CS 559: Machine Learning Fundamentals & Applications

Lecture 9: Decision Tree and Ensemble Methods

STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

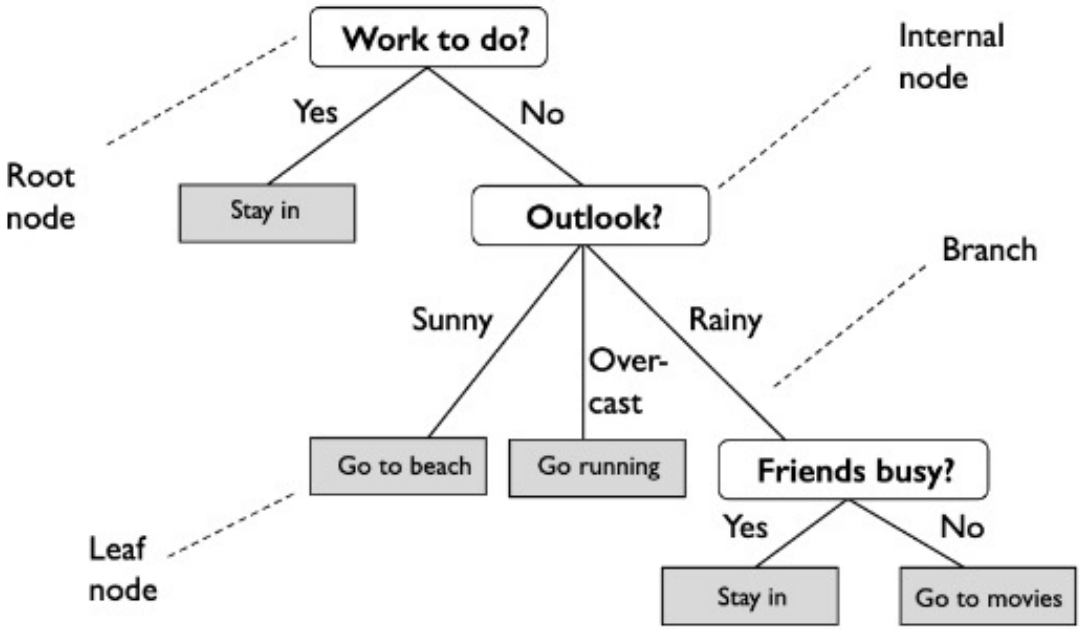
9.1. Decision Trees (DTs)

9.1.1. Introduction

- Decision Tree (DT) algorithms use **iterative, top-down** construction **decision-making** on hypothesis and visualize a hierarchy of decisions forking the dataset into subspaces.
- DT algorithms use any **Boolean function** to search the entire training dataset and split it after decision makings. There is a total of $\mathbf{2^m}$ potential splits to be evaluated given $\boldsymbol{m}$ features.
- Hypothesis space is searched greedily (make a choice that looks the best).
- The challenge is whether an algorithm can learn the right function or a good approximation.

9.1.2. Terminology

- **Root**: the node at the top of the tree – no incoming edge, zero or more outgoing edges.
- **Internal node**: one incoming edge, two or more outgoing edges.
- **Leaf**: the node at the end of the tree – no outgoing edges and assigns the class label. If nodes are not pure, it takes the majority vote.
- **Parent and child nodes**: a node at one lower depth is the parent node, and one at a higher depth is the child node.
- **Branch**: Connects nodes.
- **Depth**: This refers to the top-down height of the tree from the root.
- **Height**: Refers to the bottom-up height of the tree from the leaf.



9.1.2. Ruled-based Learning

- We can think of DT as nested "if-else" rules that are simple conjunction of conditions. For example,

$$Rule\ 1 = (if\ x = 1) \cap (if\ y = 2) \cap \cdots$$

Eq. 9 - 1

- Multiple rules can then be joined into a set of rules and be applied to predict the target value of training data. For example,

$$Class\ 1 = if\ (Rule\ 1 = True) \cup (Rule\ 2 = True) \cup \cdots$$

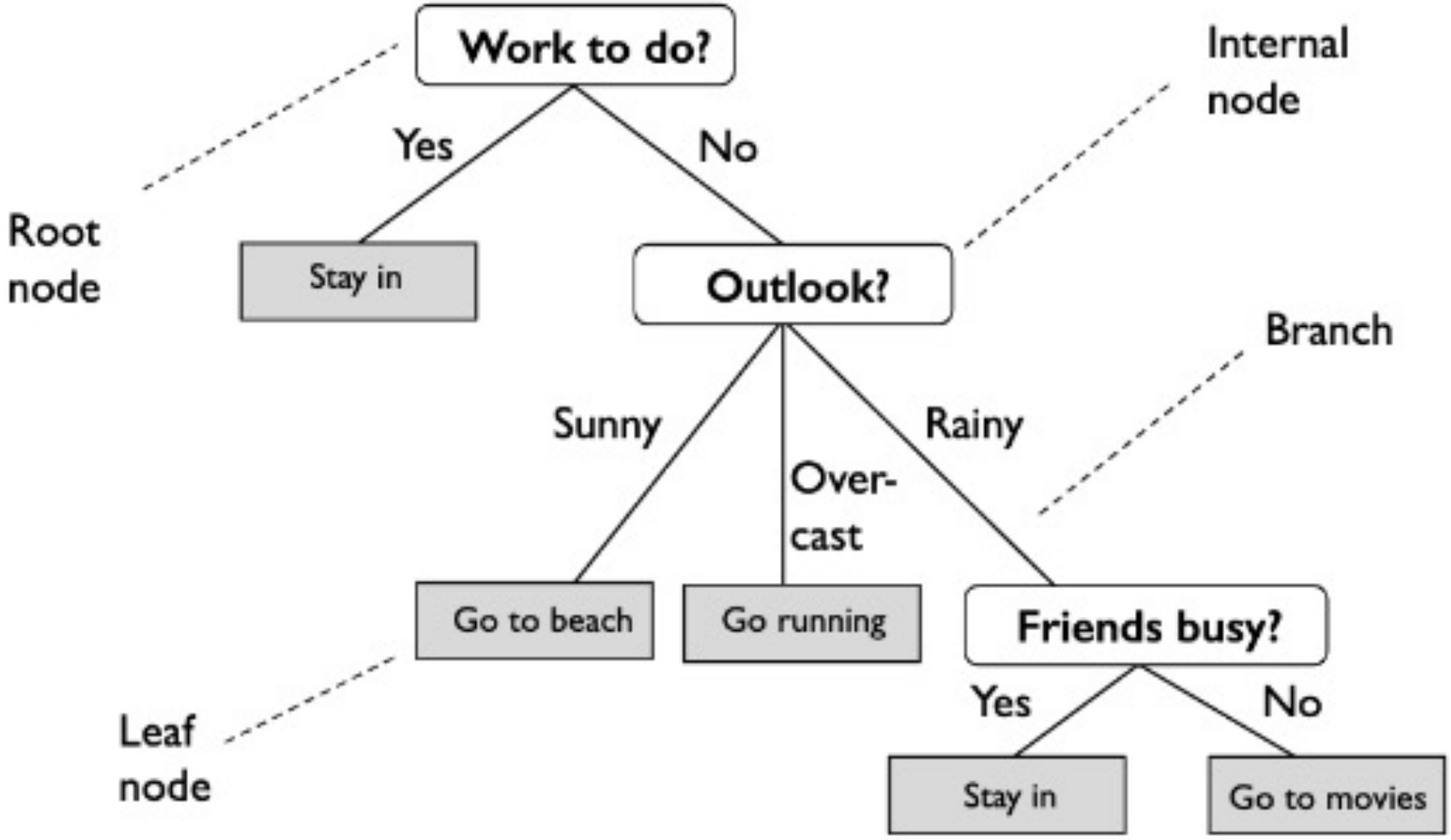
Eq. 9 - 2

- Each leaf node in a DT represents a set of rules. For example,

$$(Work\ to\ do? = No) \cap (Outlook? = Rainy) \cap (Friends\ busy? = No)$$

Eq. 9 - 3

9.1.2. Ruled-based Learning



9.1.3. Things to concern?

Even rules can be constructed from DT easily, sometimes…

- It is not always possible to build a DT from a set of rules which may not be immediately apparent how.
- A rule set evaluation is much more expensive than a tree evaluation.
- We can have multiple rulesets if we are not careful.
- Because of the flexible rules, DT is more **prone to overfitting**. Especially if there are more hypothesis spaces than DTs.

9.1.4. Different DT algorithms - ID3, C4.5, and CART

General Differences:

Most DT algorithms differ in the following ways:

1. Splitting criteria – Information gain (Shannon Entropy, Gini impurity, misclassification error), use of statistical tests, objective function, etc.
2. Binary split vs. multi-way splits
3. Discrete vs. continuous variables
4. Pre- vs. post-pruning

9.1.4. Different DT algorithms - ID3, C4.5, and CART

Iterative Dichotomizer 3

- Described in Quinlan, J. R. (1986). Introduction of decision trees.
- One of the earliest DT algorithms
- Can only handle discrete features
- Multi-category splits
- No pruning, prone to overfitting
- Short and wide trees compared to CART
- Maximizes information gain and minimizes entropy

9.1.4. Different DT algorithms - ID3, C4.5, and CART

C4.5

- Described by Quinlan, J. R. (1993). C4.5: Programming for machine learning. *Morgan Kauffmann, 38, 48*
- Handles both continuous and discrete features. Continuous feature splitting is very expensive.
- The splitting criterion is computed via the gain ratio.
- Handles missing attributes by ignoring them in the computation.
- Performs post-pruning (bottom-up pruning)

9.1.4. Different DT algorithms - ID3, C4.5, and CART

CART – Classification and Regression Trees

- Described in Breiman, L. (1984). *Classification and regression trees*. Belmont, Calif: Wadsworth International Group.
- Handles continuous and discrete features.
- Strictly binary splits (results trees are taller compared to ID3 and C4.5)
  - Generate better trees than C4.5 but tend to be larger and harder to interpret.
  - For k attributes, there are $2^{k-1} - 1$ ways to create a binary partitioning.
- Variance reduction in regression trees.
- Uses Gini impurity in classification trees.
- Performs cost-complexity pruning.

