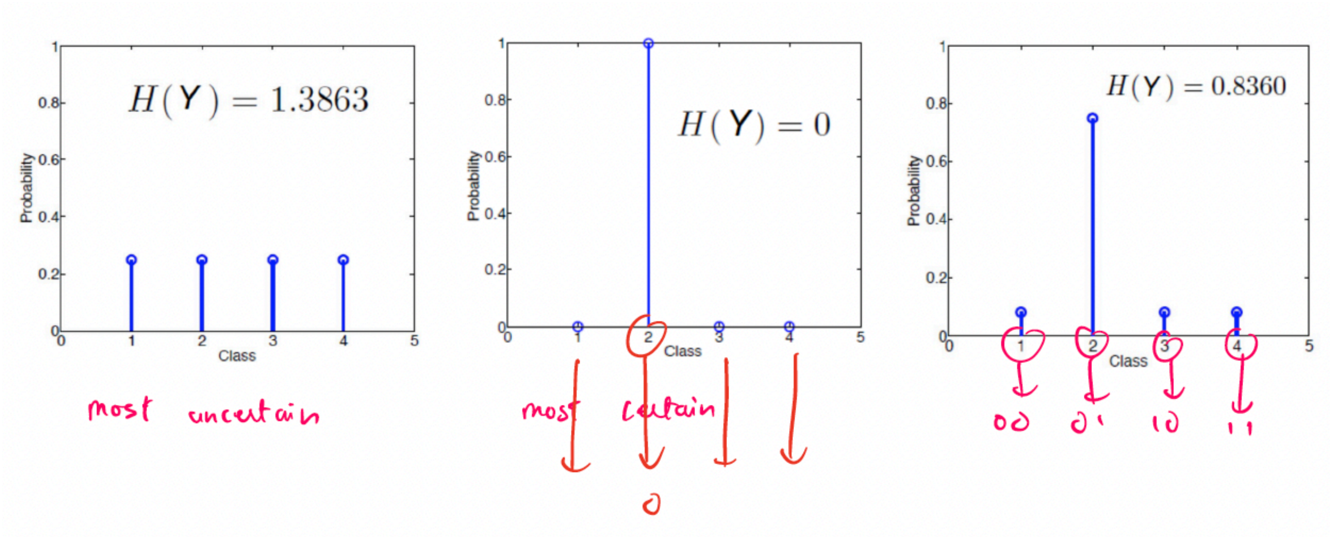
$$\begin{aligned} H(P) &= \mathbb{E}_{Y\sim P}\Big[\log(\frac{1}{P(Y)})\Big] \\ &= \sum_{k=1}^{C} P(Y=k)\log P(Y=k) \\ &= -\sum_{k=1}^{C} P(Y=k)\log P(Y=k) \end{aligned}$$

$\log(\frac{1}{P(Y)})$ is a measure of how unlikely the outcome was, $E_{Y\sim P}\Big[\log(\frac{1}{P(Y)})\Big]$ is overage "unlikeliness" that we sample outcomes from P .

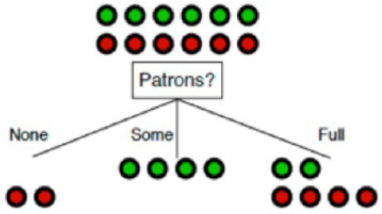
- the base of log can be $2, e\ or\ 10$.
- always non-negative.
- it’s the smallest codeword length to encode symbols drawn from P (assign shorter codes to outcomes which are more likely).
- maximized if P is uniform ($\max = \ln C$): most uncertain case
- minimized if P focuses on one class ($\min = 0$): most certain case

Thus, less entropy means more certain. We need to choose features that are more certain.

