

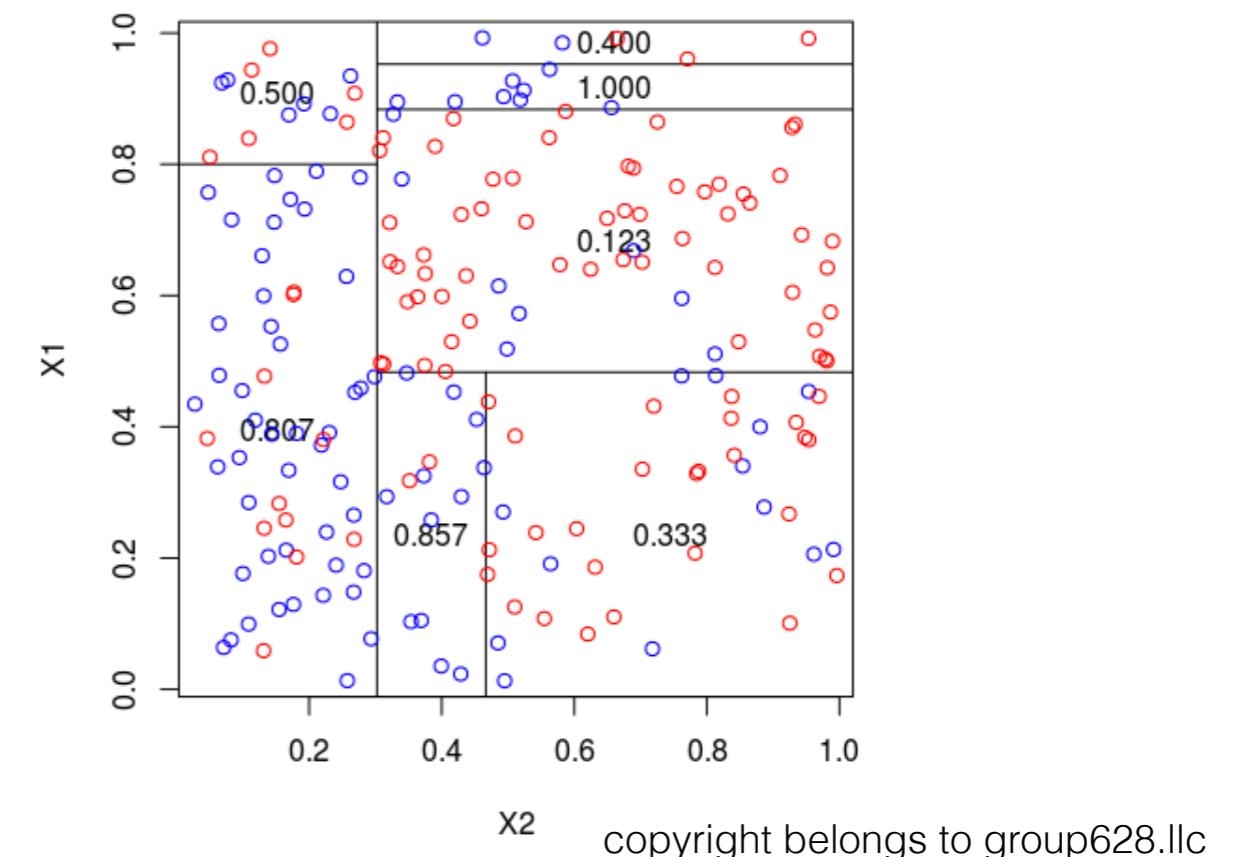
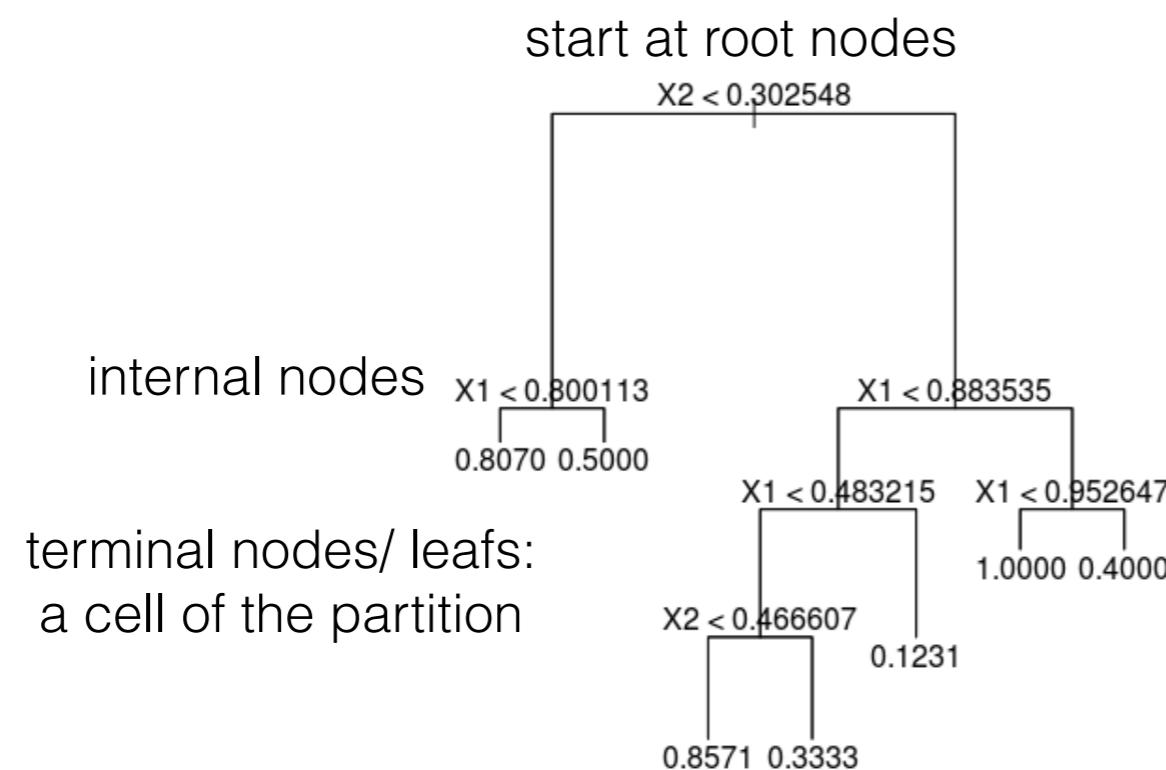
# Trees

# Outline

- Regression Trees
- Classification Trees
- Bagging: Averaging Trees
- Random Forest: Cleverer Averaging of Trees
- Boosting: Cleverest Averaging of Trees

# Regression Tree

- Nonlinear predictive model: When the data has lots of features which interact in complicated and nonlinear ways
- Recursive partitioning: partition the space into smaller regions, where the interactions are more manageable, then partition the sub-divisions again
- Two steps:
  - segment the predictor space to disjoint and non-overlapping regions
  - For every observation that falls to a specific region, predict the mean of the response values
  - That is, suppose the points  $(x_1, y_1), (x_2, y_2), \dots, (x_c, y_c)$  are all the samples belonging to the leaf-node  $i$ . Then our model for  $i$  is just  $\hat{y} = \frac{1}{c} \sum_{i=1}^c y_i$



# Regression Tree

This is a piecewise-constant model.

There are several advantages to this:

- Making predictions is fast (no complicated calculations, just looking up constants in the tree)
- It's easy to understand what variables are important in making the prediction (look at the tree)
- If some data is missing, we might not be able to go all the way down the tree to a leaf, but we can still make a prediction by averaging all the leaves in the sub-tree we do reach
- The model gives a jagged response, so it can work when the true regression surface is not smooth.
- There are fast, reliable algorithms to learn these trees

The algorithm has three main components:

- A way to select a split (the splitting rule).
- A rule to determine when a tree node is terminal (termination criterion).
- A rule for assigning a value to each terminal node.

# How to split?

- ultimate goal to minimize RSS: find the rectangle box  $R_1, \dots, R_j$  to minimize the squared difference between the response and the mean response for the training observation within the  $j$ th region

$$\sum_{j=1}^J \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

- at each split, recursive binary splitting: each inner node of these trees has two descendent nodes. These inner nodes split the training set into two subsets depending on the result of a test on one of the input variables.
- greatest reduction in the RSS based on predictor  $X$  and outpoint  $s$ 
  - if the condition is satisfied ( $X < s$ ), move down the left branch ( $R_1$ )
  - if the condition is not satisfied ( $X > s$ ), move down the right branch ( $R_2$ )

$$\sum_{i: x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i: x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

# Split on Continuous Variables

The best split  $s^*$  is the split belonging to S that maximizes the expression

$$\frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}}$$

where,  $S_L = \sum_{D_{t_L}} y_i$  and  $S_R = \sum_{D_{t_R}} y_i$

*Input :  $n_t$  cases, sum of their Y values (  $S_t$  ), the variable  $X_v$   
Output : The best cut-point split on  $X_v$*

Sort the cases according to their value in  $X_v$

$S_R = S_t; S_L = 0$

$n_R = n_t; n_L = 0$

$BestTillNow = 0$

FOR all instances  $i$  DO

$S_L = S_L + y_i; S_R = S_R - y_i$

$n_L = n_L + 1; n_R = n_R - 1$

    IF (  $\mathbf{x}_{i+1,v} > \mathbf{x}_{i,v}$  ) THEN *%Not trial if values are equal*