9.1.5. Information Gain (IG)

- The IG is the standard criterion for choosing the splitting in DT.
  - The higher the IG, the better the split.
- IG relies on the concept of mutual information – the reduction of the entropy of one variable by knowing the other.
- We want to maximize mutual information when defining splitting criteria.

$$Gain(D, x_j) = H(D) - \sum_{v \in Values(x_j)} \frac{|D_v|}{|D|} H(D_v)$$

Eq. 9 - 4

where D is the training set at the parent node, and D_v is a dataset at a child node upon splitting.

9.1.6. Information Theory and Entropy

- In ID3, Shannon Entropy is used to measure improvement in a DT; i.e., we use it as an optimization metric (or impurity measure).
- It was originally proposed in the context of encoding digital information in the form of Bits (0s or 1s).
- Consider it as a measure of the amount of information of discrete random variables (two outcomes, Bernoulli distribution)

9.1.6. Information Theory and Entropy

- Shannon information:
  - Shannon defined information as the number of bits to encode a number $1/p$, where p is the probability that an event is a rule ($\frac{1}{1-p}$ is the uncertainty).
  - The number of bits for encoding $\frac{1}{p}$ is $\log_2\left(\frac{1}{p}\right)$.
  - $-\log_2 p \rightarrow [\infty, 0]$; if we are 100% certain about an event, we gain 0 information.
  - E.g., assume two soccer teams both teams have a win probability of 50%.
    - If the information team 1 wins is transmitted 1 bit: $\log_2\left(\frac{1}{0.5}\right) = \log_2(2) = 1$

9.1.6. Information Theory and Entropy

- Shannon entropy is then the "average information"
  - Entropy: $H(p) = \sum_i p_i \log_2\left(\frac{1}{p_i}\right) = -\sum_i p_i \log_2(p_i)$

Eq. 9 - 5

  - E.g., assume team 1 and 2 have win possibilities 75% and 25%, respectively. The average information content is

$$H(p) = (-0.75 \log_2 0.75 - 0.25 \log_2 0.25) = 0.81$$

9.1.7. Growing DT via Entropy or Gini Impurity than Misclassification Error

Consider the general measurement of information gain,

$$G(D, x_j) = I(D) - \sum_{v} \frac{|D_v|}{|D|} I(D_v)$$

Eq. 9 - 6

where I is a function of the impurity measurement.

9.1.7. Growing DT via Entropy or Gini Impurity than Misclassification Error

Let the misclassification error be

$$E(D) = \frac{1}{N}\sum_{n}^{N} L\left(\hat{y}^{[n]}, y^{[n]}\right),$$

Eq. 9 - 7

with the 0-1 loss

$$L(\hat{y}, y) = \begin{cases} 0 & if\ \hat{y} = y \\ 1 & otherwise. \end{cases}$$

Eq. 9 - 8

9.1.8. Gini Impurity

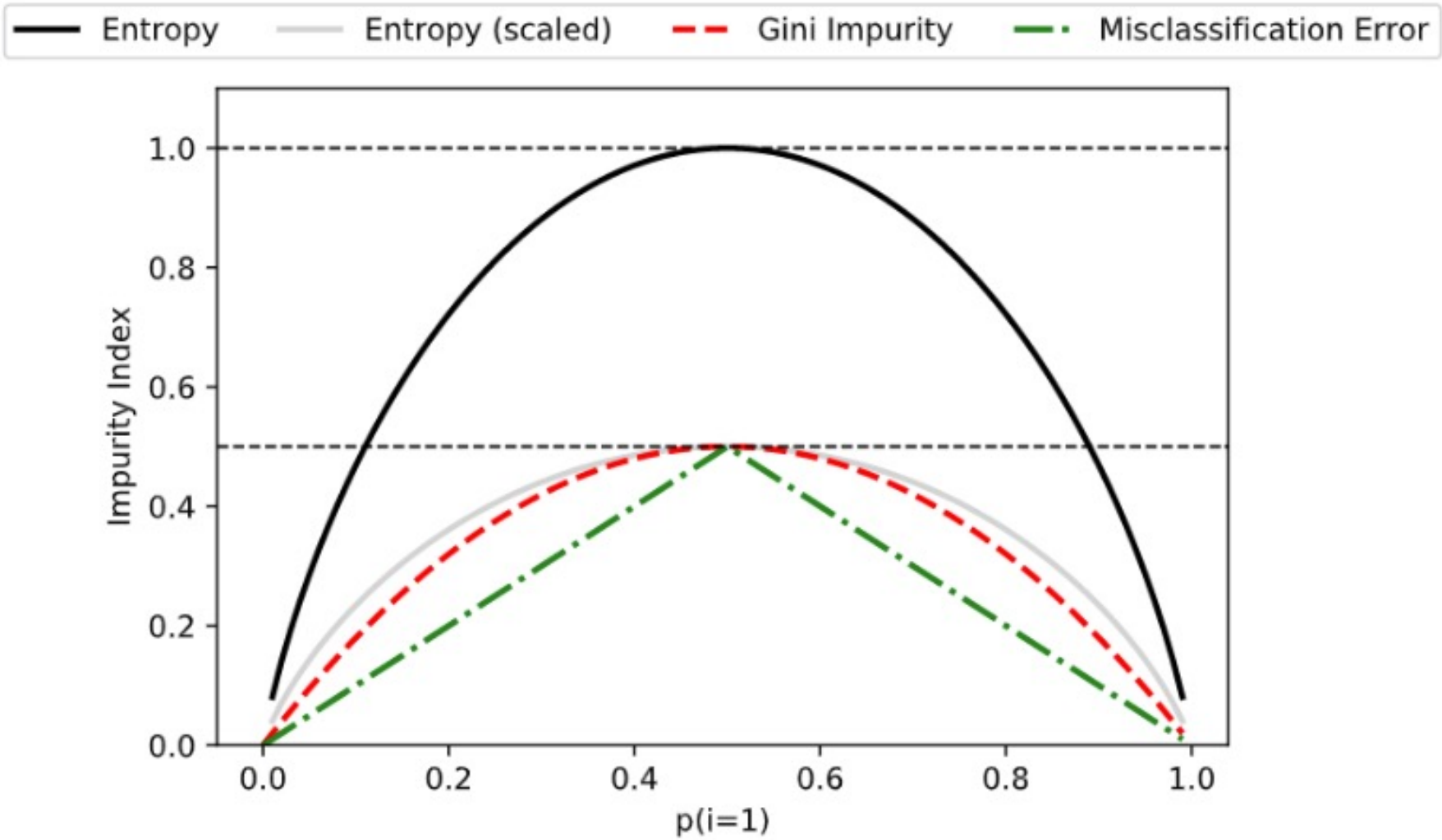
- Gini impurity is measured used in CART as opposed to entropy:

$$Gini(t) = 1 - \sum_{i} (p(c = i)^2)$$

Eq. 9 - 9

- In practice, the use of Gini Impurity or entropy really does not matter. Both will have the same concave which is essential.
- Gini is computationally more efficient to compute than entropy which could make code negligibly more efficient.

9.1.8. Gini Impurity



9.1.8. Gini Impurity

Gain Ratio

- The gain ratio was introduced by Quinlan penalizes splitting categorical attributes with many values via

$$GainRatio(D, x_j) = \frac{Gain(D, x_j)}{SplitInfo(D, x_j)}.$$

Eq. 9 - 10

- SplitInfo measures the entropy of the attribute itself:

$$SplitInfo(D, x_j) = -\sum_{v \in x_j} \frac{|D_v|}{|D|} \log_2 \frac{|D_v|}{|D|}.$$

Eq. 9 - 11

Gini Impurity

Overfitting

- If DTs are not pruned, they have a high risk to overfit the training data to a high degree.
- Overfitting occurs if models pick up noise or errors in the training dataset and it can be seen as a performance gap between training and test data.
- DT pruning is a general approach for minimizing overfitting.
- Pre-Pruning:
  - Set a depth cut-off (maximum tree depth) at the beginning.
  - Cost-complexity pruning: $I + \alpha|N|$, where I is an impurity measure, α is a tuning parameter, and $|N|$ is the total number of nodes.
  - Stop growing if a split is not statistically significant.
  - Set a minimum number of data points for each node.

9.1.8. Gini Impurity

Overfitting

- Post-Pruning:
  - Grow a full tree then remove nodes.
  - Reduces error by removing nodes from validation.
  - Convert trees to rules first and then prune the rules
    - There is one rule per leaf node.
    - If rules are not sorted, rule sets are costly to evaluate but more expressive.
    - In contrast to pruned rule sets, rules from DTs are mutually exclusive.

9.1.9. DT for Regression

- Grow the tree through variance reduction at each node.
- Use a metric for the continuous target variables comparison to the predictions such as the mean squared error at a given node *t*:

$$MSE = \frac{1}{N_t} \sum_{n=1, n \in D_t}^{N} \left(y_t^{[n]} - h\left(\boldsymbol{x}^{[n]}\right)_t \right)^2 .$$

Eq. 9 - 12

- It is often referred to as "within-node variance" and the splitting criterion is called "variance reduction".

9.1.9. Conclusion

Pros:

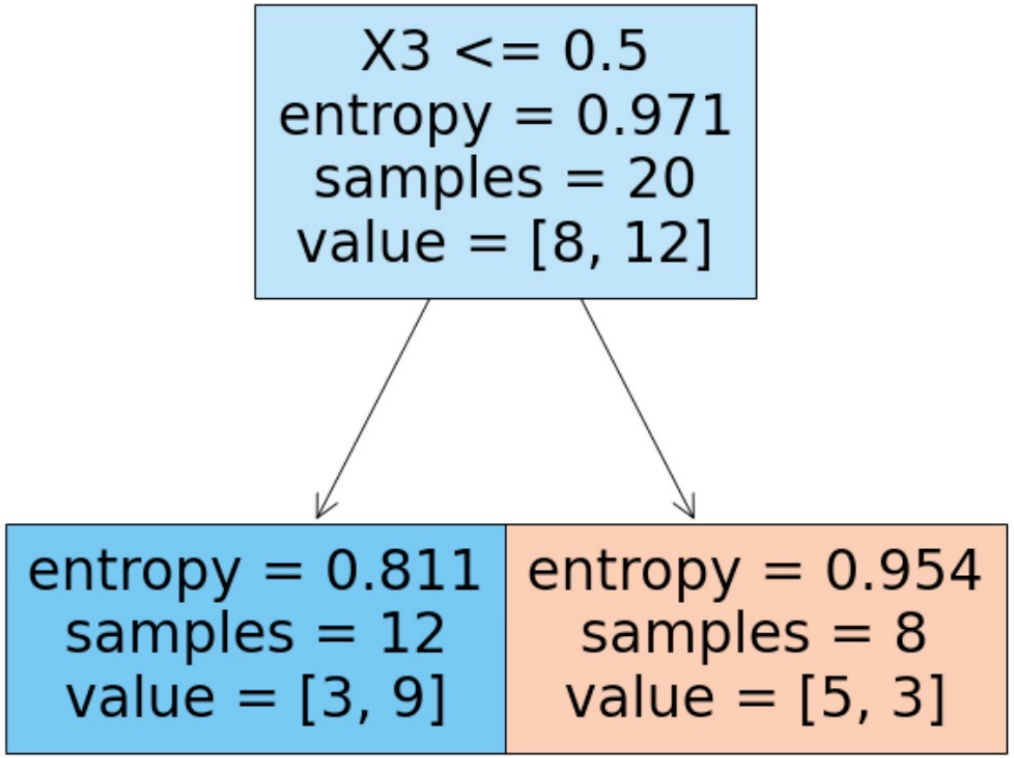
- Easy to interpret and communicate
- Independent of feature scaling

Cons:

- Easy to overfit
- Elaborate pruning required
- Output range is bounded in regression trees.

