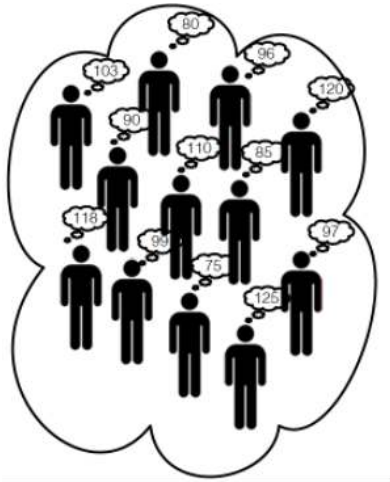
ARTIFICIAL INTELLIGENCE 2

Ensemble Methods

Francesco Spinnato

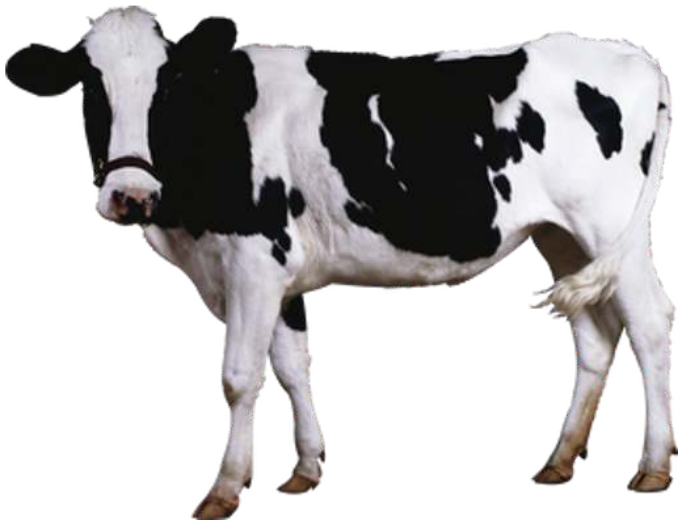
* francesco.spinnato@unipi.it
University of Pisa

The Wisdom of the Crowds



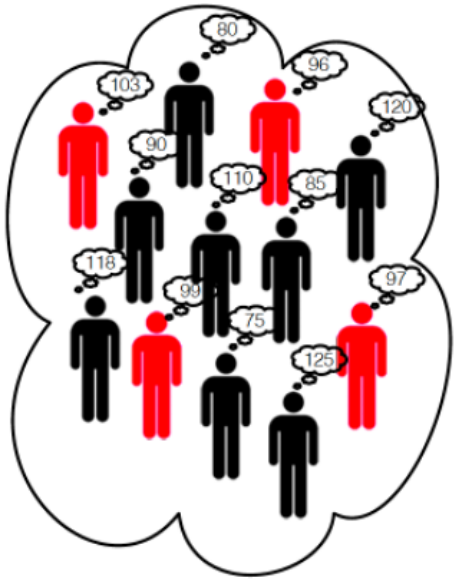
The aggregate judgment of a **diverse** and **independent** group of people can outperform the judgment of a single individual.

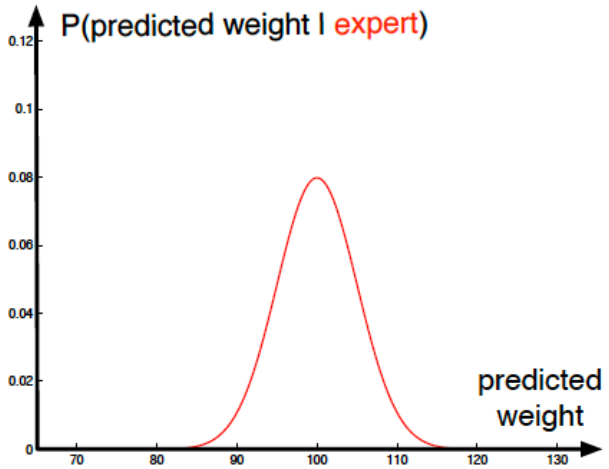
- How much does the cow weigh?
- The crowd's prediction is the average of all predictions
- This crowd predicts 99.8333 kg
- The cow weighs 99.657 kg.



The Wisdom of the Individuals

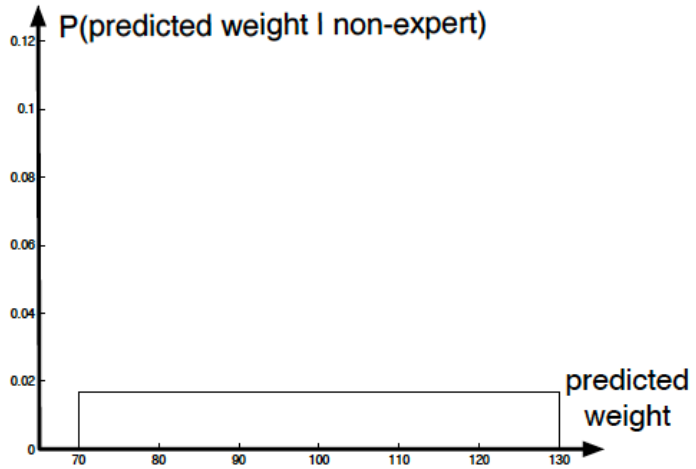
Suppose the crowd is composed of 30% experts and 70% non-experts, with the expertise distribution shown below...