Examples of computing entropy



Entropy in each child if root tests on "patrons":



For None branch

$$-\Big(\frac{0}{0+2}\log\frac{0}{0+2}+\frac{2}{0+2}\log\frac{2}{0+2}\Big)=0 \quad i.e. -(green+red)$$

For Some branch

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

For Full branch

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

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:

$$\begin{aligned} H(Y|A) &= \sum_a P(A=a)H(Y|A=a) \\ &= \sum_a P(A=a)\Big(-\sum_{k=1}^{C} P(Y|A=a)\log P(Y|A=a)\Big) \\ &= \sum_a {''fraction\ of\ examples\ at\ node\ A=a''} \times {''entropy\ at\ node\ A=a''} \end{aligned}$$

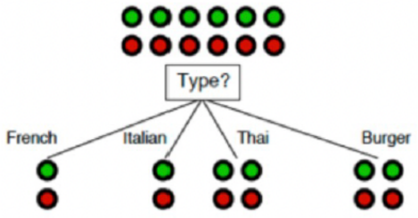
$H(Y|A=a)$ is the calculation from e.g. above.

Pick the feature that leads to the smallest conditional entropy.

Deciding the root

Intuition: Pick the feature that leads to the smallest conditional entropy.

Again, for "types":



For French and Italian branch

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

For Thai and Burger branch

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

From formula above, the conditional entropy is

$$\frac{2}{12}\times 1+\frac{2}{12}\times 1+\frac{4}{12}\times 1+\frac{4}{12}\times 1=1>0.45$$

Remember that less entropy means more certain. We want features that are more certain (has less entropy).

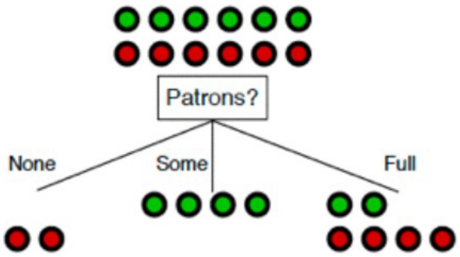
So splitting with "patrons" is better than splitting with "type". (We want more certainty - "patron")

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

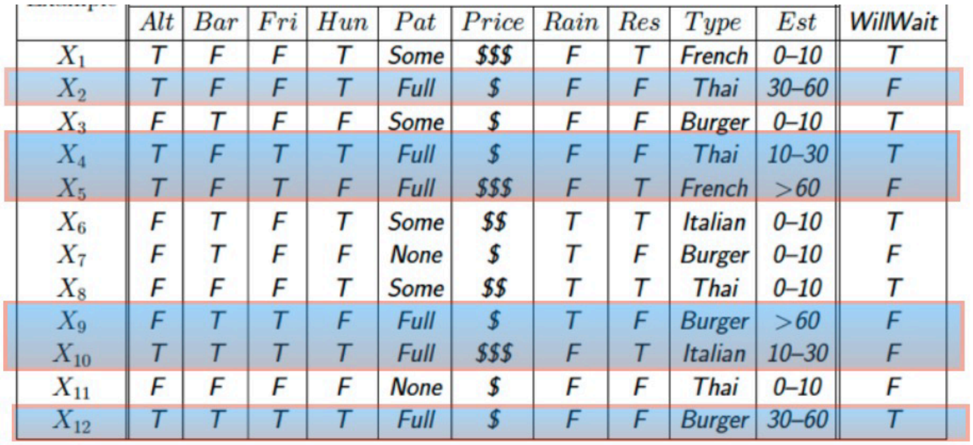
We are now done with building the root, this is also called a stump (a decision tree with only a root).

Repeat recursively

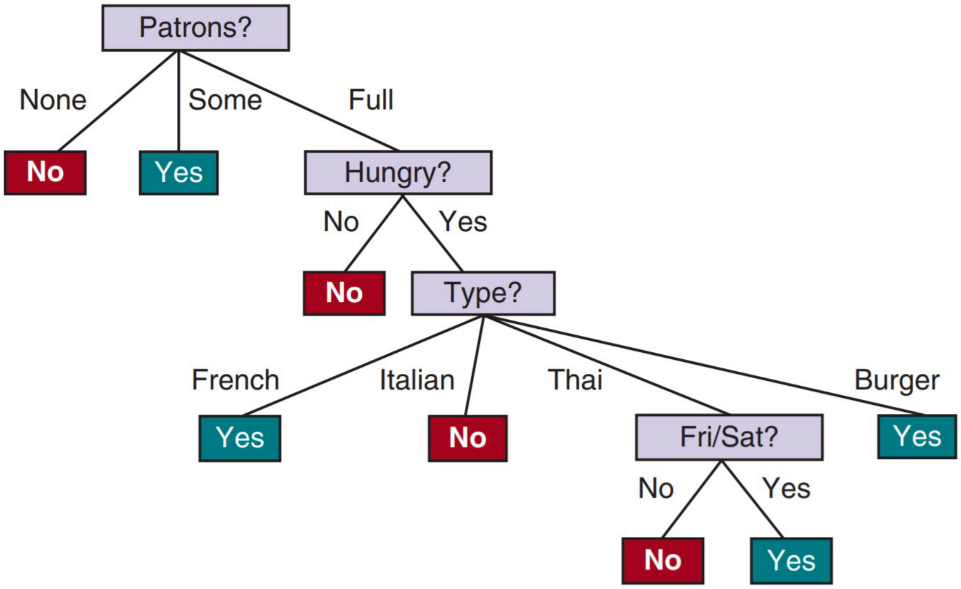
Split each child in the same way.