9.1.10. DT Classifier Example

```
1  x1 = [1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1]
2  x2 = [0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1]
3  x3 = [0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0]
4  y = [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1]
✓ 0.2s
```



9.1.10. DT Classifier Example

N =	20				
N(Y=1)	12	N(Y=0)	8		
p(y=1)	0.6	p(y=0)	0.4		
H(Y)=	0.970950594				
P(x1=1)	0.5	P(x1=0)	0.5		
P(x2=1)	0.55	P(x2=0)	0.45		
P(x3=1)	0.4	P(x3=0)	0.6		
p(X1=1|Y=1)	0.3	p(X2=1|Y=1)	0.4	p(X3=1|Y=1)	0.15
p(X1=1|Y=0)	0.2	p(X2=1|Y=0)	0.15	p(X3=1|Y=0)	0.25
p(X1=0|Y=0)	0.2	p(X2=0|Y=0)	0.25	p(X3=0|Y=0)	0.15
p(X1=0|Y=1)	0.3	p(X2=0|Y=1)	0.2	p(X3=0|Y=1)	0.45
H(Y|X1=1)	0.492737649	H(Y|X2=1)	0.5166238	H(Y|X3=1)	0.3642179
H(Y|X1=0)	0.492737649	H(Y|X2=0)	0.4339735	H(Y|X3=0)	0.5573677
H(Y|X1)	0.985475297	H(Y|X2)	0.9505974	H(Y|X3)	0.9215857
IG(X1)	-0.014524703	IG(X2)	0.0203532	IG(X3)	0.0493649

9.2. Ensemble Methods I

9.2.1. Introduction

- Ensemble method is simply to combine and average individual models to **perform better**.

- It involves voting schemes among **high-variance models** to prevent "outlier" predictions and overfitting or involves boosting "weak learners" to become "strong learners".

9.2.2. Ensemble Classifiers

Basic idea: build different **"experts"** and let them **vote**

Advantages:

- Improve predictive performance
- Different types of classifiers can be directly included
- Easy to implement
- Not too much parameter tuning

Disadvantages:

- The combined classier is not transparent (black box)
- Not a compact representation

9.2.2. Ensemble Classifiers

Predict class label for unseen data by aggregating a set of predictions (classifiers learned from the training data)

- Bagging (Breiman 1994 “Bagging Predictors”)
- Random forests (Breiman 2001 “Random Forests”)
- Boosting (Freund and Schapire 1995, Friedman et al. 1998)

9.2.2. Ensemble Classifiers

Large volumes of data: Sometimes, the amount of data to be analyzed can be too large to be handled by a single classifier.

- Partition the data into smaller subsets;
- Train different classifiers;
- Combine their outputs using a combination rule

Too little data: A reasonable sized set of training data is crucial to learn the underlying data distribution.

- Draw overlapping random subsets of the available data using resampling techniques
- Train different classifiers, creating the ensemble

9.2.2. Ensemble Classifiers

Divide and conquer:

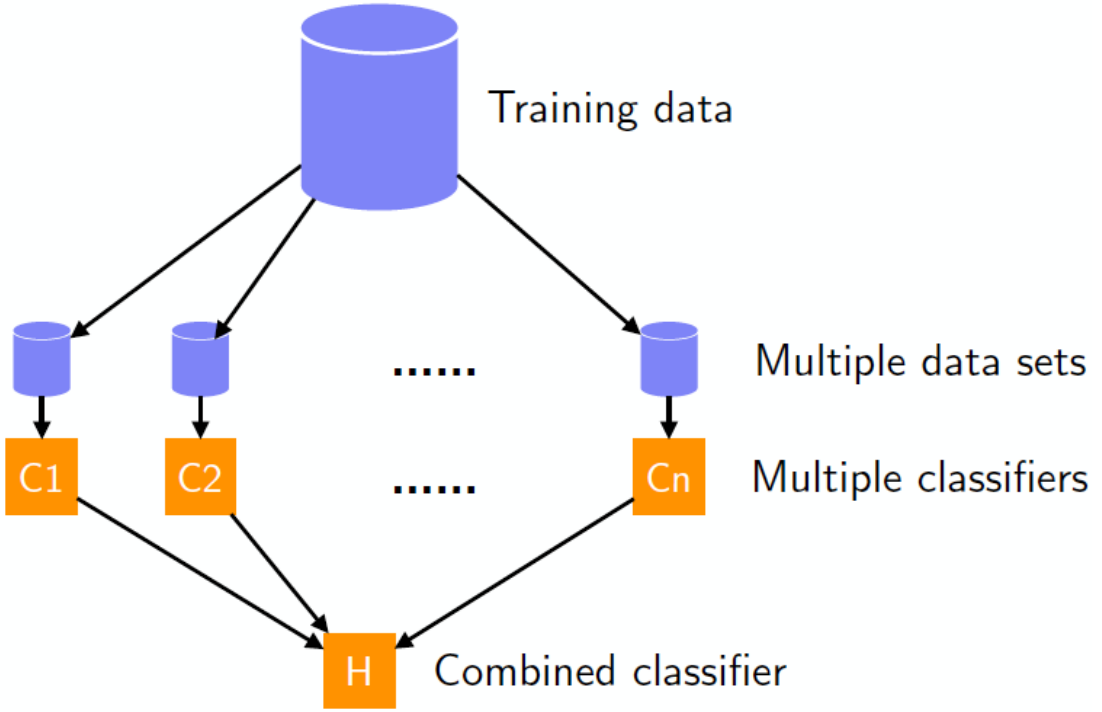
- The given task may be too complex or lie outside the space of functions that can be implemented by the chosen classier method
- Appropriate combinations of simple (e.g., linear) classifiers can learn complex (e.g., non-linear) boundaries

Data fusion:

- Several sets of data were obtained from different sources, where the nature of features is different (e.g.: categorical and numerical features)
- Data from each source can be used to train a different classifier, thus creating ensemble boundaries

9.2.3. Ensemble Classifiers – General Idea

- A method to generate the individual classifiers of the ensemble
- A method for combining the outputs of these classifiers
- The individual classifiers must be diverse (errors on different data)
- If they make the same errors, such mistakes will be carried into the final prediction
- The component classifiers need to be "reasonably accurate" to avoid poor classifiers to obtain the majority of votes.



9.2.4. Bagging: Bootstrap Aggregating

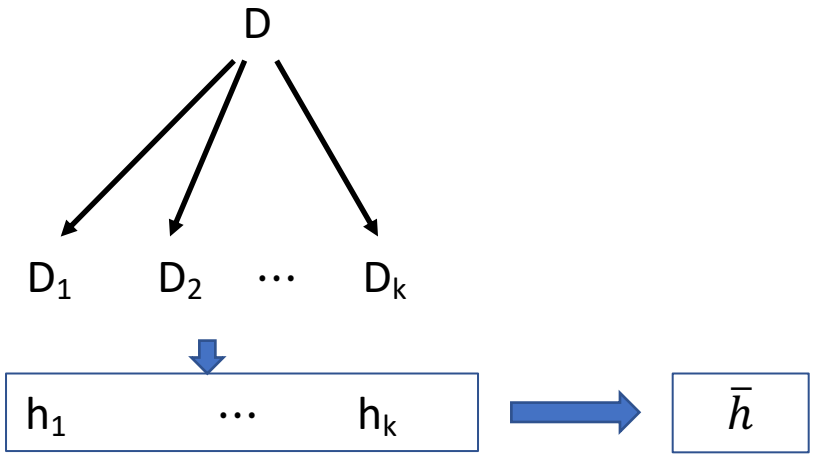
- Take repeated bootstrap samples from training set D (Breiman, 1994)
- Bootstrap sampling: Given set D containing N training examples, create D' by drawing N examples **at random with replacement** from D
- Bagging:
  - Create k bootstrap samples $D_1, \ldots, D_k$
  - Train distinct classifier on each D_i
  - Classify new instances by majority vote/average

$$h(x) = \frac{1}{k}\sum_{j=1}^{k} h_{D_j}(x) \xrightarrow{k\to\infty} \bar{h}(x) \qquad \textit{Eq. 9 - 13}$$