$P(\text{pred. weight} \mid \text{expert}) :$

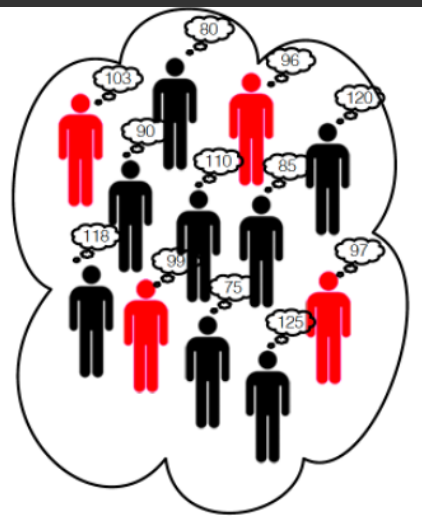
$$\mathcal{N}(100, 5^2)$$



$P(\text{pred. weight} \mid \text{non-expert}) :$

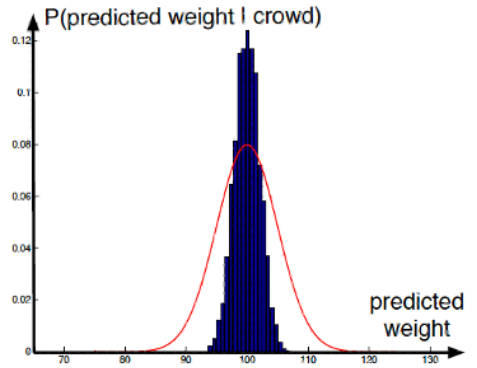
$$\mathcal{U}(70, 130)$$

Single Expert vs. the Crowd



If the crowd contains 50 independent people, its average estimate can outperform that of a typical individual expert.

Thus, under these assumptions, the crowd can be "wiser" than each individual expert.



$\Uparrow$

$P(\text{pred. weight}|\text{crowd})$ **and**
$P(\text{pred. weight}|\text{expert})$

The Wisdom of the Crowds

Why not just ask a bunch of experts?

- a large enough crowd has a high probability that a sufficient number of experts will be in the crowd.
- random selection avoids bias in the choice of experts.
- for some questions it may be hard to identify a diverse set of experts.

J. Surowiecki proposes four elements required to form a wise crowd:

- **Diversity of opinion**. People in the crowd should have a range of experiences, education and opinions (this encourages independent predictions).
- **Independence**. The prediction by a person in the crowd is not influenced by other people in the crowd.
- **Decentralization**. People have specializations and local knowledge.
- **Aggregation**. There is a mechanism for aggregating all predictions into one.

Ensembles

Ensembles are "crowds" (collections) of models that exploit the idea behind the wisdom of the crowd for specific tasks, by combining the predictions of multiple models.

Using ensemble methods means combining models into an ensemble such that **the ensemble performs better than an individual model on average**.

The two main approaches are:

- **Bagging**: voting schemes among high-variance models to prevent “outlier” predictions and overfitting
- **Boosting**: supercharging “weak learners” to become “strong learners.”

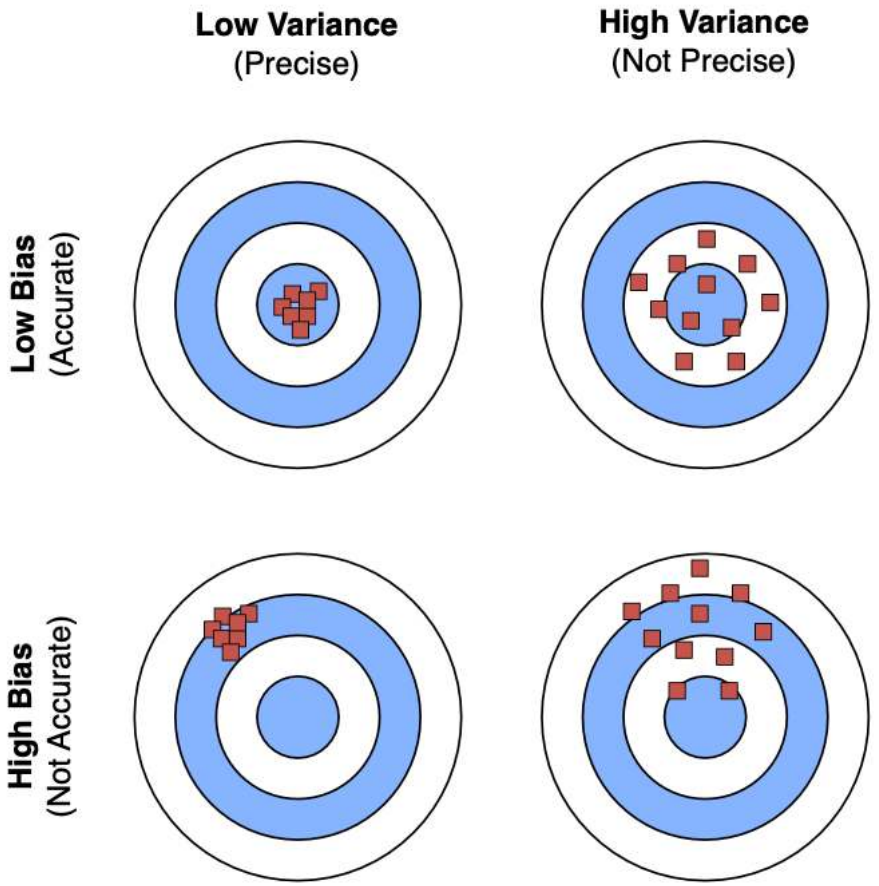
Bias-Variance Trade-off

Every model makes errors. Those errors come from two distinct sources: **bias** and **variance**.