- 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 data examples:



Example	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



Full Learning Algorithm

We put the above process 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.

Replace entropy by Gini impurity:

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

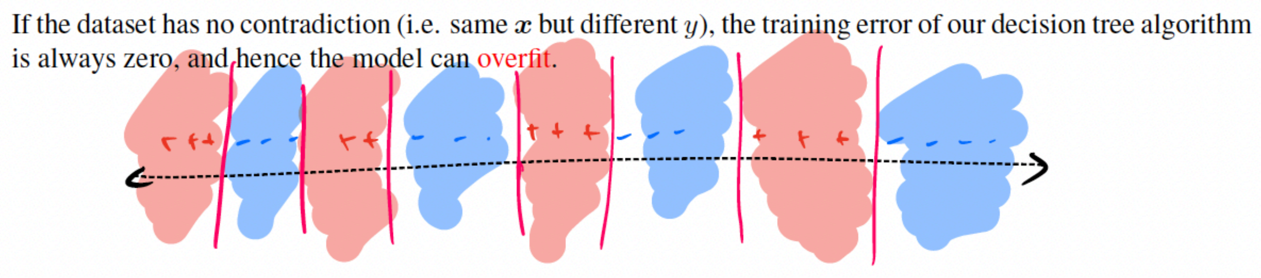
$P(Y = k)$ means pick example at random, $(1 - P(Y = k))$ means the prob of that same being misclassified if we predict its label based on the label of another randomly sampled point.

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 x but different y), the training error of our decision tree algorithm is always zero, and hence the model can overfit.



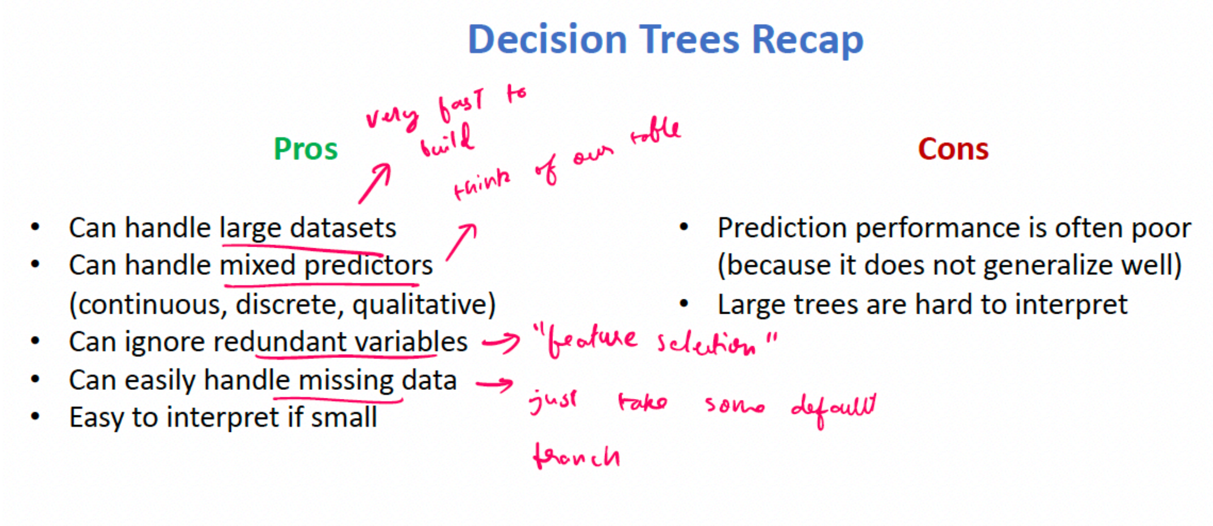
If the dataset has no contradiction (i.e. same x but different y), the training error of our decision tree algorithm is always zero, and hence the model can overfit.