  - Goal: Reduce the variance $E\left[\left(h_D(x) - \bar{h}(x)\right)^2\right]$

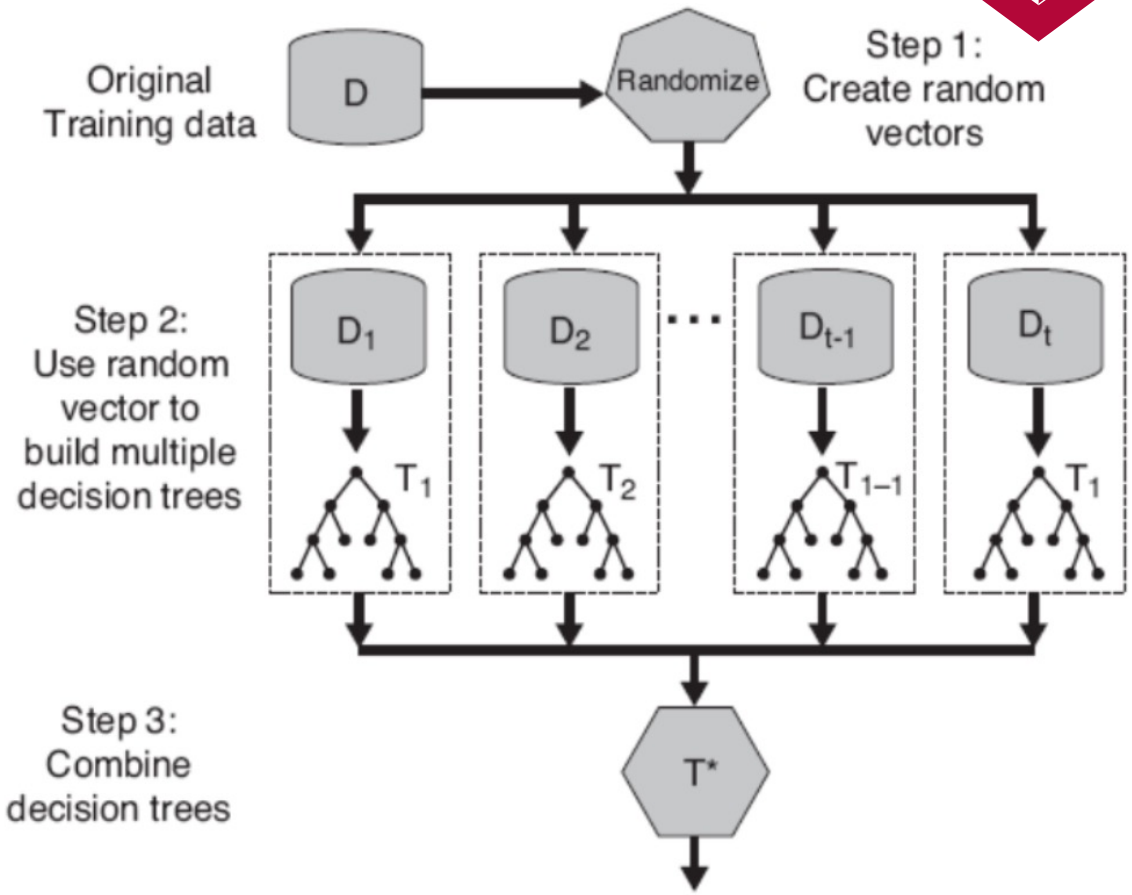
9.2.4. Bagging: Bootstrap Aggregation

- To ensure diverse classifiers, the base classier should be unstable. Small changes in the training set should lead to large changes in the classier output.
- Large error reductions have been observed with decision trees and bagging. This is because decision trees are highly sensitive to small perturbations of the training data.
- Bagging is not effective with nearest-neighbor classifiers. NN classifiers are highly stable with respect to variations in the training data.
- When the errors are highly correlated, bagging becomes ineffective.



9.2.5. Random Forests

- Ensemble method specifically designed for decision tree classifiers.
- Two sources of randomness: “bagging” and “random input vectors”
- Use bootstrap aggregation to train many decision trees.
  - Randomly subsample n examples
  - Train decision tree on subsample
  - Use average or majority vote among learned trees as prediction
- Also randomly subsample features: best split at each node is chosen from a random sample of m attributes instead of all attributes



9.2.5. Random Forests - Algorithm

For b = 1 to B

- Draw a bootstrap sample of size N from the data D with k attributes.
- Grow a random forest tree T_b using the bootstrap sample as follows:
  - Choose m attributes ($m < k$) uniformly at random from the data
  - Choose the best attribute among the m to split on
  - Split on the best attribute and recursively until partitions have fewer than s_{min} number of nodes
- Prediction for a new data point x
  - Regression: $\frac{1}{B}\sum_b T_b(x)$
  - Classification: choose the majority class label among $T_1(x), \ldots, T_B(x)$

Note:

1. Split each training set into two partitions, P and Q, to make the classifier consistent.
2. Do not grow a tree to end. Instead, prune based on the leave-out sample.

9.2.5. Random Forests

Bagging – to average many noisy but approximately unbiased models and reduce the variance.

- Averaging benefits since each tree is **identically distributed** and the expectation of an average of B is the same as the expectation of any one of individual **bootstrapped** trees.
- Identically distributed independent random variables, each with variance σ^2, has the variance $\frac{\sigma^2}{B}$.
- If the variables are dependent with positive pairwise correlation ρ, the variance of the average becomes

$$\sigma^2\left(\rho + \frac{1-\rho}{B}\right) \qquad \textit{Eq. 9 - 14}$$

RF – to improve the variance reduction of bagging by reducing the correlation between the trees without increasing the variance too much during the tree-growing process through random selection of the input variables.

- Typical values of $m = \sqrt{k}$ for classification and the minimum node size is one
- Typical values of $m = k/3$ for regression and the minimum node size is five
- Since $m < k$, it will reduce the correlation between any pair of trees and hence, reduce the variance of the average.
  - As m decreases, $\rho\sigma^2$ decreases but $\frac{1-\rho}{B}\sigma^2$ increases.
  - However, as $B \to \infty$, decrease even though the individual tree variance does not change.

9.2.5. Random Forests

Out of Bag Sample (OOB):

- After fitting on the bootstrap sample, it make predictions on the rest of the data set(out of the bootstrap sample).
- Can not control the numbers of observations in the out of bag sample.
- Almost identical to the obtained K-fold cross validation – RF can be fit in one sequence with the cross-validation being performed along the way.
- The training can be stopped once the OOB error stabilizes.

Variable Importance

The improvement in the split is the importance measure attributed to the splitting variable and is accumulated over all the trees in the forest separately for each variable.

RF uses OOB sample to construct a different variable-importance measure to measure the prediction strength of each variable.

- Consider a b^{th} tree, Θ_b:
  - The prediction accuracy is recorded.
  - The values for j^{th} variable are randomly permuted in the samples and the accuracy gets recomputed.
  - The changes of accuracy as a result of permuting is averaged over all trees indicates the importance of j^{th} variable.
- Similar to setting a coefficient to zero in a linear model.

9.2.6. The Boosting Approach

Bagging reduces variance by averaging but has little effect on bias.
Can we average and reduce bias? (Michael Kerns in 1988)

- Yes, Boosting! (Robert Schapire in 1990)

- devise computer program for deriving rough rules (weak classifier)
- apply procedure to subset of examples and obtain a simple rule
- apply to 2nd subset of examples and obtain a 2nd rule
- repeat T times

How to choose examples on each round?

- concentrate on "hardest" examples (those most often misclassified by previous rule)

How to combine the rules into single prediction rule?

- take (weighted) majority vote of rules

boosting = general method of converting rough rules into highly accurate prediction rule
technically

- assume given "weak" learning algorithm that can consistently find classifiers at least slightly better than random, say, accuracy 55%
- given sufficient data, a boosting algorithm can provably construct single classier with very high accuracy say, 99%

9.2.6. Boosting - gradient descent in function space

Let $\mathcal{H}$ be hypothesis class and H be the ensemble classifier,

$$l(H) = \frac{1}{n}\sum_{i=1}^{n} l(H(x_i), y_i)$$

where $H(x) = \sum_{t=1}^{T} \alpha h_t(x)$ and $h_{t+1} = argmin_{h\in\mathcal{H}} l(H_t + \alpha h_t)$.

Once h_{t+1} is found, add to the ensemble $H_{t+1} = H_t + \alpha h_{t+1}$.

Taylor Approximation. Constant and can be ignored when we minimize it.

Inner product

$$l(H + \alpha h) \approx l(H) + \alpha < \nabla l(H), h >$$

$$\underset{h\in\mathcal{H}}{\text{argmin}}\, l(H + \alpha h) \approx \underset{h\in\mathcal{H}}{\text{argmin}} < \nabla l(H), h >$$

$$= \underset{h\in\mathcal{H}}{\text{argmin}} \sum_{i=1} \frac{\partial l}{\partial H(x_i)} h(x_i)$$

Eq. 9 - 15

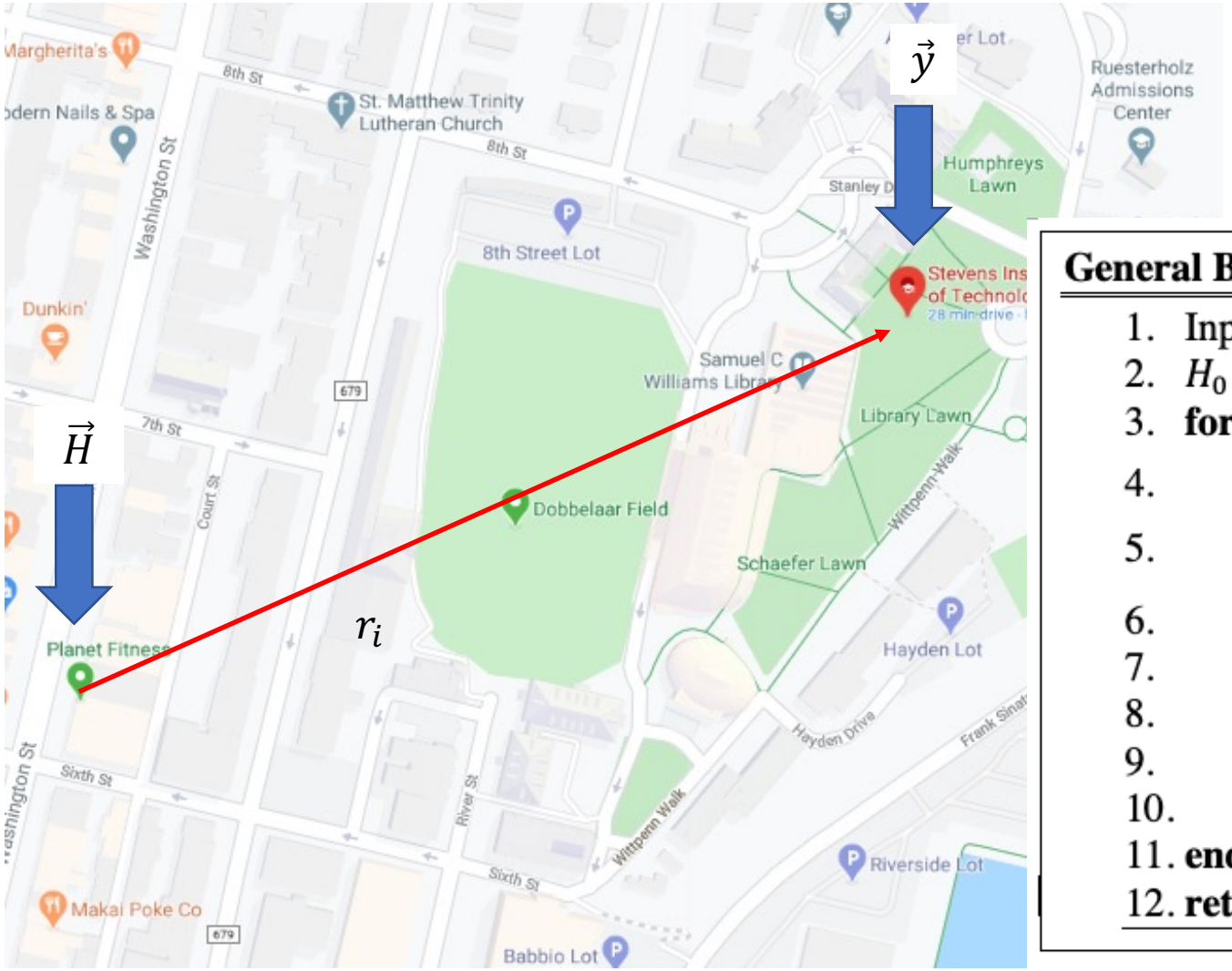
We can do the boosting if we have an algorithm that solves as long as