$NewSplitValue = (S_L^2 / n_L) + (S_R^2 / n_R)$

        IF (  $NewSplitValue > BestTillNow$  ) THEN

$BestTillNow = NewSplitValue$

$BestCutPoint = (\mathbf{x}_{i+1,v} + \mathbf{x}_{i,v}) / 2$

        ENDIF

    ENDIF

ENDFOR

- Sorting the observations by the values of the variable being examined
- Try these cut-point values as they are the only ones that may change the value of the score, because they modify the set of cases that go to the left and right branches.
- Evaluate all candidate splits, find the split that maximize the expression

# Split on Continuous Variables

The best split  $s^*$  is the split belonging to S that maximizes the expression

$$\frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}}$$

where,  $S_L = \sum_{D_{t_L}} y_i$  and  $S_R = \sum_{D_{t_R}} y_i$

- sort the observations by the values of the variable being examined  
sort x from smallest to largest: 1, 3, 5, 10  
corresponding y: 0.6, 2, 1, 1.5
- all observations are split to the right branch, no observation in the left branch

obs	x	y
1	1	0.6
2	5	1
3	3	2
4	10	1.5

$$S_L = 0 \quad S_R = 0.6 + 2 + 1 + 1.5 = 5.1 \quad \frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}} = 5.1^{**} 2 / 4 = 6.5025$$
$$n_L = 0 \quad n_R = 4$$

- when  $s=(1+3)/2=2$   
left branch has 1 observation, right branch has 3 observations

$$S_L = 0.6 \quad S_R = 2 + 1 + 1.5 = 4.5 \quad \frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}} = 0.6^{**} 2 / 1 + 4.5^{**} 2 / 3 = 7.11$$
$$n_L = 1 \quad n_R = 3$$

- when  $s=(3+5)/2=4$   
left branch has 2 observations, right branch has 2 observations

$$S_L = 0.6 + 2 = 2.6 \quad S_R = 1 + 1.5 = 2.5$$
$$n_L = 2 \quad n_R = 2$$

$$\frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}} = 2.6^{**} 2 / 2 + 2.5^{**} 2 / 2 = 4.815$$

# Split on Continuous Variables

The best split  $s^*$  is the split belonging to S that maximizes the expression

$$\frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}}$$

where,  $S_L = \sum_{D_{t_L}} y_i$  and  $S_R = \sum_{D_{t_R}} y_i$

obs	x	y
1	1	0.6
2	5	1
3	3	2
4	10	1.5

- when  $s=(5+10)/2=7.5$   
left branch has 3 observations, right branch has 1 observation
- $S_L = 0.6 + 2 + 1 = 3.6$      $S_R = 1.5$   
 $n_L = 3$                           $n_R = 1$
- $\frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}} = 3.6^2/3 + 1.5^2/1 = 6.57$
- Evaluate all candidate splits, find the split that maximize the expression  
 $s=2$

# Splits on Categorical Variables

The method suggested by Breiman et. al. (1984, p.247) involves an initial stage where the observations of the node are sorted as follows. Assuming that  $B$  is the set of values of  $X_v$  that occur in the current node  $t$ , and defining  $\bar{y}(b_i)$  as the average  $Y$  value of the observations having value  $b_i$  in variable  $X_v$ , sort the values such that

$$\bar{y}(b_1) \leq \bar{y}(b_2) \leq \dots \leq \bar{y}(b_{\#B})$$

The best split on discrete variable  $X_v$  in node  $t$  is one of the  $\#B-1$  splits

$$X_v \in \{b_1, b_2, \dots, b_h\}, h = 1, \dots, \#B-1$$

Suppose that we have the following instances in a node  $t$ :

COLOR	...	Y	leading to the averages
green	...	24	$\bar{y}(\text{green}) = (24 + 29 + 13)/3 = 22$
red	...	56	$\bar{y}(\text{red}) = (56 + 45)/2 = 50.5$
green	...	29	and $\bar{y}(\text{blue}) = (120 + 100)/2 = 110$
green	...	13	
blue	...	120	
red	...	45	
blue	...	100	

If we sort the values according to their respective average  $Y$  values we obtain the ordering  $\langle \text{green}, \text{red}, \text{blue} \rangle$ . According to Breiman's theorem the best split would be one of the  $\#B-1$  (in this case  $2 = 3-1$ ) splits, namely  $X_v \in \{\text{green}\}$  and  $X_v \in \{\text{green}, \text{red}\}$ .

The best split  $s^*$  is the split belonging to  $S$  that maximizes the expression

$$\frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}}$$

$$\text{where, } S_L = \sum_{D_{t_L}} y_i \text{ and } S_R = \sum_{D_{t_R}} y_i$$

# Splits on Categorical Variables

Suppose that we have the following instances in a node  $t$ :

<i>COLOR</i>	...	<i>Y</i>	leading to the averages
green	...	24	$\bar{y}(\text{green}) = (24 + 29 + 13)/3 = 22$
red	...	56	$\bar{y}(\text{red}) = (56 + 45)/2 = 50.5$
green	...	29	and $\bar{y}(\text{blue}) = (120 + 100)/2 = 110$
green	...	13	
blue	...	120	
red	...	45	
blue	...	100	

If we sort the values according to their respective average  $Y$  values we obtain the ordering  $\langle \text{green}, \text{red}, \text{blue} \rangle$ . According to Breiman's theorem the best split would be one of the #B-1 (in this case 2 = 3-1) splits, namely  $X_v \in \{\text{green}\}$  and  $X_v \in \{\text{green, red}\}$ .

- split 1:  $s = \text{'green'}$   
left branch has 3 observations, right branch has 4 observations

$$\begin{aligned} S_L &= 24 + 29 + 13 = 66 & S_R &= 56 + 45 + 120 + 100 = 321 & \frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}} &= 66^{**}2/3 + 321^{**}2/4 = 27212.25 \\ n_L &= 3 & n_R &= 4 & & \end{aligned}$$

- split 2:  $s = [\text{'green'}, \text{'red'}]$   
left branch has 5 observations, right branch has 2 observations

$$\begin{aligned} S_L &= 24 + 29 + 13 + 56 + 45 = 167 & S_R &= 120 + 100 = 220 & \frac{S_L^2}{n_{t_L}} + \frac{S_R^2}{n_{t_R}} &= 167^{**}2/5 + 220^{**}2/2 = 29777.8 \\ n_L &= 5 & n_R &= 2 & & \end{aligned}$$

- Evaluate all candidate splits, find the split that maximize the expression  
 $s = [\text{'green'}, \text{'red'}]$

# How to prevent overfitting?

- overfitting: each observation has its own terminal node, RSS is equal to 0
- A typical stopping criterion is to stop growing the tree when they would result in nodes containing less than, say, five percent of the total data.
- prune the trees (cost complexity tree pruning): consider T trees and choose the subtree that minimize

$$\sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

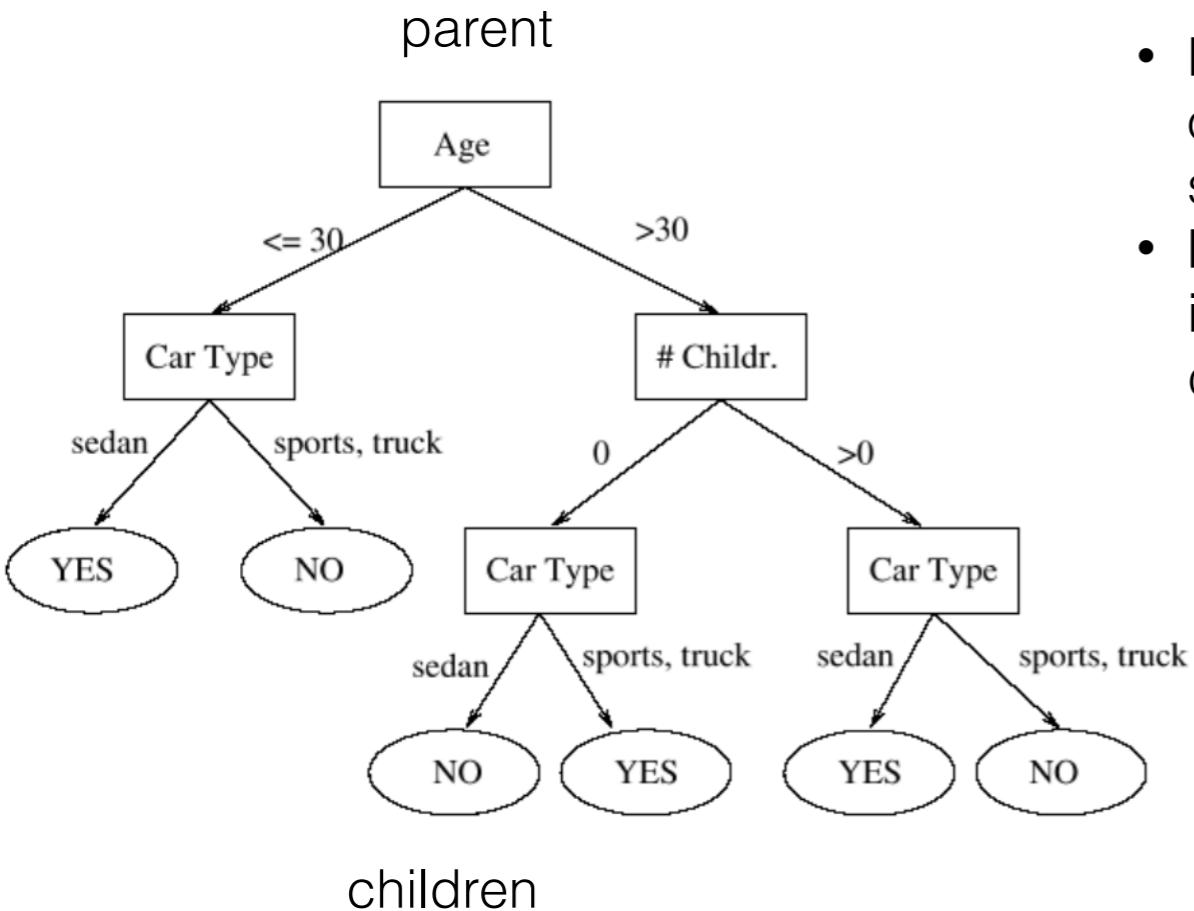
- balance the tradeoff between the complexity of the trees and fitness of the model
- $|T|$ : total number of terminal nodes in the subtree
- $\alpha$  : tuning parameter
- larger  $\alpha$ : have limited terminal nodes, smaller  $\alpha$ : have many terminal nodes