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.

Ensemble Methods Intro

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



Decision Trees Recap

Pros

very fast to build

think of our table

- Can handle large datasets
- Can handle mixed predictors (continuous, discrete, qualitative)
- Can ignore redundant variables → "feature selection"
- Can easily handle missing data → just take some default branch
- 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 (a model whose predictions can vary a lot based on randomness in data), which can overfit.

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

Bootstrap Aggregating: 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.

Detail

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

$$f_{agg}(x) = \begin{cases} \frac{1}{T}\sum_{t=1}^{T} f_t(x) \quad for\ regresson, \\ sign(\frac{1}{T}\sum_{t=1}^{T} f_t(x)) = Majority\ Vote\{f_t(x)\}_{t=1}^{T} \quad for\ classification. \end{cases}$$

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)?

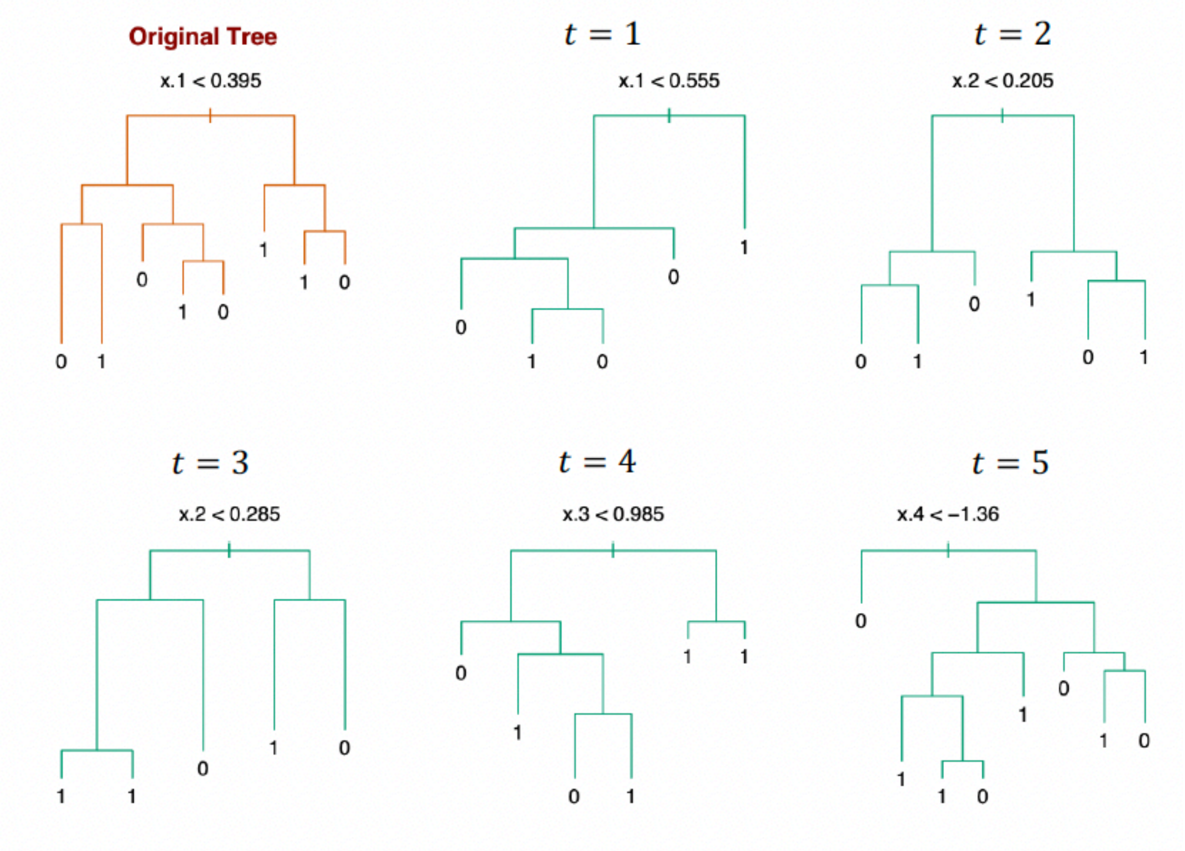
Let $BinCDF(k, n, p)$ be the CDF at value k of the Binomial distribution corresponding to n trials and each trial having probability p of success.