$$h_{t+1} = \underset{h\in\mathcal{H}}{\text{argmin}} \sum_{i=1} \frac{\partial l}{\partial H(x_i)} h(x_i) < 0$$

$$\| \quad r_i$$

Eq. 9 - 16

9.2.7.Boosting – general pseudo-code



General Boosting Algorithm

1. Input: $l, \alpha, \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}, \mathbb{A}$
2. $H_0 = 0$
3. **for** $t = 0$ to $T - 1$ **do**
4. $\quad \forall i$: $r_i = \frac{\partial l(\{H_t(\boldsymbol{x}_1), \boldsymbol{y}_1\}, \ldots, \{H_t(\boldsymbol{x}_n), \boldsymbol{y}_n\})}{\partial H(\boldsymbol{x}_i)}$
5. $\quad h_{t+1} = \mathbb{A}(\{x_1, r_1), \ldots, (x_n, r_n)\}) = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{n} r_i h(x_i)$
6. $\quad$ **if** $\sum_{i=1}^{n} r_i h_{t+1}(x_i) < 0$ **do**
7. $\quad\quad H_{t+1} = H_t + \alpha_{t+1} h_{t+1}$
8. $\quad$ **else**
9. $\quad\quad$ **return** H_t
10. $\quad$ **end**
11. **end**
12. **return** H_T

9.2.8. Boosting – Gradient Boost

- Classification & Regression
- Weak learners, $h \in \mathcal{H}$, are regressors $h(\boldsymbol{x}) \in \mathcal{R}, \forall \boldsymbol{x}$, typically fixed-depth (between 4-6) regression trees.
- Step size α is fixed to a small constant.
- Loss function: Any differentiable convex that decomposes over the sample

$$\mathcal{L}(H) = \sum_{i=1}^{n} l\big(H(\boldsymbol{x_i})\big)$$

- Must to find a tree $h(\cdot)$ that maximizes

$$h = argmin_{h \in \mathcal{H}} \sum_{i=1}^{n} \underbrace{\frac{\partial l}{\partial H(\boldsymbol{x_i})}}_{\| \; r_i} h(\boldsymbol{x_i})$$

9.2.8. Boosting – Gradient Boost

Assumptions:

1. $\sum_{i=1}^{n} h^2(x_i) = Constant$ – simple to normalize the predictions and important since we can always decrease $\sum_{i=1}^{n} h(x_i) r_i$ by rescaling h with a large constant.
2. CART trees are closed.
3. Define the negative gradient as $t_i = -r_i$.

3: we can square this as this does not have nothing to do with *h()*

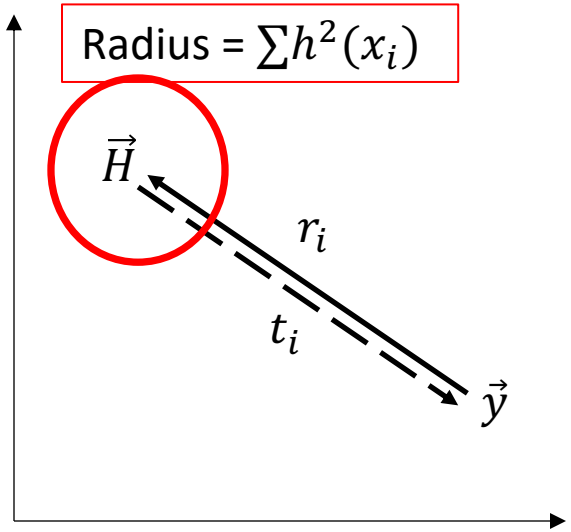
$$argmin_{h \in \mathcal{H}} \sum_{i=1}^{n} r_i h(x_i) = -\operatorname{argmin}_{h \in \mathcal{H}} \sum_{i=1}^{n} t_i^2 - 2 t_i h(\boldsymbol{x_i}) + \left(h(\boldsymbol{x_i})\right)^2$$

1: Enforcing interested only in direction

$$= -\operatorname{argmin}_{h \in \mathcal{H}} \sum_{i=1}^{n} (h(\boldsymbol{x_i}) - t_i)^2$$

2: minimizing is half is good as twice.

We can use the good old regression trees.



9.2.8. Boosting – Gradient Boost

If the loss is a square loss, $l(H) = \frac{1}{2}\sum_{i=1}^{n}(H(x_i) - y_i)^2$,

$$t_i = -\frac{\partial l}{\partial H(x_i)} = y_i - H(\boldsymbol{x_i})$$

It is a residual.

We can use any differentiable and convex loss function and the solution for the next weak learner will always be the regression tree minimizing the square loss.

Gradient Boosting Algorithm

1. Input: $l, \alpha, \{(\boldsymbol{x}_i, y_i)\}, \mathbb{A}$
2. $H = 0$
3. **for** $t = 1$ to T **do**
4. $\quad \forall i: t_i = (y_i - H(\boldsymbol{x}_i))$
5. $\quad h = \underset{h \in \mathcal{H}}{\operatorname{argmin}}(h(\boldsymbol{x}_i) - t_i)^2$
6. $\quad H \leftarrow H + \alpha h$
7. **end**
8. **return** H

9.3.Boosting - AdaBoost

- Classification ($y_i \in \{+1, -1\}$)
- Weak learners, $h \in \mathcal{H}$ are binary, $h(x_i) \in \{-1, +1\}, \forall x$
- We perform line-search to obtain best step-size α
- Exponential loss $l(H) = \sum_{i=1}^{n} e^{-y_i H(x_i)}$

- The gradient is $r_i = -y_i e^{-y_i H(x_i)}$
- Notations:
  - Let $w_i = \frac{e^{-y_i H(x_i)}}{Z}$ where z is the normalizing factor $Z = \sum_{i=1}^{n} e^{-y_i H(x_i)}$.
  - This makes $\sum_{i=1}^{n} w_i = 1$ and w_i is the weight.
- The next weak learner can be solved by optimization.

Exponential loss function

$$h(x_i) = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{n} -y_i e^{-y_i H(x_i)} h(x_i) = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{n} -y_i w_i h(x_i) = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \sum_{h(x_i) \neq y_i} w_i + \sum_{h(x_i) = y_i} w_i$$

Substitute in w_i

$y_i h(x_i) \in \{-1, +1\}$

$$= \underset{h \in \mathcal{H}}{\operatorname{argmin}} \sum_{h(x_i) \neq y_i} w_i$$

The weighted classification error, $\epsilon < 0.5$

$$1 - \sum_{h(x_i) \neq y_i} w_i$$

9.2.9. Boosting - AdaBoost

- We can find the optimal step-size in the closed form every time we take a "gradient" step.
- With given l, H, h

$$\alpha = \underset{\alpha}{\operatorname{argmin}} l(H + \alpha h) = \underset{\alpha}{\operatorname{argmin}} \sum_{i=1}^{n} e^{-y_i[H(x_i)+\alpha h(x_i)]}$$

- Differentiate respect to α and set to 0:

$$0 = \sum_{i=1}^{n} y_i h(x_i) e^{-y_i[H(x_i)+\alpha h(x_i)]} = -\sum_{h(x_i)y_i=1} e^{-y_i[H(x_i)+\alpha h(x_i)]} + \sum_{h(x_i)y_i=-1} e^{-y_i[H(x_i)+\alpha h(x_i)]}$$

$y_i h(x_i) \in \{-1, +1\}$

Substitute in w_i

Substitute in w_i

$$= -\sum_{h(x_i)y_i=1} w_i e^{-\alpha} + \sum_{h(x_i)y_i=-1} w_i e^{\alpha} \Rightarrow -(1-\epsilon)e^{-\alpha} + \epsilon e^{\alpha} = 0$$

Weighted error

$$\Rightarrow \alpha = \frac{1}{2} \ln \frac{1-\epsilon}{\epsilon}$$

9.2.9. Boosting - AdaBoost

- After a taking a step, we need to re-compute all the weights and then re-normalize.
- Let the unnormalized weight be $\widehat{w}_i$

$$\widehat{w}_i \leftarrow \widehat{w}_i e^{-\alpha h(x_i) y_i}$$

- The normalizer Z becomes

$$Z \leftarrow Z\left(2\sqrt{\epsilon(1-\epsilon)}\right)$$

- Then

$$w_i \leftarrow \frac{w_i e^{-\alpha h(x_i) y_i}}{2\sqrt{\epsilon(1-\epsilon)}}$$

AdaBoost Algorithm

1. Input: $l, \alpha, \{(\boldsymbol{x}_i, y_i)\}, \mathbb{A}$
2. $H_0 = 0$
3. $\forall i: \sum_i w_i = \frac{1}{n}$
4. **for** $t = 0$ to $T - 1$ **do**
5. calculate h and ϵ
6. **if** $\epsilon < 0.5$ **then**
7. calculate α
8. update $H: H_{t+1} = H_t + \alpha h$
9. renormalize w_i
10. **else**
11. **return** H_t
12. **return** H_T