Procedure:

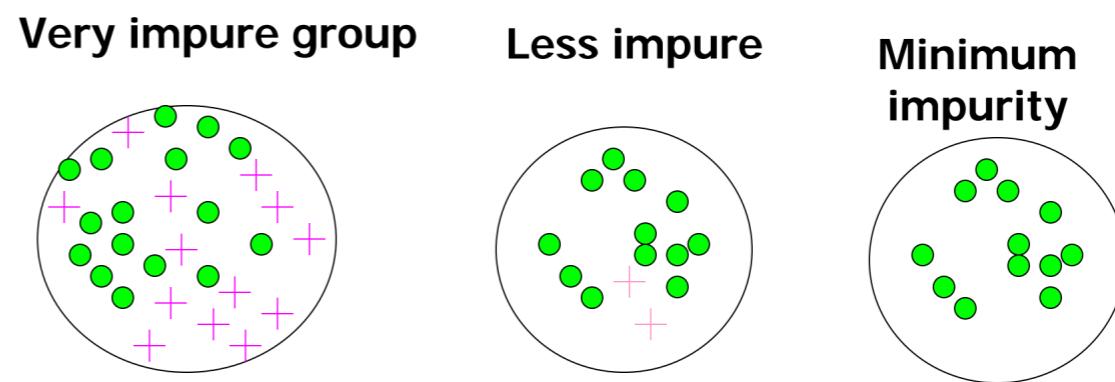
- use recursive binary split to build a large tree
- apply cost complexity pruning to get a sequence of subtree as function of  $\alpha$
- use k-fold cross validation to find the best  $\alpha$  that minimize the above function
- return the subtree that corresponds to the best  $\alpha$

# Classification trees

For every observation that falls to a specific leaf, predict the majority class of the response values



- How to select best split is based on the degree of impurity of the leafs. The smaller the degree of impurity, the more skewed the class distribution, the better of split.
- For example, a node with class distribution (0,1) has zero impurity (best children), whereas a node with uniform class distribution (0.5, 0.5) has highest impurity (worst children)



# Entropy: a common way to measure impurity

$$\text{Entropy} = \sum_i -p_i \log_2 p_i$$

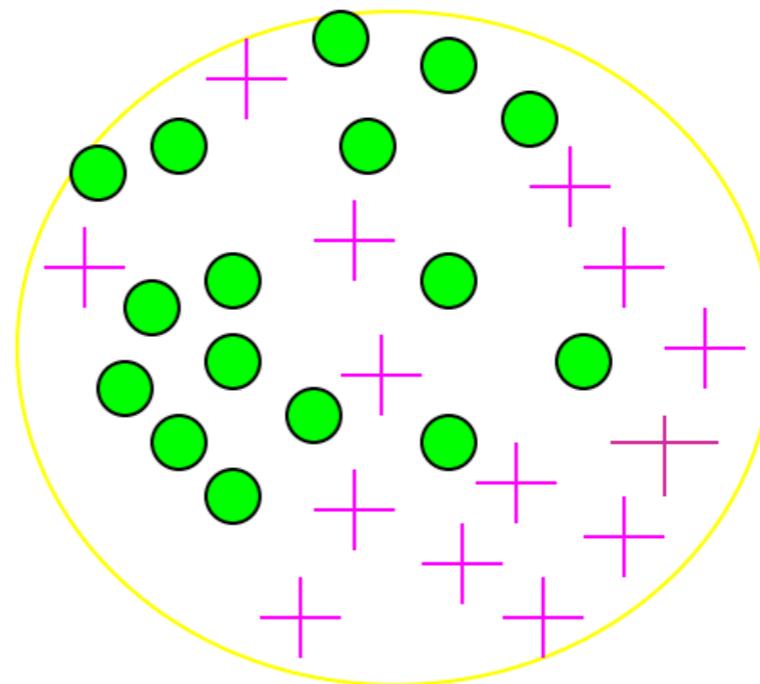
$p_i$  is the probability of class i

Compute it as the proportion of class i in the set.

16/30 are green circles; 14/30 are pink crosses

$\log_2(16/30) = -.9$ ;  $\log_2(14/30) = -1.1$

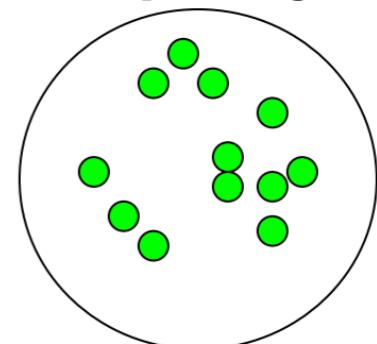
Entropy =  $-(16/30)(-.9) - (14/30)(-1.1) = .99$



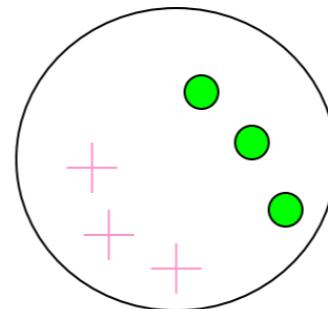
What is the entropy of a group in which all examples belong to the same class?

- entropy =  $-1 \log 1 = 0$
- best children
- not a good training set for learning (worst parent)

**Minimum impurity**



**Maximum impurity**



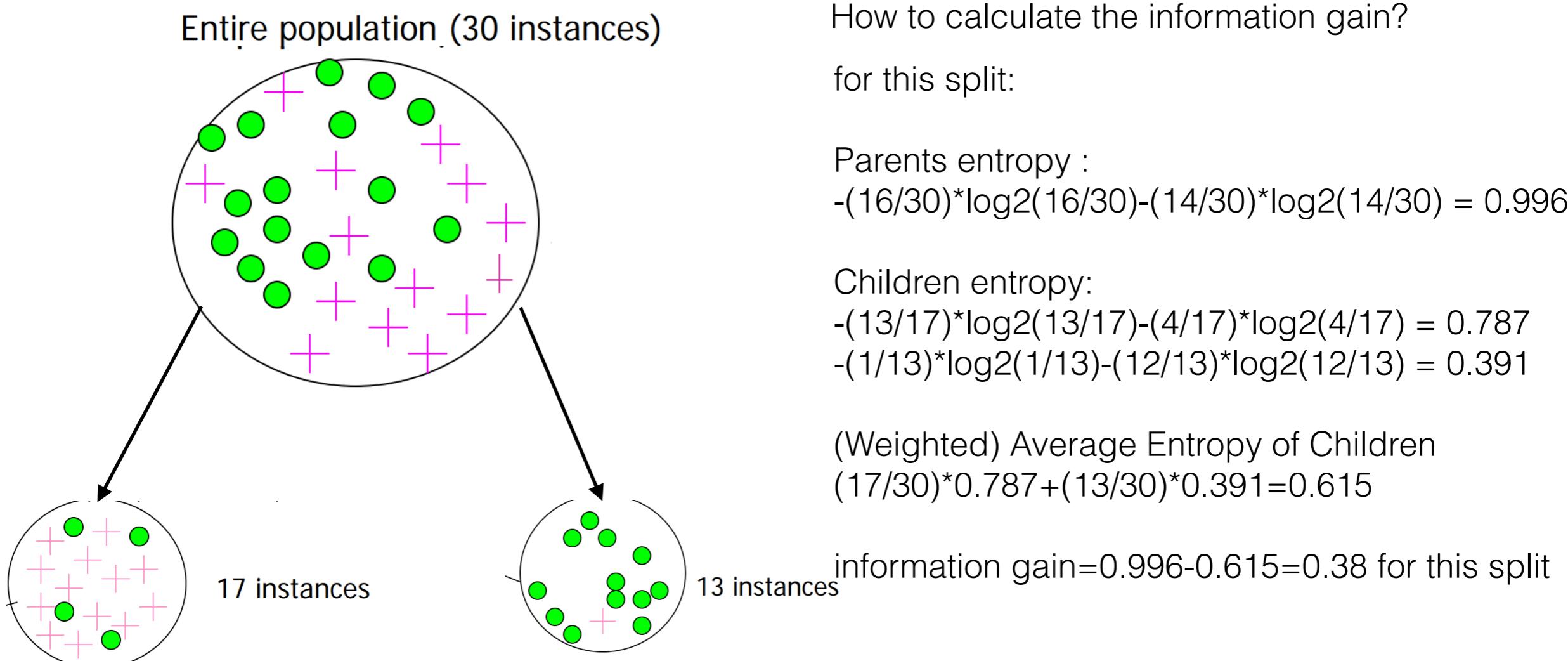
What is the entropy of a group with 50% in either class?