$BinCPF(k, n, p) =$ If I flip n coins, each of which is "heads" with probability p , what is probability that #heads $\in k$.

$$BinCDF(50, 100, 0.6) = C_{100}^{1} \cdot 0.6 \cdot 0.4^{99} + \cdots + C_{100}^{50} \cdot 0.6^{50} \cdot 0.4^{50} \approx 0.03$$

Probability(Majority vote of 100 people being correct) $= 1 - BinCDF(50, 100, 0.6) \approx 0.97$.

Example:



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). (Training for one tree has nothing to do with training for trees.)
- But loses on interpretability to single decision tree (true for all ensemble methods..)

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 (randomly choose part of features for training each tree.)

k is a hyperparameter, tuned via a validation set..

We have two randomness, the first is the data randomness from bootstrap, the second is the feature randomness from choosing feature randomly for each tree before splitting.

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.

Boosting

Recall that the bagged/random forest classifier is given by

$$f_{agg}(x) = sign\Big(\frac{1}{T}\sum_{t=1}^{T} f_t(x)\Big)$$

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? (parallel vs sequential)

More formally, what is the best classifier $sign(h(x))$, where

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

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

- 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")

  e.g. Suppose I have a weighted training set as input to decision tree, that for any node calculate the conditional entropy by taking weights into account.
- 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 $S = \frac{1}{n}\sum_{i=1}^{n} l(f(x_i), y_i)$. These weight are modified for weighted data.

Boosting: Idea

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

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

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

The goal is to minimize $l(h(x), y)$ 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 $h(x)$?

A: Minimize the loss by a greedy approach, i.e. find $\beta_t, f_t(x)$ one by one for $t = 1, \cdots, T$.

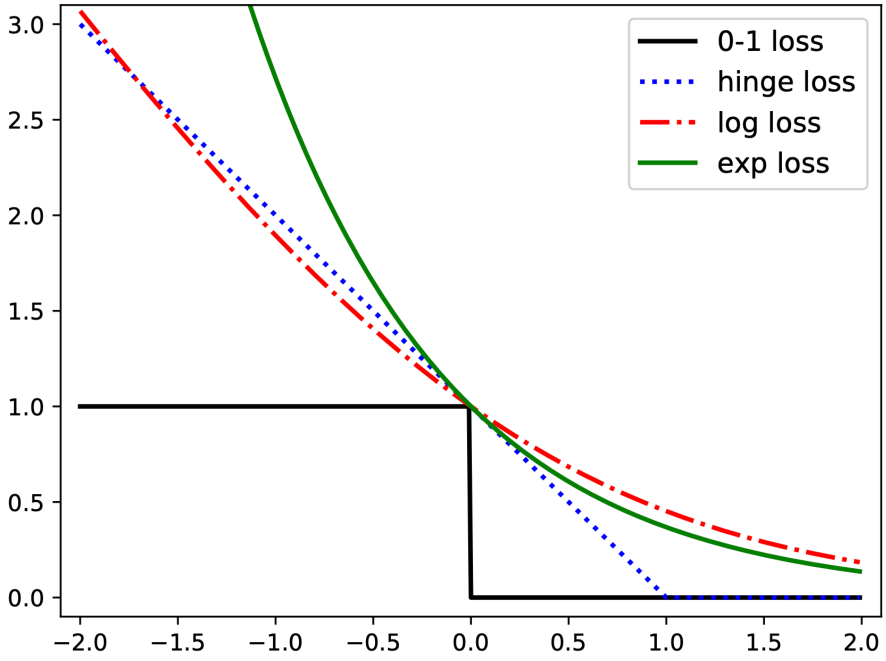
Specifically, let $h_t(x) = \sum_{\tau=1}^{t} \beta_\tau f_\tau(x)$. Suppose we have found $h_{t-1}(x)$, how do we find $\beta_t, f_t(x)$?