9.2.10. Boosting - XGBoost

- **Regularization:** XGBoost has an option to penalize complex models through both L1 and L2 regularization.
- **Handling sparse data:** Missing values or data processing steps like one-hot encoding make data sparse.
- **Weighted quantile sketch:** Most existing tree based algorithms can find the split points when the data points are of equal weights (using quantile sketch algorithm). However, they are not equipped to handle weighted data. XGBoost has a distributed weighted **quantile sketch algorithm** to effectively handle weighted data.
- **Block structure for parallel learning:** For faster computing, XGBoost can make use of multiple cores on the CPU. This is possible because of a block structure in its system design. Data is sorted and stored in in-memory units called blocks. Unlike other algorithms, this enables the data layout to be reused by subsequent iterations, instead of computing it again. This feature also serves useful for steps like split finding and column sub-sampling
- **Cache awareness:** Non-continuous memory access is required to get the gradient statistics by row index. Hence, XGBoost has been designed to make optimal use of hardware. This is done by allocating internal buffers in each thread, where the gradient statistics can be stored
- **Out-of-core computing:** This feature optimizes the available disk space and maximizes its usage when handling huge datasets that do not fit into memory

9.2.10. Boosting - XGBoost

Regression – Unique Regression Tree

- Starts from a single leaf with initial predicted value p^o $(= 0.5)$
- Calculate the similarity score $\left(S = \frac{(\sum_{i=1}^{n}(y_i - p_i))^2}{n+\lambda}\right)$
- Split by the threshold
  - Needs to quantify how much better the leaves cluster similar residuals than the root by Gain (G)
  - G= sum of similarity score of left and right leaves – similarity score of root
  - Calculate the output value (O) on each leaf: $O = \frac{\sum_{i=1}^{n}(y_i - p_i)}{n+\lambda}$

x	y	y-p
10	-11	-10.5
20	7	6.5
25	8	7.5
35	-8	-7.5

$$S_1 = \frac{(-10.5+6.5+7.5-7.5)^2}{4+\lambda}$$

We can split by the threshold based on quantile: 15, 22.5, and 30
In each split, find Gain (if we split with $x < 15$)

- Similarity Score on left: $S_{2,L} = \frac{(-10.5)^2}{1+\lambda}$
- Similarity Score on right: $S_{2,R} = \frac{(6.5+7.5-7.5)^2}{3+\lambda}$
- $G = S_{2,L} + S_{2,R} - S_1$
- The preproperate split is at the quantile that gives the highest gain!

9.2.10. Boosting – XGBoost Regression

How to prune the tree?

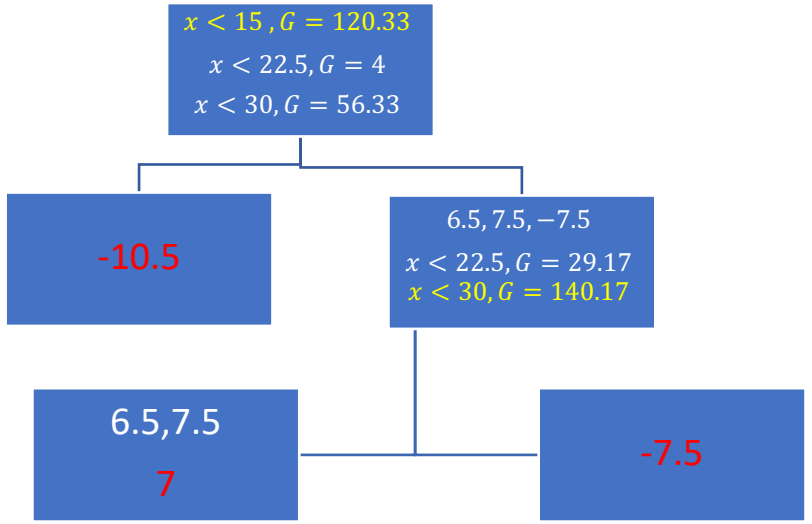
- Based on $G - \gamma$
- If $G - \gamma < 0$, remove the branch
- If $G - \gamma > 0$, do not remove the branch
- When $\lambda > 0$, it is easier to prune leaves because S & G gets smaller.

How to make a new prediction?

- $p^1 = p^o + \eta(O)$
- $\eta = 0.3$ by default
- We build trees until we reach the minimum value of residuals.

- For the simplicity, let $\lambda = 0$.
- We find that the 1st split has the highest gain when $x < 15$
- The 2nd split (2nd branch) is when $x < 30$

x	y	$y - p^o$	$y - p^1$
10	-11	-10.5	-2.65
20	7	6.5	2.6
25	8	7.5	2.6
35	-8	-7.5	-1.75



9.2.10. Boosting – XGBoost Classification

Similarity Score: $S_{i-1} = \left(\frac{\sum(y_i - p_i)^2}{\sum(P_{i-1}(1-P_{i-1}))+\lambda}\right)$ where P is the probability.

Cover C calculates the minimum number of residuals in each leave: $C = \sum(P_{i-1}(1 - P_{i-1}))$

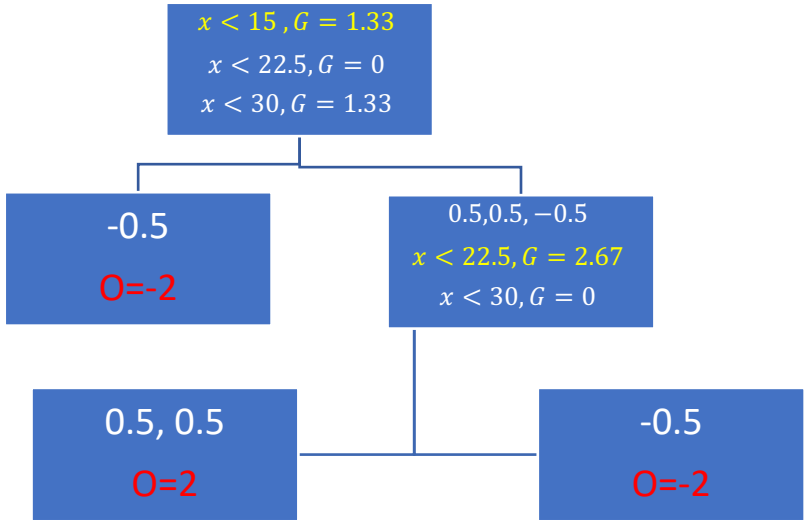
- In regression, $C = n$.
- $C_{min} = 1$ by default.
- In classification, we can set $C = 0$ and that is **min_child_weight**.

Output Value: $O_i = \left(\frac{\sum(y_i - p_i)}{\sum(P_{i-1}(1-P_{i-1}))+\lambda}\right)$

Prediction: $\log(P_i) = \log\left(\frac{P^o}{1-P^o}\right) + \eta O_i \rightarrow P = \frac{e^{\log(P_i)}}{1+e^{\log(P_i)}}$

$S_0 = 0$

x	y	y-p	P_0	P_1
10	0	-0.5	0	0.35
20	1	0.5	0	0.65
25	1	0.5	0	0.65
35	0	-0.5	0	0.35



$$S_{0,1} = \frac{(-0.5)^2}{0.5(1-0.5)} = 1, S_{0,1} = \frac{(-0.5+0.5+0.5)^2}{3\times0.5(1-0.5)} = 0.33$$

$$G_0 = 1 + 0.33 - 0 = 1.33$$

$$S_{1,1} = \frac{(-0.5)^2}{0.5(0.5)} = 1, S_{1,1} = \frac{(0.5+0.5)^2}{2\times0.5(0.5)} = 2$$

$$G_1 = 1 + 2 - 0.33 = 2.67$$

$$O_{-0.5} = -\frac{0.5}{0.5\times0.5} = -2, O_{0.5,0.5} = \frac{0.5+0.5}{2\times0.5(0.5)} = 2$$

9.2.10. Boosting – XGBoost Optimization

- Regression Loss Function: $\frac{1}{2}(y_i - p_i)$
- Classification Loss Function: $-[y_i \log(p_i) - (1 - y_i)\log(1 - p_i)]$
- Generally, $\sum_{i=1}^{n} L(y_i, p_i) + \gamma T + \frac{1}{2}\lambda O_i^2 = \sum_{i=1}^{n} L(y_i, p^o + O_i) + \frac{1}{2}\lambda O_i^2$
- where the loss function uses a second order Tayler Approximation for the optimal Output Value (both)

$$L(y, p_i + O) \approx L(y, p_i) + \frac{d}{dp_i} L(y, p_i) O + \frac{1}{2}\left[\frac{d^2}{dp_i^2} L(y, p_i)\right] O^2$$

Regression:

$$\left[\sum_{i=1}^{n} L(y_i, p_i^o + O)\right] + \frac{1}{2}\lambda O^2 = L(y_1, p_1^o) + g_1 O + \frac{1}{2} h_1 O^2 + \cdots + L(y_n, p_n^o) + g_n O + \frac{1}{2} h_n O^2 + \frac{1}{2}\lambda O^2$$

To find the optimal O value, we make a derivative r.p.t. O and set equals to 0.

$$\sum_{i=1}^{n} g_i + \sum_{i=1}^{n} (h_i + \lambda) O = 0$$

$$\Rightarrow O = -\sum_{i=1}^{n} \frac{g_i}{h_i + \lambda} = \sum_{i=1}^{n} \frac{(y_i - p_i)}{n + \lambda}$$

9.2.10. Boosting – XGBoost Optimization

XG Boost uses the **simplified equation** to calculate the similarity score.
From the previous slide,

$$\left[\sum_{i=1}^{n} L(y_i, p_i^o + O)\right] + \frac{1}{2}\lambda O^2 = L(y_1, p_1^o) + \sum_{i=1}^{n} g_i O + \frac{1}{2}\sum_{i=1}^{n}(h_i + \lambda)O^2$$