- entropy =  $-0.5 \log 0.5 - 0.5 \log 0.5 = 1$
- worst children
- good training set for learning (best parent)

# Information Gain

We want to determine which feature in a given training set is most useful for discriminating between the classes to be learned.

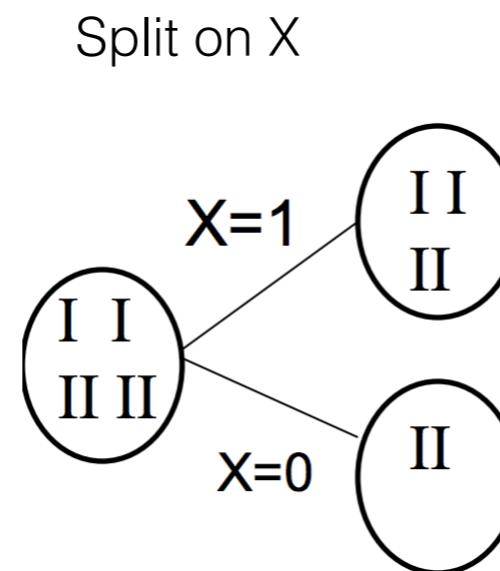
- Information gain tells us how important a given feature is
- We will use it to decide the ordering of feature in the nodes of a decision tree
- Information Gain =  $\text{entropy}(\text{parent}) - [\text{average entropy}(\text{children})]$



# Using Information Gain to Construct a Decision Tree

- Choose the feature with highest information gain (minimum average entropy) for the full training set at the root of the tree.

X	Y	Z	C
1	1	1	I
1	1	0	I
0	0	1	II
1	0	0	II



$$E_{\text{child}1} = -(1/3)\log_2(1/3) - (2/3)\log_2(2/3) = .9184$$

$$E_{\text{child}2} = 0$$

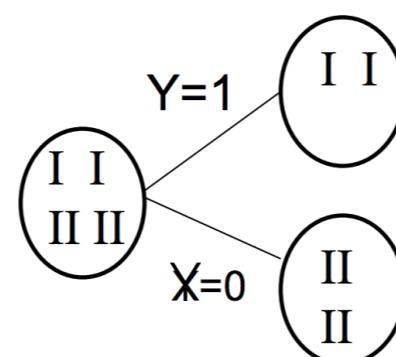
$$E_{\text{parent}} = 1$$

$$\text{GAIN} = 1 - (3/4)(.9184) - (1/4)(0) = .3112$$

# Using Information Gain to Construct a Decision Tree

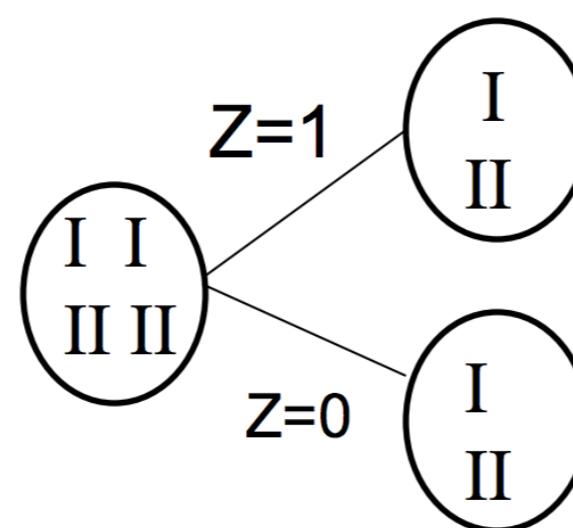
X	Y	Z	C
1	1	1	I
1	1	0	I
0	0	1	II
1	0	0	II

Split on Y



Echild1=0  
Echild2= 0  
Eparent= 1  
 $GAIN = 1 - (1/2)0 - (1/2)0 = 1$   
Best split

Split on Z



Echild1= 1  
Echild2= 1  
Eparent= 1  
 $GAIN = 1 - (1/2)(1) - (1/2)(1) = 0$   
Worst split

- conclusion: split on Y

# Entropy vs Gini Index

- For each terminal node, calculate the entropy

$$E(S) = -p_1 \log p_1 - p_2 \log p_2 - \dots - p_n \log p_n = -\sum_{i=1}^n p_i \log p_i$$

- Compute the weighted average over all terminal nodes resulting from the split, weight by their size
- Minimize average entropy

$$I(S, A) = \sum_i \frac{|S_i|}{|S|} \cdot E(S_i)$$

- Gini index: measure of terminal impurity (instead of entropy)
- It reaches its minimum (zero) when all cases in the node fall into a single target category.

$$Gini(S) = 1 - \sum_i p_i^2$$

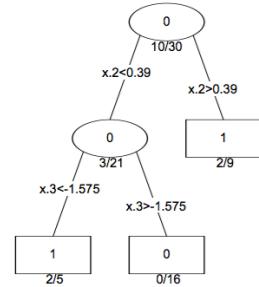
$p_i$  is the fraction of observations in each rectangle

- Minimize average Gini index (instead of average entropy)

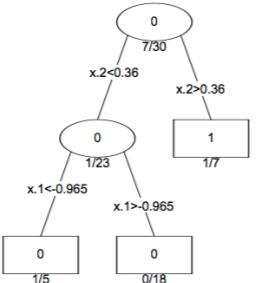
$$Gini(S, A) = \sum_i \frac{|S_i|}{|S|} \cdot Gini(S_i)$$

# Bagging

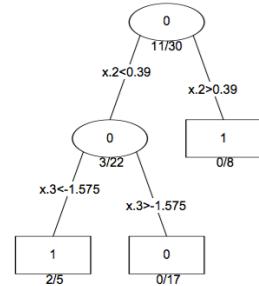
**Original Tree**



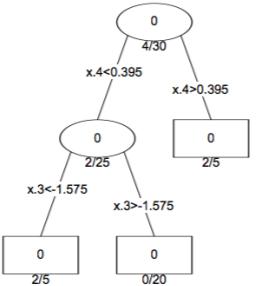
**Bootstrap Tree 1**



**Bootstrap Tree 2**



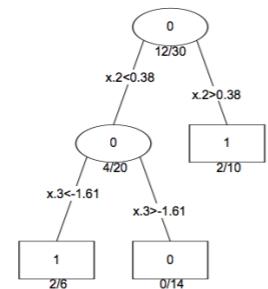
**Bootstrap Tree 3**



**Bootstrap Tree 4**



**Bootstrap Tree 5**



- bagging: averages a given procedure over many samples, reduce variance
- we have bagging regression tree and bagging classification tree
- procedure:
  - draw B bootstrap samples each of size (N/number of trees) with replacement from the training data
  - build many trees to train on the B<sup>th</sup> bootstrapped training set in order to get predictions for each observation
  - average all predictions or take the majority vote for each observation

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^B \hat{f}^{*b}(x)$$

Bagging can dramatically reduce the variance of unstable procedures of building one tree, leading to improved prediction. However any simple structure in the original data set is lost.