Find the $\beta_t, f_t(x)$ which minimizes the loss $l(h_t(x), y)$.

Different loss function l give different boosting algorithms:

$$l(h(x), y) = \begin{cases} (h(x) - y)^2 & \rightarrow \textit{Least squares boosting} \\ \exp(-h(x)y) & \rightarrow \textit{Adaboost} \end{cases}$$

AdaBoost



Input(x axis): $h(x)y$

The exp loss penalizes being from outlay on the wrong side of decision boundary a lot more.

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

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

$$\begin{aligned} \sum_{i=1}^{n} \exp(-y_i h_t(x_i)) &= \sum_{i=1}^{n} \exp(-y_i h_{t-1}(x_i)) \exp(-y_i \beta_t f_t(x_i)) \quad (\textit{by using } h_t = h_{t-1} + \beta_t f_t) \\ &= \textit{const}_t \cdot \sum_{i=1}^{n} D_t(i) \exp(-y_i \beta_t f_t(x_i)) \end{aligned}$$

where the last step is by defining the weights:

$$D_t(i) = \frac{\exp(-y_i h_{t-1}(x_i))}{const_t}$$

This is the weight for data. $const_t$ is a normalizing constant to make $\sum_{i=1}^n D_t(i) = 1$ (To get a distribution).

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

$$\begin{aligned}
&\sum_{i=1}^n D_t(i) \exp(-y_i \beta_t f_t(x_i)) \\
&\quad = \sum_{i: y_i \neq f_i(x_i)} D_t(i) e^{\beta_t} + \sum_{i: y_i = f_i(x_i)} D_t(i) e^{-\beta_t} \quad (\because f(t) \in \{\pm 1\}) \\
&\quad = \epsilon_t e^{\beta_t} + (1-\epsilon_t) e^{-\beta_t} \quad (where\ \epsilon_t = \sum_{n: y_i \neq f_t(x_i)} D_t(i)\ is\ weighted\ error\ of\ f_t) \\
&\quad = \epsilon_t (e^{\beta_t} - e^{-\beta_t}) + e^{-\beta_t}
\end{aligned}$$

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

When $f_t(x)$ (and thus ϵ_t) is fixed, we then find β_t to minimize:

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

the solution is given by:

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

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

$$\begin{aligned}
D_{t+1}(i) &= \frac{\exp(-y_i h_t(x_i))}{const_{t+1}} \\
&= \frac{\exp(-y_i h_{t-1}(x_i))}{const_{t+1}} \cdot \exp(-y_i \beta_t f_t(x_i)) \quad (\because h_t = h_{t-1} + \beta_t f_t) \\
&= \left(D_t(i) \frac{const_t}{const_{t+1}} \right) \cdot \exp(-y_i \beta_t f_t(x_i))
\end{aligned}$$

$$\Rightarrow D_{t+1}(i) \propto D_t(i) \exp(-\beta_t y_i f_t(x_i)) = \begin{cases} D_t(i) e^{-\beta_t} & if\ f_t(x_i) = y_i \\ D_t(i) e^{\beta_t} & if\ f_t(x_i) \neq y_i \end{cases}$$

This exponentially decrease / increase weights on correctly / incorrectly classification.

AdaBoost: Full algorithm

Given a training set S and a base algorithm A, initialize D_1 to be uniform.

For $t = 1, \cdots, T$

- obtain a weak classifier $f_t(x) \leftarrow A(S, D_t)$

- calculate the weight β_t of $f_t(x)$ as

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

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

  $\beta_t > 0$ means the base algorithm should do something non-trivial.