Imagine retraining your model many times, each time on a different sample drawn from the same distribution. For each trained model you record its prediction on the same test point.

A **high-bias** model is systematically wrong because it's too simple to capture the true pattern (**underfitting**)

A **high-variance** model is overly sensitive to the specific training data it saw (**overfitting**)



Notes: https://sebastianraschka.com/pdf/lecture-notes/stat451fs20/07-ensembles__notes.pdf

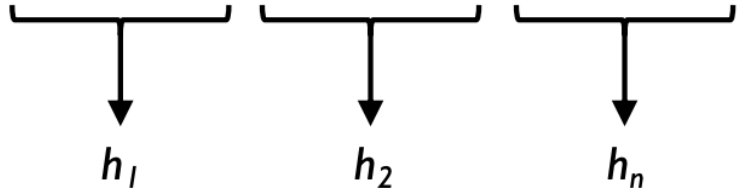
Bagging

Bootstrap

A **bootstrap sample** is a sample of size n drawn **with replacement** from an original training set of size n.
Consequently:

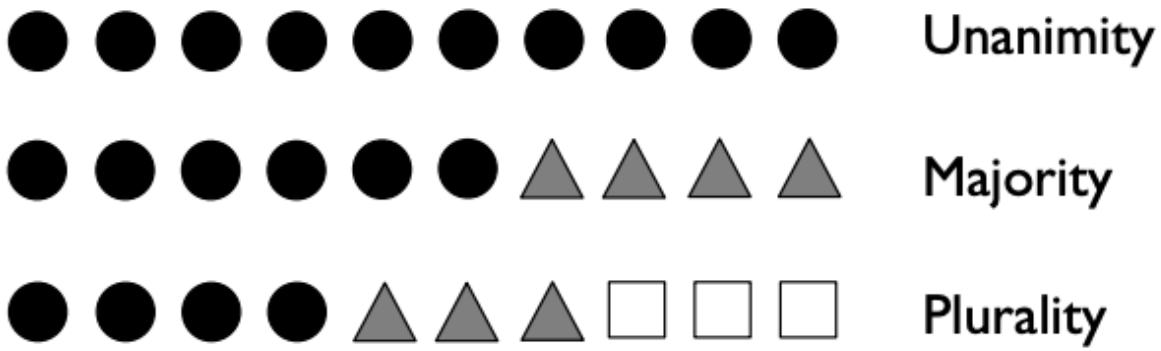
- some training examples are duplicated in each bootstrap sample,
- some training examples do not appear in a given bootstrap sample at all

Training example indices	Bagging round 1	Bagging round 2	…
1	2	7	…
2	2	3	…
3	1	2	…
4	3	1	…
5	7	1	…
6	2	7	…
7	4	7	…



Aggregating

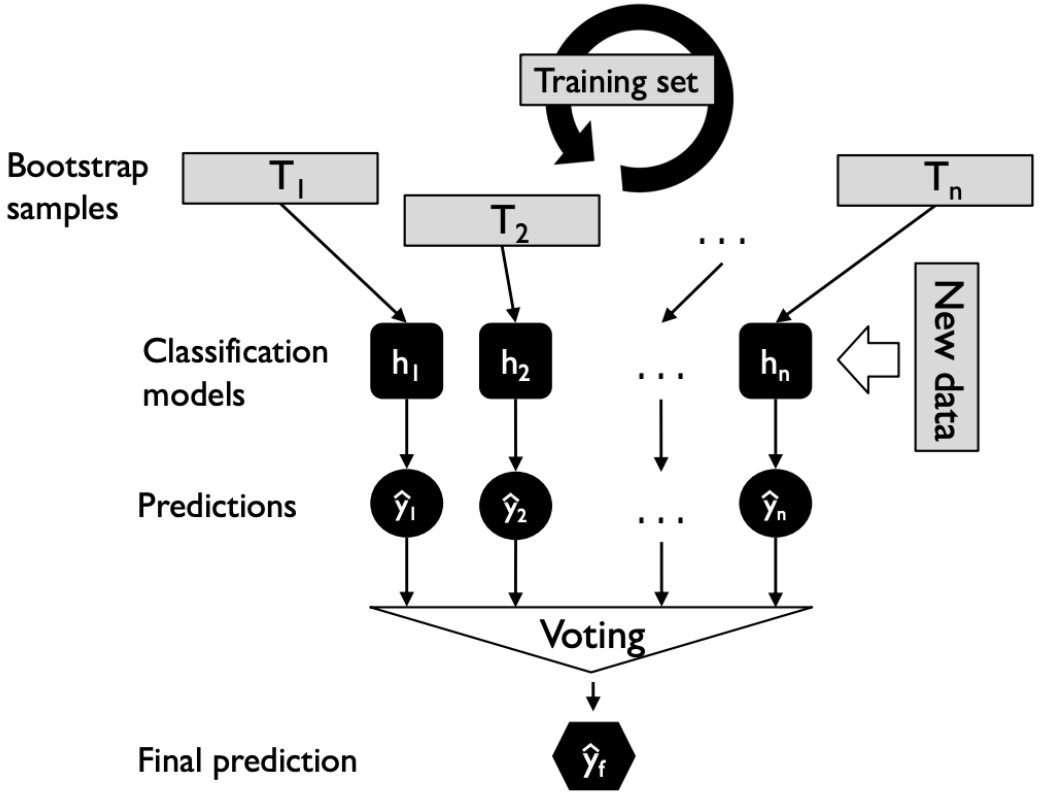
Bagging aggregates predictions across models, typically by **majority voting** in classification and **averaging** in regression, and uses the same learning algorithm (typically a decision tree algorithm) to fit models on different bootstrap samples.