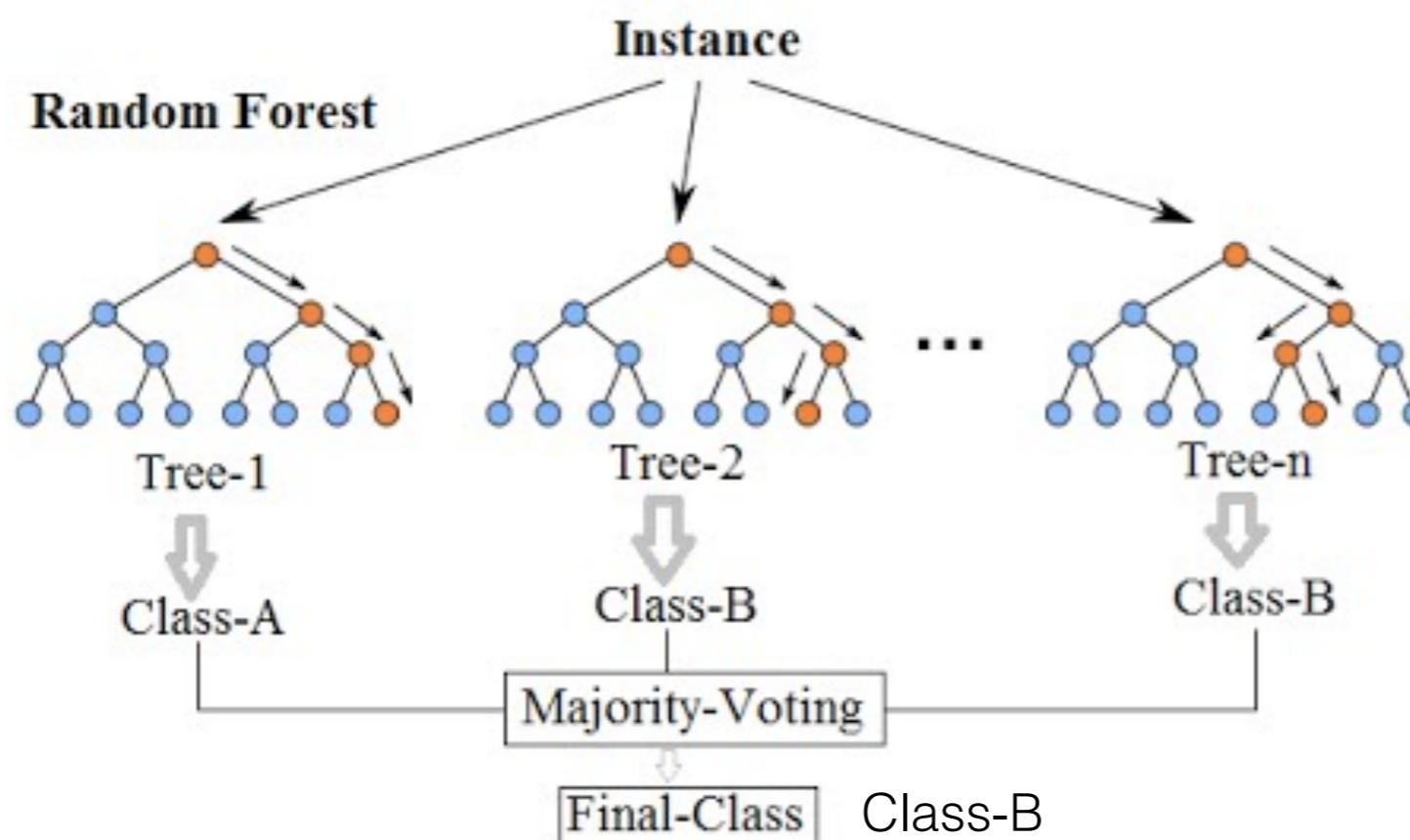
# Estimating test error

- testing error:
  - ‘in the bag’: the observations that are used to fit the tree, training error
  - ‘out of bag’: the observations that are not used to fit the tree, testing error
- calculate the testing error:
  - predict the response of a give observation at ‘out of bag’
  - average the result of all the trees
  - calculate out of bag error estimate
    - regression: RSS, MSE, RMSE ( $\sqrt{\text{MSE}}$ ), RSE
    - classification: misclassification rate, accuracy, roc\_auc

# Random Forest

- refinement of bagged trees, we have random forest regression tree and random forest classification tree
- procedure:
  - build various decision trees on bootstrapped training sample
  - each tree is constructed using a different bootstrap sample from the original data.
  - at each tree split, a random sample of  $m$  features is drawn, and only those  $m$  features are considered for splitting. Experience pint  $m=\sqrt{p}$ , where  $p$  is the number of features
  - For each tree grown on a bootstrap sample, the “out-of-bag” error rate is monitored

## Random Forest Simplified



# Random Forest

- random forests tries to improve on bagging by “de-correlating” the trees. Each tree has the same expectation
- it can never overfit by adding more trees
- it can handle missing values. If the  $m$ th variable is not categorical, the method computes the median of all values of this variable in class  $j$ , then it uses this value to replace all missing values of the  $m$ th variable in class  $j$ . If the  $m$ th variable is categorical, the replacement is the most frequent non-missing value in class  $j$ .

# Feature Importance

Will adding a predictor that has no (or has some) significative impact by itself to the set of features actually improve the model?

Several measures are available for feature importance:

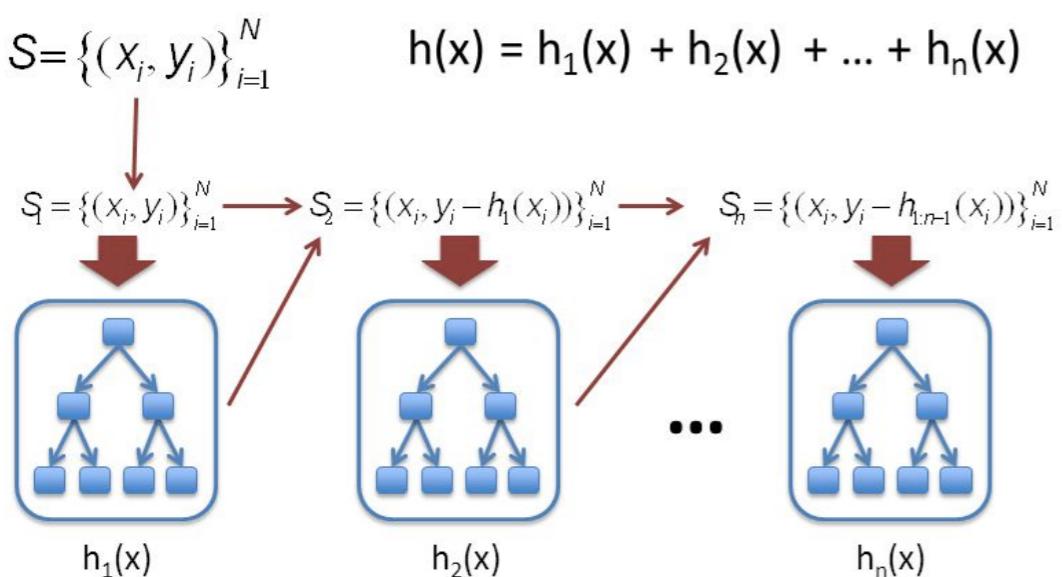
- RSS: RSS is decreased due to splits over a given predictor, averaged over all B trees. A large value indicates an important predictor.
- Gini Index: The Gini index is decreased by splits over a given predictor, averaged over all B trees. A large value indicates an important predictor.
- A relatively large value indicates a notable drop in the RSS or Gini index, and thus a better fit to the data; corresponding variables are relatively important predictors.

# Boosting

- Boosting is a sequential technique which works on the principle of ensemble. It combines a set of weak learners and delivers improved prediction accuracy.
- gradient boosting:
  - fit separate and dependent decision trees to each training set, each tree is generated using information from previous tree, and the addition of a new tree will improve the performance of the previous tree

## Gradient Boosting (Simple Version)

(For Regression Only)



Boosting regression tree: fit a new decision tree to the residuals of the current decision tree and add the new decision tree to the current decision tree, the residuals will be updated.

# Boosting

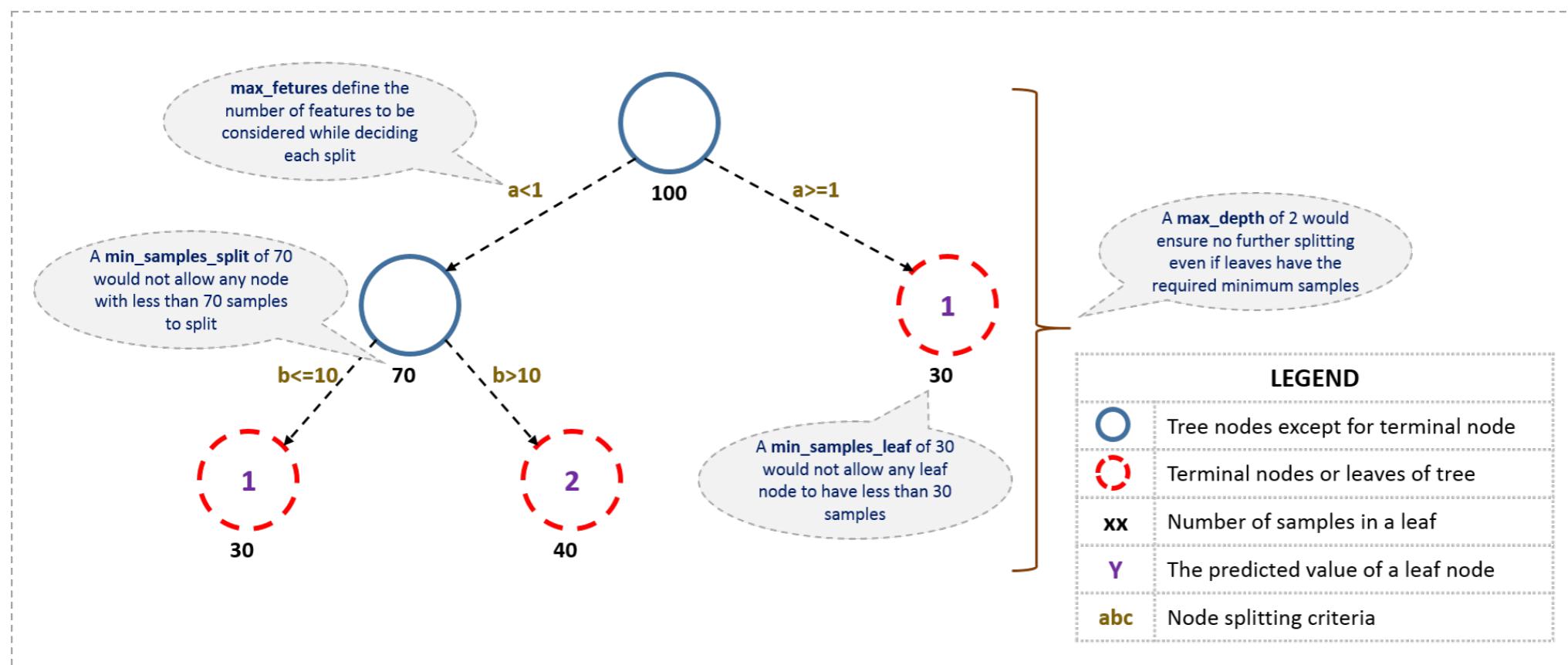
- Boosting classification tree:
  - Initialize the outcome
  - Iterate from 1 to total number of trees
    - Update the weights for targets based on previous run (higher weight for the ones mis-classified and lower weight for the ones classified)
    - Fit the model on selected subsample of data
    - Make predictions on the full set of observations
    - Update the output with current results taking into account the learning rate
  - Return the final output.