- XG Boost starts by **multiplying everything by -1**
- Substitute optimal O from the previous slide,

$$-\sum_{i=1}^{n} g_i \frac{-g_i}{h_i + \lambda} - \frac{1}{2}\sum_{i=1}^{n}(h_i + \lambda) \times \left(\frac{-g_i}{h_i + \lambda}\right)^2 = \frac{1}{2}\sum_{i=1}^{n}\frac{g_i^2}{h_i + \lambda}$$

using $g_i = y_i - p_i$ and $\sum_{i=1}^{n} h_i = n$, we find that

$$\frac{1}{2}\sum_{i=1}^{n}\frac{g_i^2}{h_i + \lambda} = \frac{1}{2}S$$

9.2.11. Summary

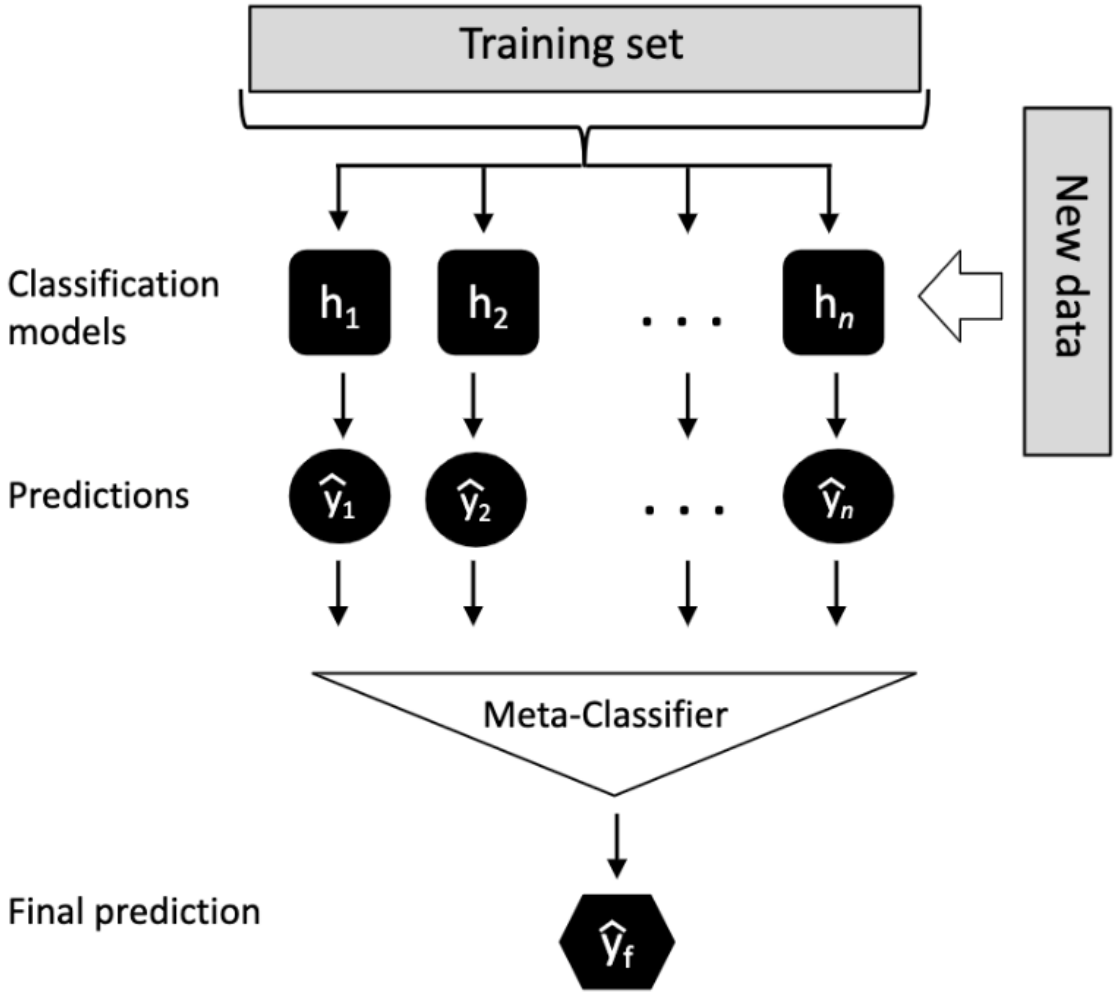
- Decision Trees: need to reduce variance. How?
- Bagging: Bootstrap (random subsampling with replacement)
- Random Forest
  - Bagging method with full decision tree method
  - Easy, feature selection, less data pre-processing
  - But… How to reduce bias?
- Boosting
  - Gradient Boost
    - Good for classification & regression
    - Simple when we use the square loss function
    - Constant small step-size
    - Works with any convex differentiable loss function
  - AdaBoost
    - Only for classification
    - Invented first but turned to be one of gradient boost (exponential loss function)
    - Need to compute weight and step-size for every iteration
  - XG Boost
    - Extreme Gradient Boost

9.3. Ensemble Methods II – Stacking

9.3.1. Stacking - Overview

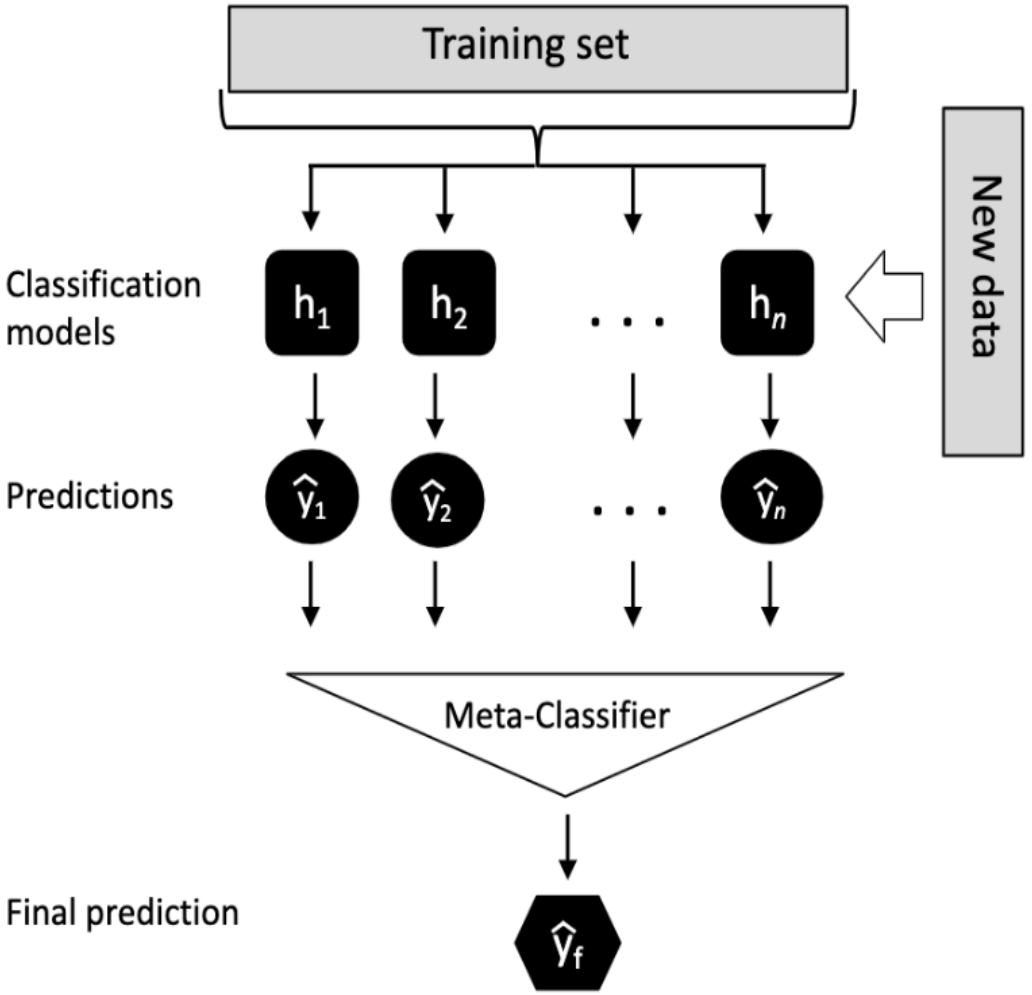
- Stacking is a special case of ensemble method where we combine an ensemble of models through a so-called **meta-classifier**.
- We have "**base learner**" that learn from the initial training set and resulting models then make predictions that serve as input features to a "meta-learner".

9.3.2. Naïve Stacking



- It has a high tendency to suffer from extensive overfitting.
  - The meta-learning strictly relies on base-learning.
- We can use *k*-fold or leave-one-out cross-validation to avoid the overfitting.