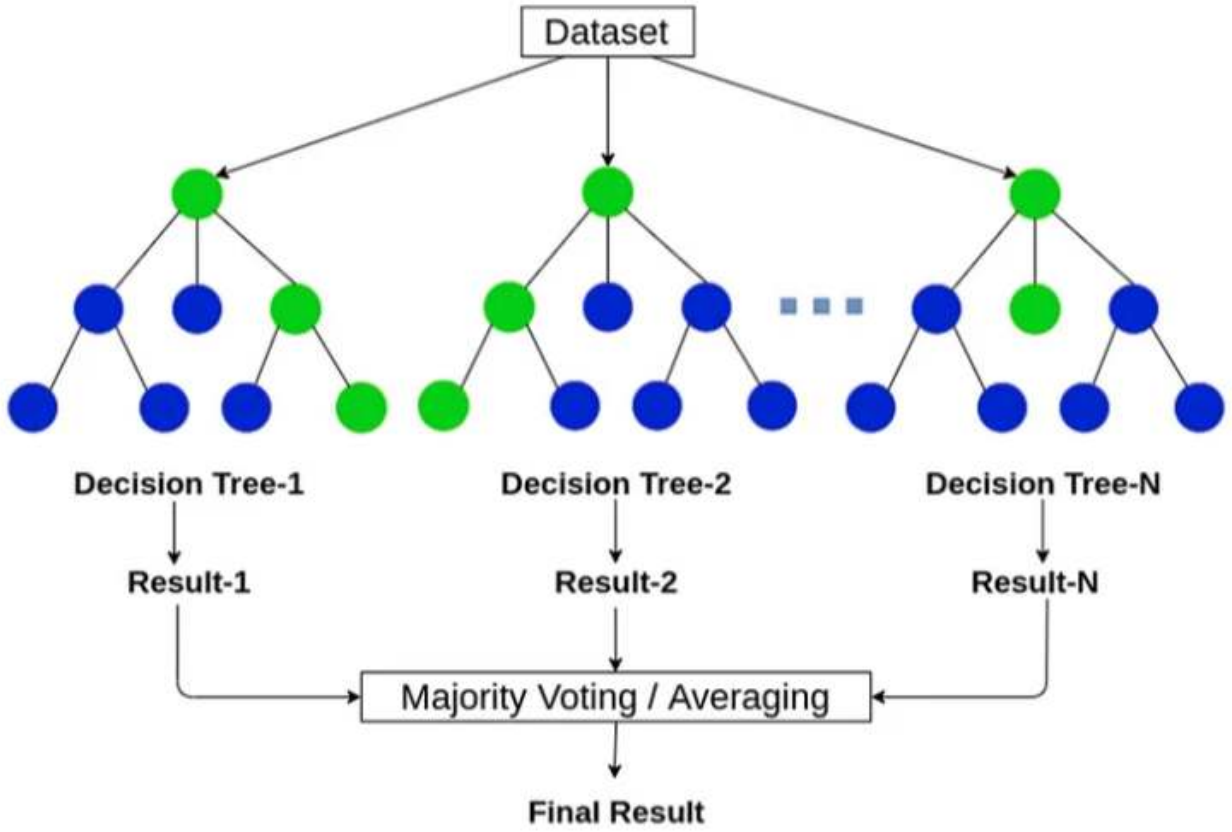
Bagging Algorithm

Algorithm 1 Bagging

1: Let n be the number of bootstrap samples
2:
3: **for** i=1 to n **do**
4: Draw bootstrap sample of size m, $\mathcal{D}_i$
5: Train base classifier h_i on $\mathcal{D}_i$
6: $\hat{y} = mode\{h_1(\mathbf{x}), ..., h_n(\mathbf{x})\}$



Random Forest



A **random forest uses bagging with decision trees**, but instead of growing the decision trees by basing the splitting criterion on the complete feature set, we use **random feature subsets**.