- Update distributions

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

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

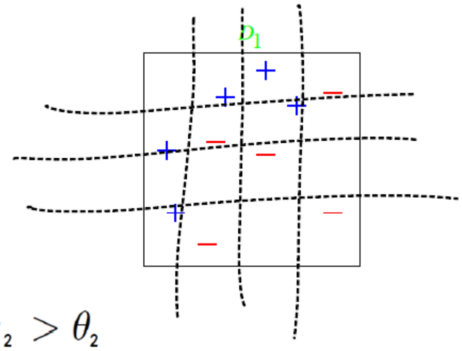
AdaBoost: Example

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

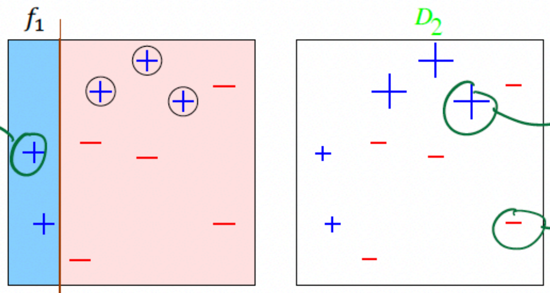


None of these stumps are individually good.

Base algorithm is decision stump:

$x_1 > \theta_1$
$x_2 > \theta_2$

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



all points have weight $\frac{1}{10} = 0.1$ (since D_1 is uniform)

weight = $\frac{0.1\, e^{\beta_1}}{\text{const}} \approx 0.17$

weight = $\frac{0.1 \cdot e^{-\beta_1}}{\text{const}} \approx 0.07$

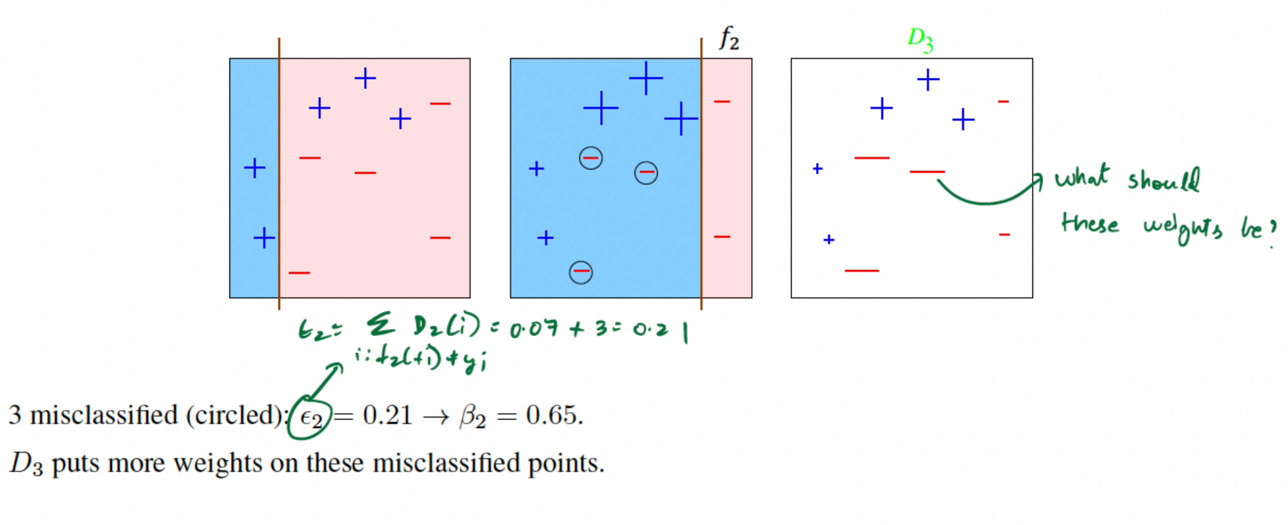
normalizing const to make D_2 a distribution

$\epsilon_1 = \sum_{i:f_1(x_i)\neq y_i} D_1(i) = 3 \times 0.1 = 0.3$

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

D_2 puts more weights on these misclassified points.