Learning rate:

- The effect is that the model can quickly fit, then overfit the training dataset.
- A technique to slow down the learning in the gradient boosting model is to apply a weighting factor for the corrections by new trees when added to the model.
- This weighting is called the shrinkage factor or the learning rate.

# Tuning parameter of GBM

- A tuning parameter is parameter used in statistics algorithm in order to control their behavior.
- The overall parameters of GBM can be divided into 2 categories:
  - Tree-Specific Parameters: These affect each individual tree in the model.
    - max\_features: the number of features to consider while searching for a best split
    - max\_depth: the maximum depth of a tree
    - min\_samples\_leaf: defines the minimum observations required in a terminal node or leaf
    - min\_samples\_split: defines the minimum number of observations which are required in a node to be considered for splitting



# GBM Tuning parameter

- Boosting Parameters: these affect the boosting operation in the model.
  - n\_estimators: the number of sequential trees to be modeled
  - learning\_rate: this determines the impact of each tree on the final outcome. GBM works by starting with an initial estimate which is updated using the output of each tree. The learning parameter controls the magnitude of this change in the estimates.
  - subsample: the fraction of observations to be selected for each tree. Selection is done by random sampling. Values slightly less than 1 make the model robust by reducing the variance. Typical values ~0.8 generally work fine but can be fine-tuned further.

# Tuning parameter

There are two types of parameter to be tuned– tree based and boosting parameters. There are no optimum values for learning rate as low values always work better, given that we train on sufficient number of trees.

Though, GBM is robust enough to not overfit with increasing trees, but a high number for a particular learning rate can lead to overfitting. But as we reduce the learning rate and increase trees, the computation becomes expensive and would take a long time to run on personal computers.

we can take the following approach:

- Choose a relatively high learning rate. Generally the default value of 0.1 works but somewhere between 0.05 to 0.2 should work for different problems
- Determine the optimum number of trees for this learning rate.
- Tune tree-specific parameters for decided learning rate and number of trees.
- Lower the learning rate and increase the estimators proportionally to get more robust models.

# Tuning parameter

The order of tuning variables should be decided carefully.

- Tuning n\_estimators with lower learning rate
- Take the variables with a higher impact on outcome first. For instance, max\_depth and min\_samples\_split have a significant impact and we will tune those first.
- Tuning min\_samples\_leaf
- Tuning max\_features
- Tuning subsample
- making more robust models with higher n\_estimators and lower learning rate

# XGboost-extreme boosting

- Extreme Gradient Boosting (xgboost) is similar to gradient boosting framework but more efficient. It has both linear model solver and tree learning algorithms. So, what makes it fast is its capacity to do parallel computation on a single machine.
- This makes xgboost at least 10 times faster than existing gradient boosting implementations. It supports various objective functions, including regression, classification and ranking.
- Since it is very high in predictive power and relatively fast with implementation, “xgboost” becomes an ideal fit for many competitions. It also has additional features for doing cross validation and finding important variables.
- It has more tuning parameters than GBM.

XGboost documentation:

<https://xgboost.readthedocs.io/en/latest/>

How to install XGboost in terminal or anaconda prompt:

<https://anaconda.org/conda-forge/xgboost>

# Tuning parameters

General Parameters:

generally use default set

- booster [default=gbtree]
  - Select the type of model to run at each iteration. It has 2 options:
    - gbtree: tree-based models
    - gblinear: linear models
- silent [default=0]:
  - Silent mode is activated if set to 1, i.e. no running messages will be printed.
  - It's generally good to keep it 0 as the messages might help in understanding the model.
- nthread [default to maximum number of threads available if not set]
  - This is used for parallel processing and number of cores in the system should be entered.
  - If you wish to run on all cores, value should not be entered and algorithm will detect automatically

# Tuning parameters

Booster Parameters:

- **eta** [default=0.3]
  - Analogous to learning rate in GBM
  - Makes the model more robust by shrinking the weights on each step
  - Typical final values to be used: 0.01-0.2
- **min\_child\_weight** [default=1]
  - Defines the minimum sum of weights of all observations required in a child.
  - This is similar to min\_child\_leaf in GBM but not exactly. This refers to min “sum of weights” of observations while GBM has min “number of observations”.
  - Used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
  - Too high values can lead to under-fitting hence, it should be tuned using CV.
- **max\_depth** [default=6]
  - The maximum depth of a tree, same as GBM.
  - Used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
  - Should be tuned using CV.
  - Typical values: 3-10
- **max\_leaf\_nodes**
  - The maximum number of terminal nodes or leaves in a tree.
  - Can be defined in place of max\_depth. Since binary trees are created, a depth of ‘n’ would produce a maximum of  $2^n$  leaves.
  - If this is defined, GBM will ignore max\_depth.
- **gamma [default=0]**
  - A node is split only when the resulting split gives a positive reduction in the loss function(RSS/information gain). Gamma specifies the minimum loss reduction required to make a split.
  - Makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.

# Tuning parameters

Booster Parameters:

- **max\_delta\_step** [default=0]
  - In maximum delta step we allow each tree's weight estimation to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative.
  - Usually this parameter is not needed, but it might help when class is extremely imbalanced (for example, we have a lot of zeros class).
- **subsample** [default=1]
  - Same as the subsample of GBM. Denotes the fraction of observations to be randomly samples for each tree.
  - Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
  - Typical values: 0.5-1
- **colsample\_bytree** [default=1]
  - Similar to max\_features in GBM. Denotes the fraction of columns to be randomly samples for each tree.
  - Typical values: 0.5-1
- **colsample\_bylevel** [default=1]
  - Denotes the subsample ratio of columns for each split, in each level.
- **lambda** [default=1]
  - L2 regularization term on weights (analogous to Ridge regression)
  - This used to handle the regularization part of XGBoost. Though many data scientists don't use it often, it should be explored to reduce overfitting.
- **alpha** [default=0]
  - L1 regularization term on weight (analogous to Lasso regression)
  - Can be used in case of very high dimensionality so that the algorithm runs faster when implemented
- **scale\_pos\_weight** [default=1]
  - A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.

# Tuning parameters

Learning Task Parameters:

- **objective** [default=reg:linear]
  - This defines the loss function to be minimized. Mostly used values are:
    - **binary:logistic** –for binary classification, returns predicted probability (not class)
    - multi:softmax – multiclass classification using the softmax objective, returns predicted class (not probabilities)
    - you also need to set an additional num\_class (number of classes) parameter defining the number of unique classes
    - multi:softprob –same as softmax, but returns predicted probability of each data point belonging to each class.
- **eval\_metric** [ default according to objective ]
  - The metric to be used for validation data.
  - The default values are rmse for regression and error for classification.
  - Typical values are:
    - **rmse – root mean square error**
    - mae – mean absolute error
    - logloss – negative log-likelihood
    - error – Binary classification error rate (0.5 threshold)
    - merror – Multiclass classification error rate
    - mlogloss – Multiclass logloss
    - **auc: Area under the curve**
- **seed** [default=0]
  - The random number seed.
  - Can be used for generating reproducible results and also for parameter tuning.

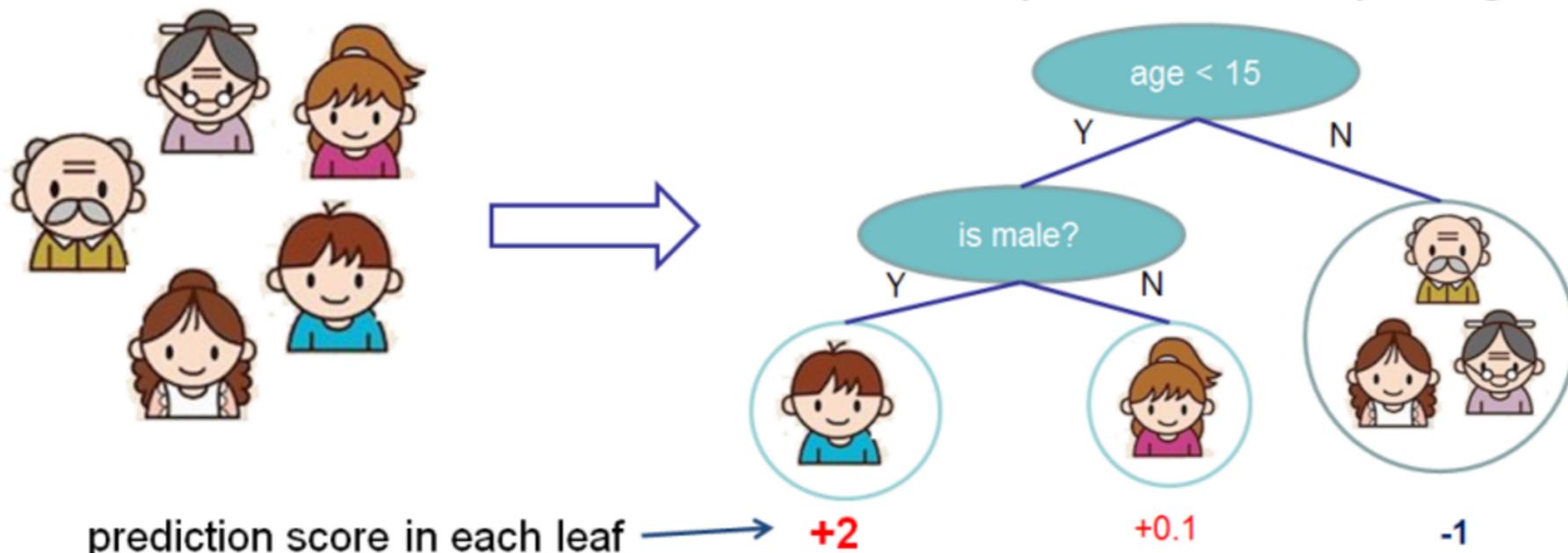
# Tree ensemble

Here's a simple example of a CART that classifies whether someone will like computer games.