In random forests,

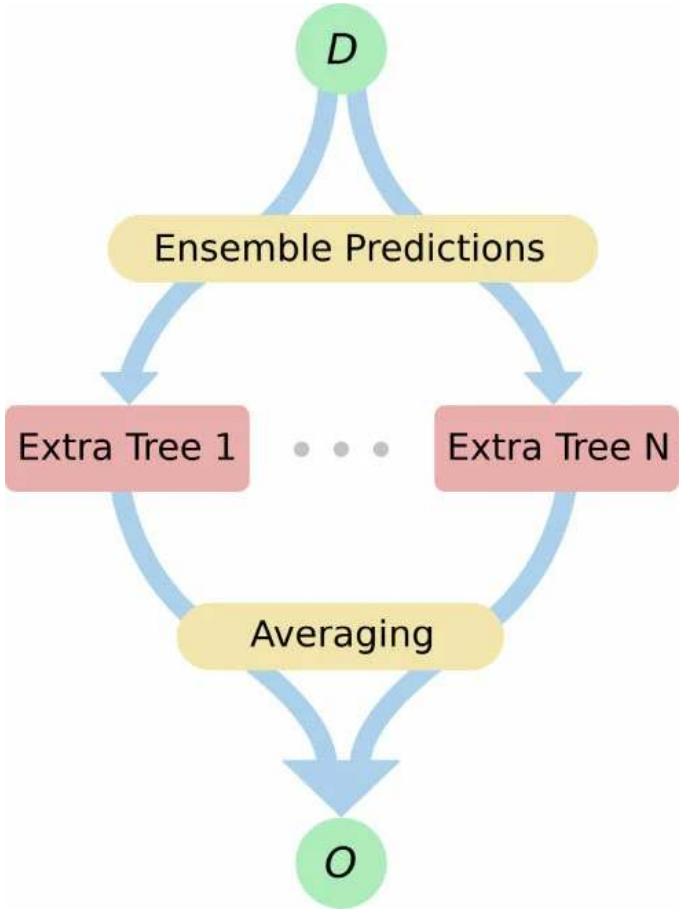
- we fit decision trees on different bootstrap samples,
- and in addition, for each decision tree, we select a random subset of features at each split to decide upon the optimal split.

Extremely Randomized Trees

A forest of extremely randomized trees (Extra Trees) introduces **even more randomness** than a random forest.

Compared with random forests:

- we typically use the whole training set for each tree instead of bootstrap samples,
- at each node, we still select a random subset of features, but the split thresholds are chosen at random rather than optimized exhaustively.
- Among these randomly generated candidate splits, we keep the one that gives the best impurity reduction.



Boosting

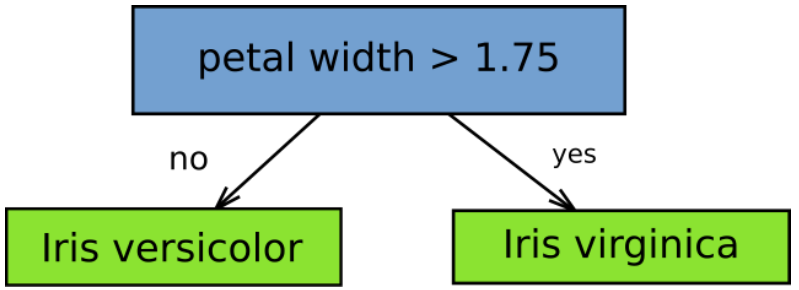
Types of Boosting

There are two broad categories of boosting: **Adaptive Boosting** and **Gradient Boosting**. They rely on the concept of boosting weak learners (e.g., decision tree stumps) to strong learners.

Boosting is an iterative, **sequential** process where new learners focus on current ensemble mistakes.

Adaptive and gradient boosting differ mainly regarding **how the weights are updated and how the classifiers are combined**.

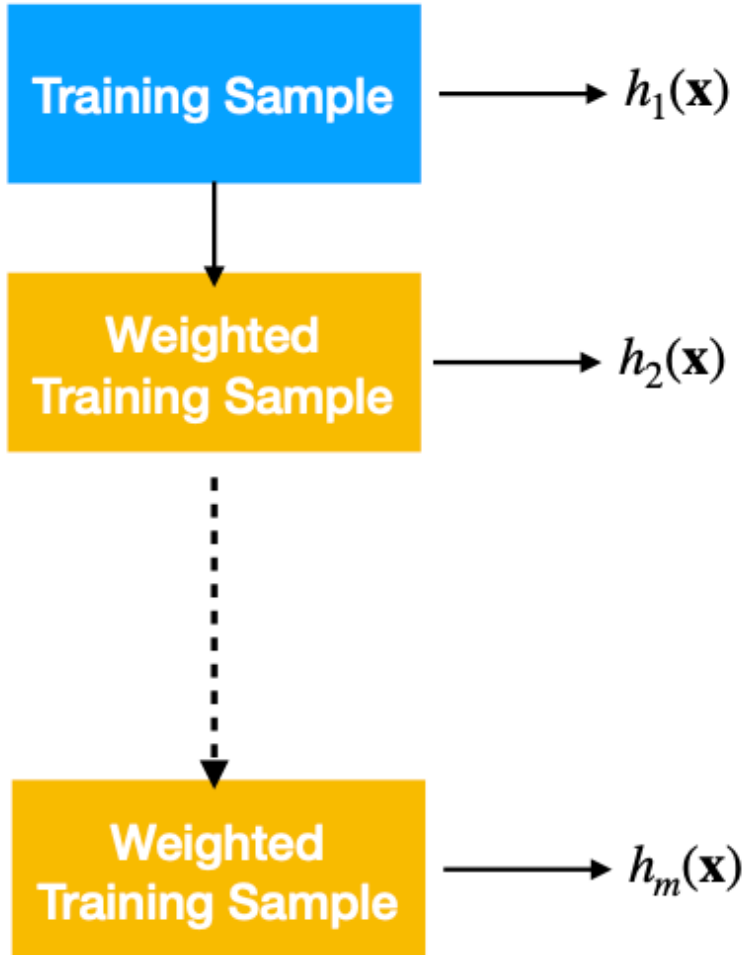
- In AdaBoost, examples are reweighted across iterations so that misclassified points receive more attention.
- In gradient boosting, each new learner is fit to the residual errors or gradients of the current ensemble.