$$const_t = 0.1 \times e^{0.42} \times 3 + 0.1 \times e^{-0.42} \times 7$$

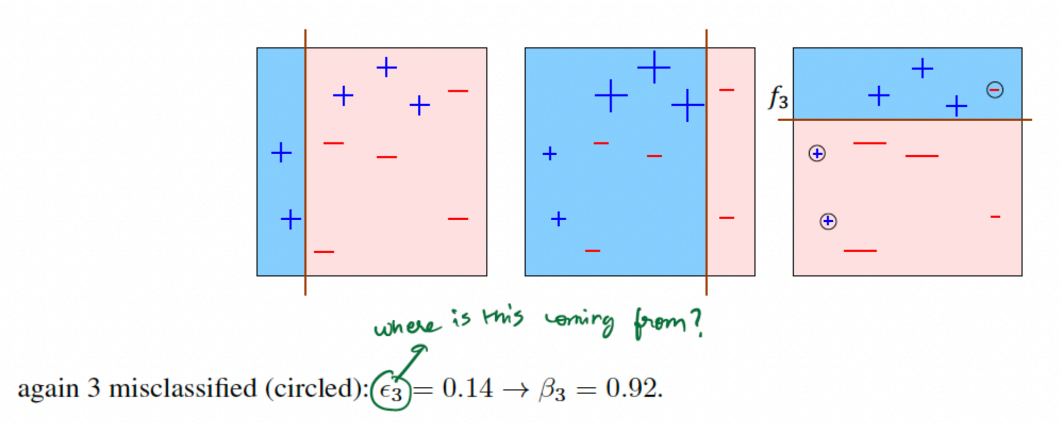


$\epsilon_2 = \sum_{i: f_2(i) \neq y_i} D_2(i) = 0.07 \times 3 = 0.21$

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

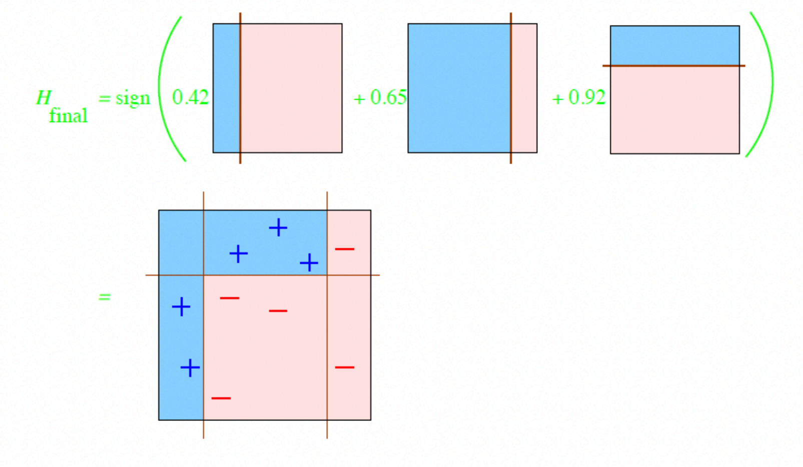
D_3 puts more weights on these misclassified points.

$$const_t = 0.07 \times e^{0.65} \times 3 + 0.07 \times e^{-0.65} \times 4 + 0.17 \times e^{-0.65} \times 3 \quad (3\ error\ -, 7\ correct\ \pm)$$



where is this coming from?

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



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

Gradient Boosting

Recall $h_t(x) = \sum_{\tau=1}^{t} \beta_\tau f_\tau(x)$, For Adaboost (exponential loss), given $h_{t-1}(x)$, we found what $f(t)$ should be.

Gradient boosting provides an iterative approach for general (any) loss function $l(h(x), y)$:

- For all training datapoints (x_i, y_i) find the gradient

  $$r_i = \left[\frac{\delta l(h(x_i), y_i)}{\delta h(x_i)}\right]_{h(x_i)=h_{t-1}(x_i)}$$

  r_i means how should predictions change "locally" to reduce loss. It is the gradient of the label, also the label for next step.

- Use the way learner to find f_t which fits (x_i, r_i) as well as possible:

  $$f_t = \arg\min \sum_{i=1}^{n} (r_i - f(x_i))^2$$

  r_i is what should be added to bring loss down, $f(x_i)$ means fit a model to x_i .

- Update $h_t(x) = h_{t-1}(x) + \eta f_t(x)$, for some step size η . $f_t(x)$ is to add model which improves loss locally.

Usually we add some regularization term to prevent overfitting (penalize the size of the tree etc.)

Gradient boosting is extremely successful!!

A variant XGBoost is one of the most popular algorithms for structured data (tables etc. with numbers and categories where each feature typically has some meaning, unlike images or text).

(for e.g. during Kaggle competitions back in 2015, 17 out of 29 winning solutions used XGBoost)