We classify the members of a family into different leaves, and assign them the score on the corresponding leaf.

Input: age, gender, occupation, ...

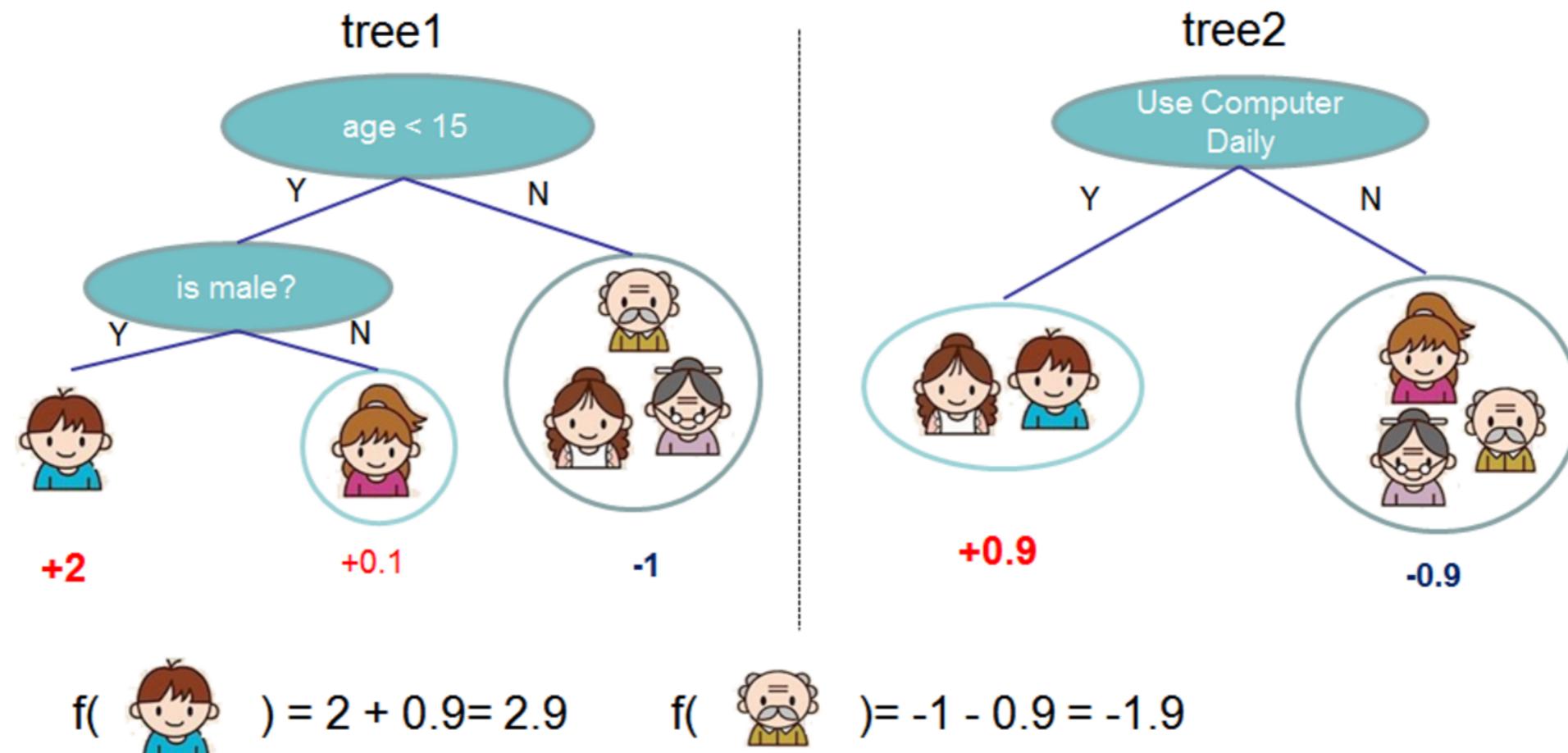
Does the person like computer games



Usually, a single tree is not strong enough to be used in practice. What is actually used is the so-called tree ensemble model, which sums the prediction of multiple trees together.

# Tree ensemble

Here is an example of a tree ensemble of two trees. The prediction scores of each individual tree are summed up to get the final score. If you look at the example, an important fact is that the two trees try to complement each other.



# Tree boosting

$$\hat{y}_i = \sum_{k=1}^K f_k(x_i), f_k \in \mathcal{F}$$

where K is the number of trees, f is a function in the functional space F, and F is the set of all possible CARTs.

Therefore our objective to optimize can be written as

$$\text{obj}(\theta) = \sum_i^n l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k)$$

Use an additive strategy: fix what we have learned, and add one new tree at a time

$$\begin{aligned}\hat{y}_i^{(0)} &= 0 \\ \hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\ \hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\ &\dots \\ \hat{y}_i^{(t)} &= \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)\end{aligned}$$

# Tree boosting

$$\begin{aligned}\text{obj}^{(t)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^t \Omega(f_i) \\ &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + \text{constant}\end{aligned}$$

For regression trees: use MSE for the first item

$$\begin{aligned}\text{obj}^{(t)} &= \sum_{i=1}^n (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \sum_{i=1}^t \Omega(f_i) \\ &= \sum_{i=1}^n [2(\hat{y}_i^{(t-1)} - y_i)f_t(x_i) + f_t(x_i)^2] + \Omega(f_t) + \text{constant}\end{aligned}$$

take the Taylor expansion of the loss function up to the second order

$$\text{obj}^{(t)} = \sum_{i=1}^n [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + \text{constant}$$

where the  $g_i$  and  $h_i$  are defined as

$$\begin{aligned}g_i &= \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \\ h_i &= \partial_{\hat{y}_i^{(t-1)}}^2 l(y_i, \hat{y}_i^{(t-1)})\end{aligned}$$

# Tree boosting

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

w: the vector of scores on leaves, r: tuning parameter,  $\lambda$ : learning rate, T is the number of leaves.

$$\text{obj}^{(t)} = \sum_{j=1}^T [G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2] + \gamma T$$

$$G_j = \sum_{i \in I_j} g_i$$

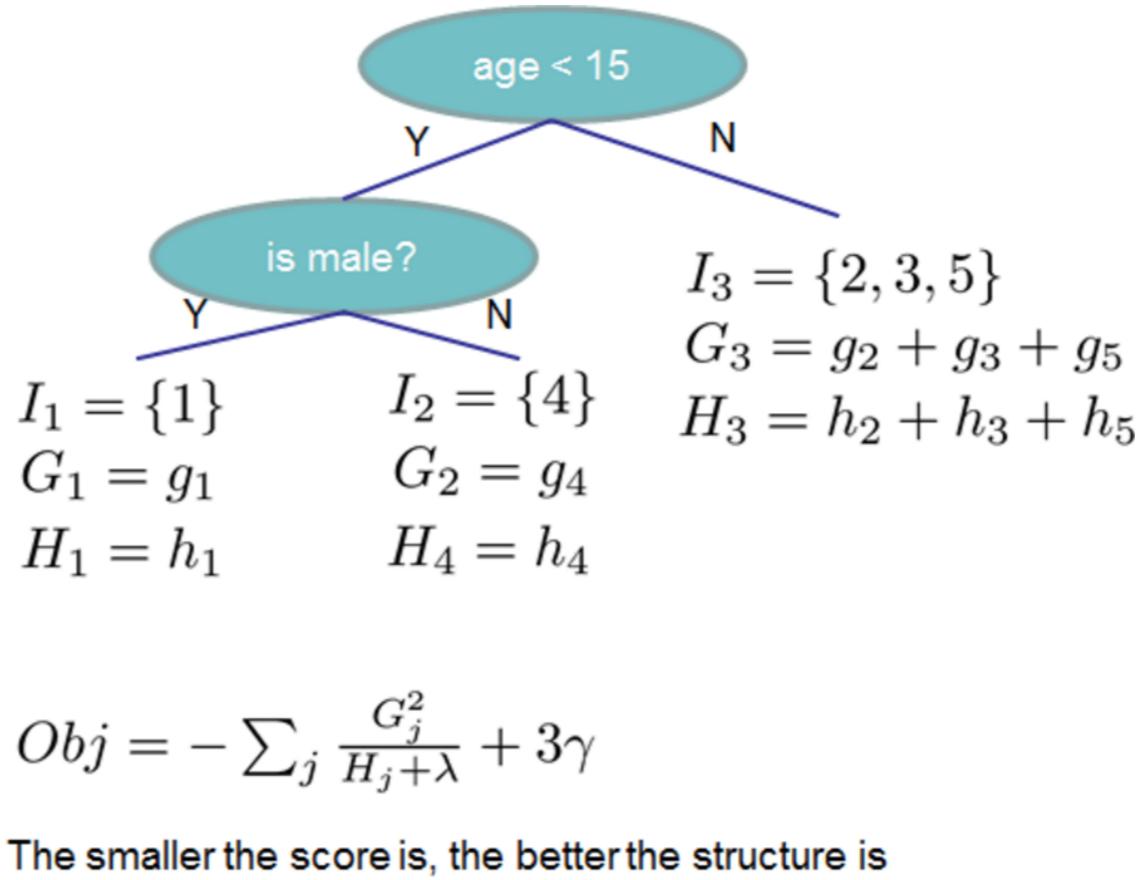
$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