Adaboost

The general boosting procedure for AdaBoost is as follows:

- Initialize a weight vector with uniform weights
- Loop:
  - Train a weak learner on the weighted training set (in some implementations, instead of the original training set, draw bootstrap samples with weighted probability)
  - Increase weight for misclassified examples
  - (Weighted) majority voting on trained classifiers



Adaboost

Chest Pain	Blocked Arteries	Patient Weight	Heart Disease
Yes	Yes	205	Yes
No	Yes	180	Yes
Yes	No	210	Yes
Yes	Yes	167	Yes
No	Yes	156	No
No	Yes	125	No
Yes	No	168	No
Yes	Yes	172	No

Given:

- a dataset with n instances;
- n corresponding labels;
- a number k of iterations

Initialize the sample weights as $w_i = 1/n$ for all i

Adaboost

Chest Pain	Blocked Arteries	Patient Weight	Heart Disease	Sample Weight
Yes	Yes	205	Yes	1/8
No	Yes	180	Yes	1/8
Yes	No	210	Yes	1/8
Yes	Yes	167	Yes	1/8
No	Yes	156	No	1/8
No	Yes	125	No	1/8
Yes	No	168	No	1/8
Yes	Yes	172	No	1/8

Given:

- a dataset with n instances;
- n corresponding labels;
- a number k of iterations

Initialize the sample weights as $w_i = 1/n$ for all i

For $r = 1, \ldots, k$:

Adaboost

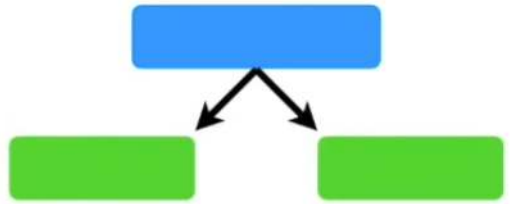
Fit a weak learner, e.g. a tree stump, on the **bootstrapped version of the dataset***, drawn using the sample weight as probability.

Compute the error, i.e., the sum of the weights corresponding to the misclassified samples:

$$\epsilon_r = \sum_{i=1}^{n} w_i \mathbf{1}[h_r(\mathbf{x}^{(i)}) \neq y^{(i)}]$$

Chest Pain	Blocked Arteries	Patient Weight	Heart Disease	Sample Weight
Yes	Yes	205	Yes	1/8
No	Yes	180	Yes	1/8
Yes	No	210	Yes	1/8
Yes	Yes	167	Yes	1/8
No	Yes	156	No	1/8
No	Yes	125	No	1/8
Yes	No	168	No	1/8
Yes	Yes	172	No	1/8

Now we need to make the first **stump** in the forest.



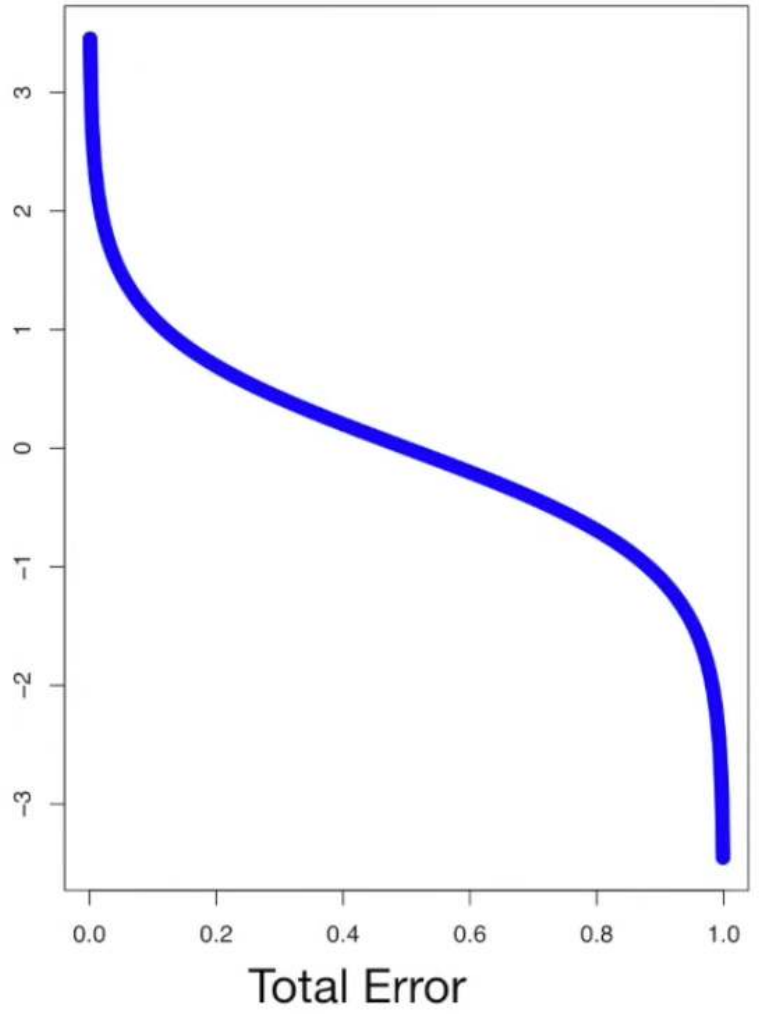
*More in general fit a weak learner on the weighted dataset. Bootstrap is one way of weighting it.