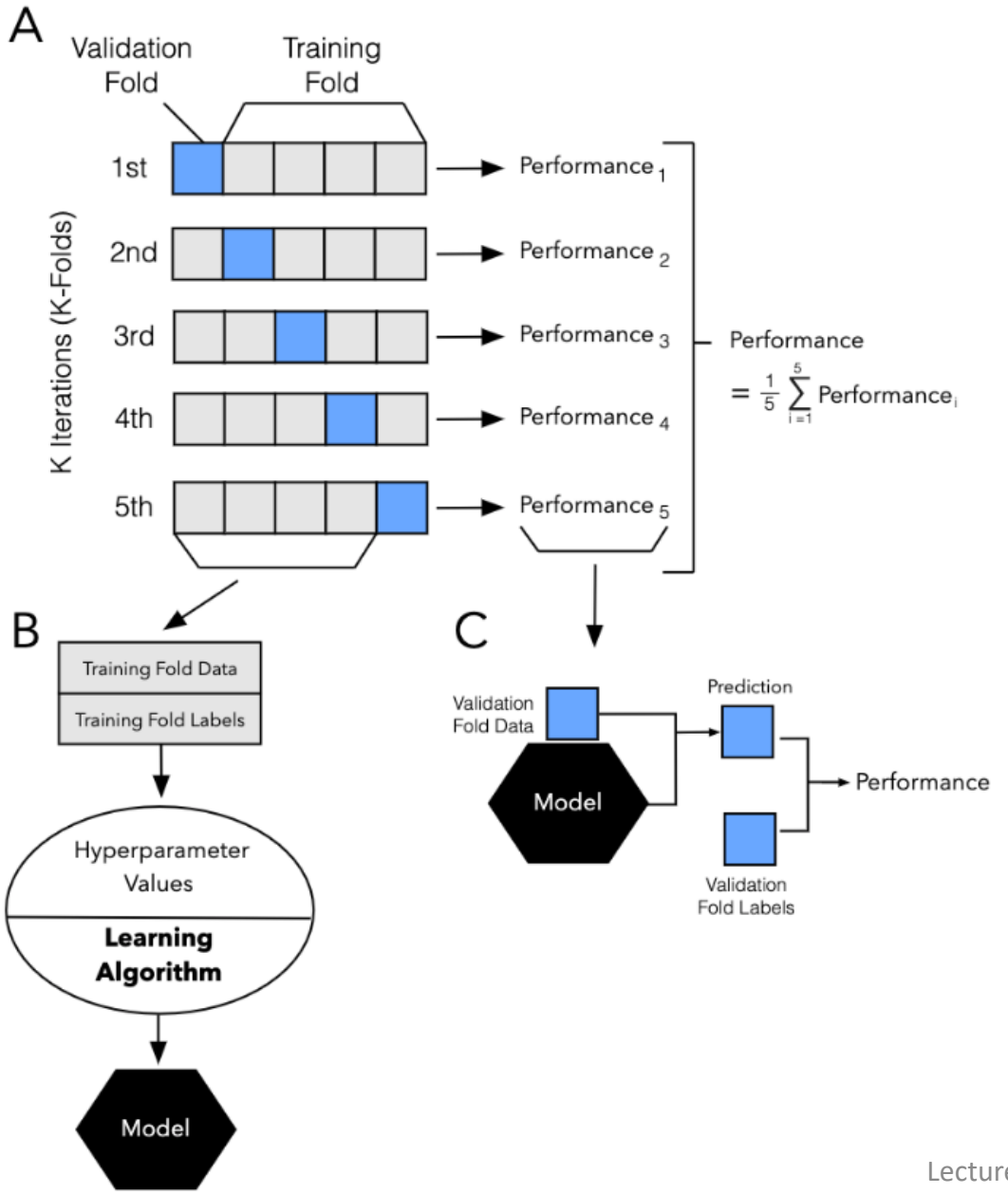
9.3.2. Naïve Stacking

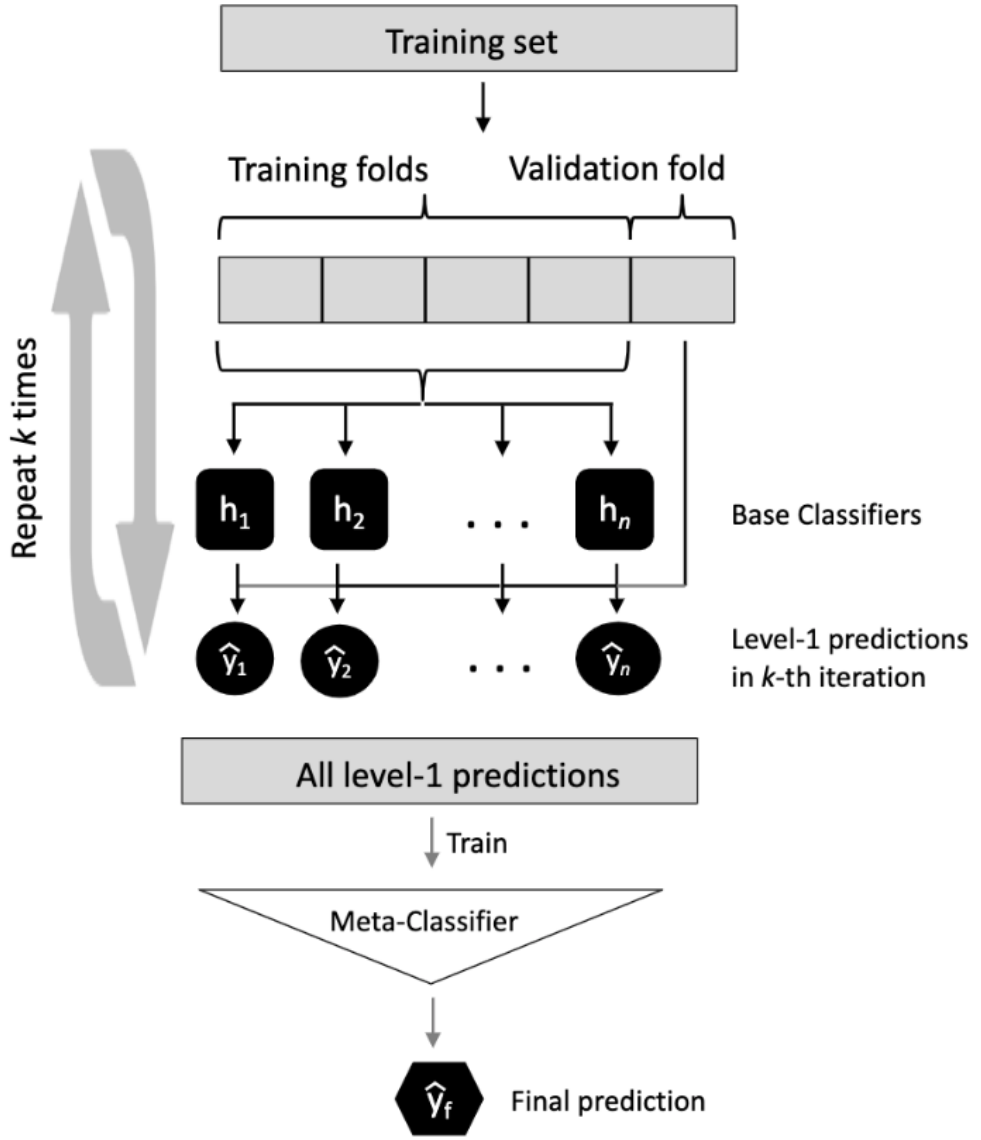


Naïve Stacking Algorithm:

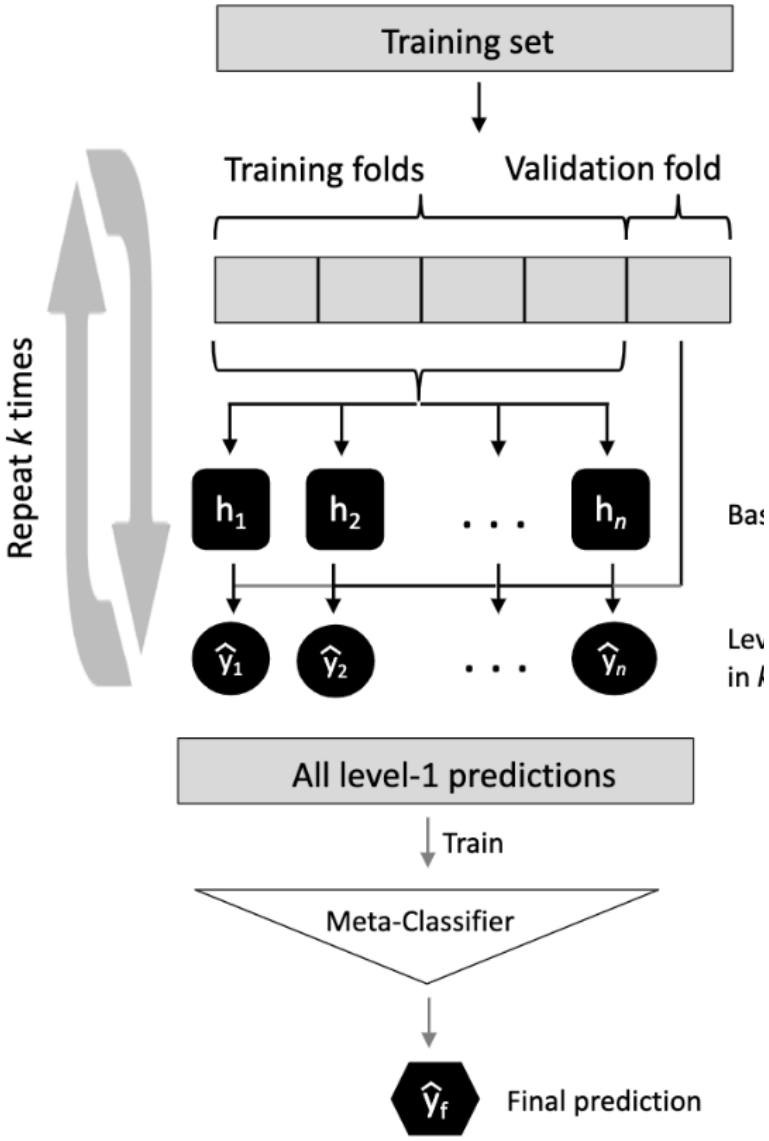
1. Input: Training set $D = \left\{\left\{\boldsymbol{x}^{[1]}, \boldsymbol{y}^{[1]}\right\}, \ldots, \left\{\boldsymbol{x}^{[n]}, \boldsymbol{y}^{[n]}\right\}\right\}$
2. Output: Ensemble classifier h_E
3. //Step 1: Base-learning
4. **for** $t \leftarrow 1$ to T **do**
5. fit base model h_t on D
6. //Step 2: Construct new dataset D' from D
7. **for** $i \leftarrow 1$ to n **do**
8. add $\left\{\boldsymbol{x}'^{[i]}, \boldsymbol{y}^{[i]}\right\}$ to new dataset, where $\boldsymbol{x}'^{[i]} = \left\{h_1\left(\boldsymbol{x}^{[i]}\right), \ldots, h_T\left(\boldsymbol{x}^{[i]}\right)\right\}$
9. //Step 3: Meta-learning
10. **return** $h_E(D')$

9.3.3. Stacking with Cross-Validation





9.3.3. Stacking with Cross-Validation



Stacking with Cross-Validation Algorithm:

1. Input: Training set $D = \left\{\left\{\boldsymbol{x}^{[1]}, \boldsymbol{y}^{[1]}\right\}, \ldots, \left\{\boldsymbol{x}^{[n]}, \boldsymbol{y}^{[n]}\right\}\right\}$
2. Output: Ensemble classifier h_E
3. //Step 1: Base-learning
4. Construct new dataset $D' = \{\}$
5. Randomly split D into k equal-size subsets: $D = \{D_1, \ldots, D_k\}$
6. **for** $j \leftarrow 1$ to k **do**
7. **for** $t \leftarrow 1$ to T **do**
8. fit base model h_t on $D \backslash D_k$
9. **for** $i \leftarrow 1$ to $n \in D \backslash D_k$ **do**
10. add $\left\{\boldsymbol{x}'^{[i]}, \boldsymbol{y}^{[i]}\right\}$ to new dataset D', where $\boldsymbol{x}'^{[i]} = \left\{h_1\left(\boldsymbol{x}^{[i]}\right), \ldots, h_T\left(\boldsymbol{x}^{[i]}\right)\right\}$
11. //Step 3: Meta-learning
12. **return** $h_E(D')$