$$H_j = \sum_{i \in I_j} h_i$$

$$\text{obj}^* = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$

# Tree boosting

Instance index	gradient statistics
1	 g <sub>1</sub> , h <sub>1</sub>
2	 g <sub>2</sub> , h <sub>2</sub>
3	 g <sub>3</sub> , h <sub>3</sub>
4	 g <sub>4</sub> , h <sub>4</sub>
5	 g <sub>5</sub> , h <sub>5</sub>



For a given tree structure, we push the statistics  $g_i$  and  $h_i$  to the leaves they belong to, sum the statistics together, and use the formula to calculate how good the tree is. This score is like the impurity measure in a decision tree, except that it also takes the model complexity into account.

# Hyperparameter Optimization

- Hyperparameter optimization or model selection is the problem of choosing a set of hyperparameters for a learning algorithm, usually with the goal of optimizing a measure of the algorithm's performance on an independent data set.
- Cross-validation is used to estimate this generalization performance.
- Optimize a loss function on the training set to ensure the model does not overfit its data by tuning, e.g., regularization.

Grid Search:

- The traditional way of performing hyperparameter optimization has been grid search, or a parameter sweep, which is simply an exhaustive searching through a manually specified value of the hyperparameter of a learning algorithm.
- A grid search algorithm must be guided by some performance metric, typically measured by cross-validation on the training set.
- Since the parameter space of a machine learner may include real-valued or unbounded value spaces for certain parameters, manually set bounds and discretization may be necessary before applying grid search.

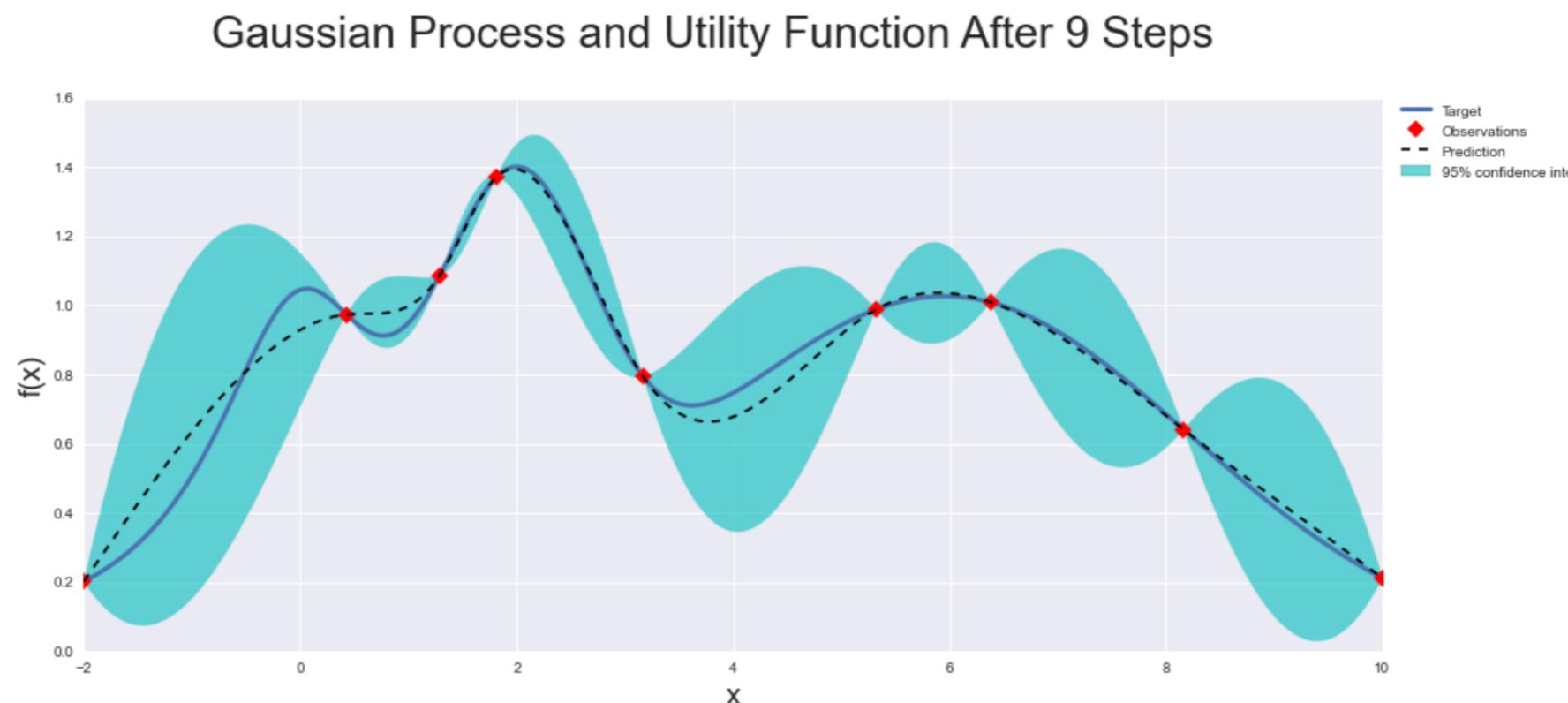
# Hyperparameter Optimization

## Bayesian optimization

- Bayesian optimization is a methodology for the global optimization of noisy black-box functions. Applied to hyperparameter optimization, Bayesian optimization consists of developing a statistical model of the function from hyperparameter values to the objective evaluated on a validation set.
- Intuitively, the methodology assumes that there is some smooth but noisy function that acts as a mapping from hyperparameters to the objective.
- In Bayesian optimization, one aims to gather observations in such a manner as to evaluate the machine learning model the least number of times while revealing as much information as possible about this function and, in particular, the location of the optimum.
- Bayesian optimization relies on assuming a very general prior functions when combined with observed hyperparameter values and corresponding outputs yields a distribution over functions.
- The methodology proceeds by iteratively picking hyperparameters to observe (experiments to run) in a manner that trades off exploration (hyperparameters for which the outcome is most uncertain) and exploitation (hyperparameters which are expected to have a good outcome).

# Hyperparameter Optimization

Bayesian optimization works by constructing a posterior distribution of functions (gaussian process) that best describes the function we want to optimize. As the number of observations grows, the posterior distribution improves, and the algorithm becomes more certain of which regions in parameter space are worth exploring and which are not, as seen in the picture below.



- In practice, Bayesian optimization has been shown to obtain better results in fewer experiments than grid search, due to the ability to reason about the quality of experiments before they are run.
- github link: <https://github.com/fmfn/BayesianOptimization>
- installation: <https://anaconda.org/conda-forge/bayesian-optimization>

# Pros and Cons of Trees Model

Classifier	Description	Pros	Cons
Random Forests	<ul style="list-style-type: none"> <li>Operate by constructing a multitude of decision trees at training time</li> <li>Output the class that is the mode of the classification or regression of the individual trees</li> <li>Correct for decision trees' habit of overfitting to their training set</li> </ul>	<ul style="list-style-type: none"> <li>Prevent from overfitting</li> <li>Good with very large data set</li> <li>No transformation needed</li> <li>Robust against outliers</li> </ul>	<ul style="list-style-type: none"> <li>Low interpretability</li> <li>Less accurate than boosted tree models</li> </ul>
Gradient Boosting	<ul style="list-style-type: none"> <li>Produces a prediction model in the form of an ensemble of decision trees</li> <li>Build the model in a stage-wise fashion</li> <li>Generalize results by allowing optimization of an arbitrary differentiable loss function</li> </ul>	<ul style="list-style-type: none"> <li>Use all feature</li> <li>Better predictability</li> <li>Lower possibility of overfitting with more trees</li> </ul>	<ul style="list-style-type: none"> <li>Susceptible to outliers</li> <li>Lack of interpretability and higher complexity</li> <li>Harder to tune parameters than other models</li> <li>Slow to train or score</li> </ul>
Xgboost	<ul style="list-style-type: none"> <li>An advanced implementation of gradient boosting algorithm</li> <li>Use a more regularized model formalization to control over-fitting, which gives it better performance</li> </ul>	<ul style="list-style-type: none"> <li>Use regularization to reduce overfitting</li> <li>Support parallel processing</li> <li>Make splits up to the <code>max_depth</code> specified and then start pruning the tree backwards and remove splits beyond which there is no positive gain.</li> <li>Built-in Cross Validation</li> </ul>	Same as Gradient Bossing