Adaboost

Compute the **amount of say**, α_r, i.e., how much a stump contributes in the ensemble:

$$\alpha_r = 1/2 \log\left(\frac{1-\epsilon_r}{\epsilon_r}\right)$$

- if the error is small α is positive
- if the error is big α is negative
- if the error is 0.5 α is zero



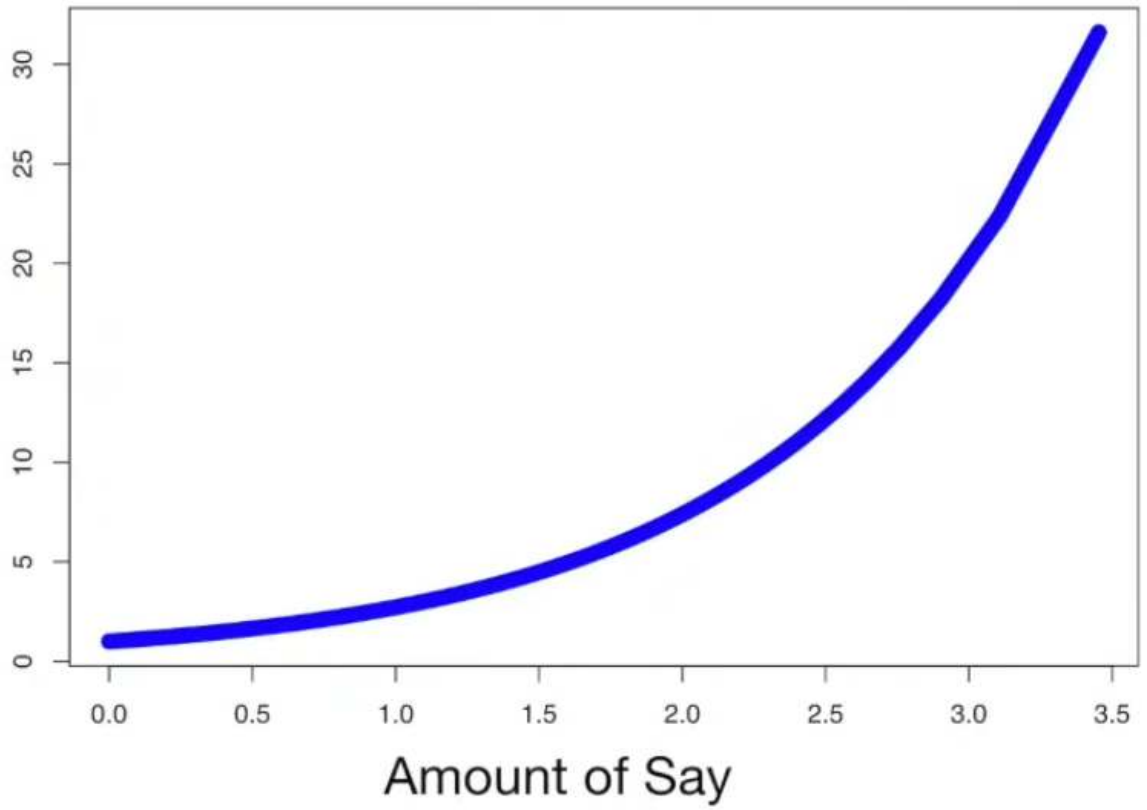
Adaboost

Compute the new sample weight, depending on if the stump is predicting the correct class for $\mathbf{x}^{(i)}$:

$$w_{\text{new}}^{(i)} = w_{\text{old}}^{(i)} \cdot \begin{cases} e^{-\alpha_r} & \text{if } h_r(\mathbf{x}^{(i)}) = y^{(i)} \\ e^{\alpha_r} & \text{if } h_r(\mathbf{x}^{(i)}) \neq y^{(i)} \end{cases}$$

If $\mathbf{x}^{(i)}$ is incorrectly classified by the stump, we want to increase its weight:

- Stump good in general, wrong prediction for $i \rightarrow$ big increase in sample weight for $\mathbf{x}^{(i)}$
- Stump bad in general, wrong prediction for $i \rightarrow$ small increase in sample weight for $\mathbf{x}^{(i)}$



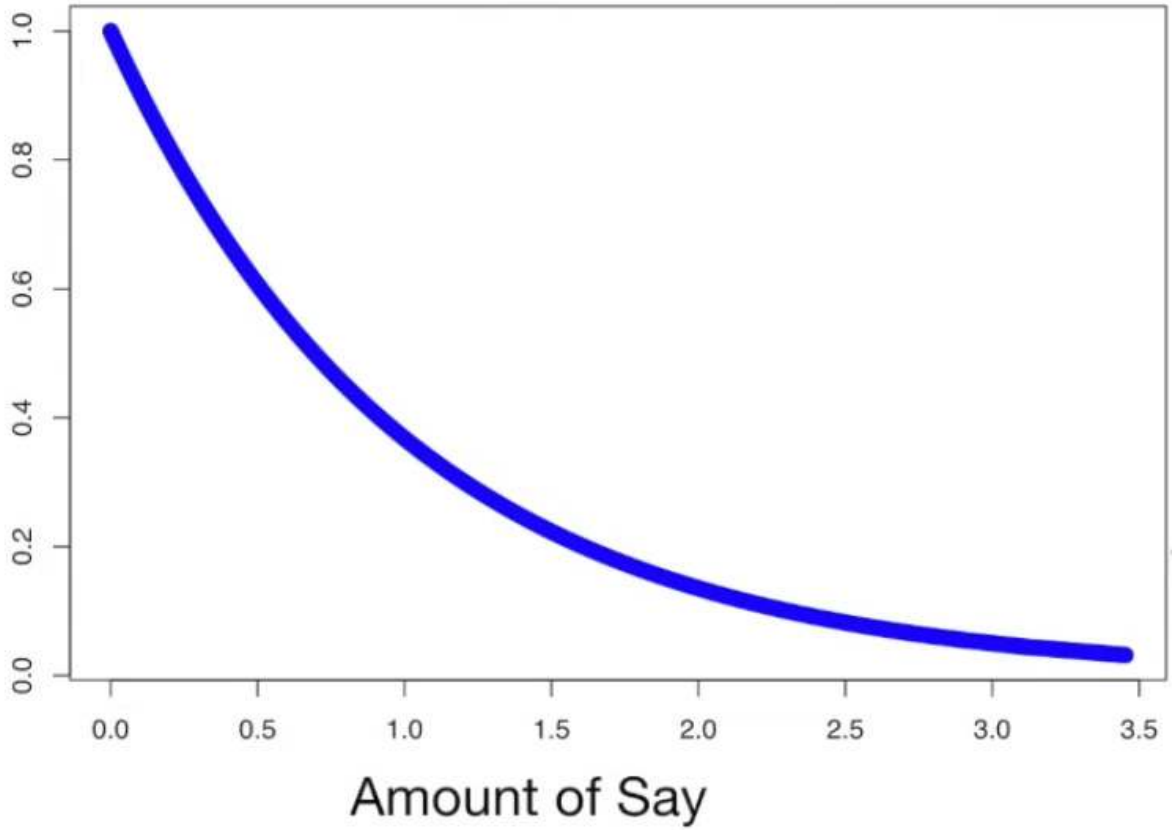
Adaboost

Compute the new sample weight, depending on if the stump is predicting the correct class for $\mathbf{x}^{(i)}$:

$$w_{\text{new}}^{(i)} = w_{\text{old}}^{(i)} \cdot \begin{cases} e^{-\alpha_r} & \text{if } h_r(\mathbf{x}^{(i)}) = y^{(i)} \\ e^{\alpha_r} & \text{if } h_r(\mathbf{x}^{(i)}) \neq y^{(i)} \end{cases}$$

If $\mathbf{x}^{(i)}$ is correctly classified by the stump, we want to decrease its weight:

- Stump good in general, correct prediction for $i \rightarrow$ big decrease in sample weight for $\mathbf{x}^{(i)}$
- Stump bad in general, correct prediction for $i \rightarrow$ small decrease in sample weight for $\mathbf{x}^{(i)}$



Adaboost

Chest Pain	Blocked Arteries	Patient Weight	Heart Disease	Sample Weight
Yes	Yes	205	Yes	0.07
No	Yes	180	Yes	0.07
Yes	No	210	Yes	0.07
Yes	Yes	167	Yes	0.49
No	Yes	156	No	0.07
No	Yes	125	No	0.07
Yes	No	168	No	0.07
Yes	Yes	172	No	0.07

Weights are normalized to sum to one, and the procedure is repeated for k steps or until convergence.

Adaboost

In binary classification with labels $y \in \{-1, +1\}$, the final prediction is a weighted vote of the stumps, based on their amount of say:

$$h_{\text{final}}(\mathbf{x}) = \text{sign}\left(\sum_{r=1}^{k} \alpha_r h_r(\mathbf{x})\right)$$

AdaBoost can be extended to the multiclass case.

Gradient Boosting

The main idea behind gradient boosting can be summarized via the following three steps:

1. Construct a "dummy" regressor that just predicts the average of the target (e.g. a tree root node)
2. Fit the next tree to the pseudo-residuals of the current ensemble.
3. Combine the initial predictor with the trees learned in the subsequent steps.

For the sake of simplicity, the following slides will illustrate gradient boosting using a squared-error regression example, but the idea is similar for other losses or classification tasks.

1. The Root Node

The first step is to construct the root node, h_0. For squared-error regression, the prediction at a node is the average target value of the training examples in that node; for the root:

$$\hat{y} = \frac{1}{n} \sum_{i=1}^{n} y^{(i)}$$

Height (m)	Favorite Color	Gender	Weight (kg)
1.6	Blue	Male	88
1.6	Green	Female	76
1.5	Blue	Female	56
1.8	Red	Male	73
1.5	Green	Male	77
1.4	Blue	Female	57

The average in this case is 71.2

2. Pseudo-residuals

Height (m)	Favorite Color	Gender	Weight (kg)	Residual
1.6	Blue	Male	88	16.8
1.6	Green	Female	76	4.8
1.5	Blue	Female	56	-15.2
1.8	Red	Male	73	1.8
1.5	Green	Male	77	5.8
1.4	Blue	Female	57	-14.2

In the first step of the example, the residuals are $y^{(i)} - 71.2$

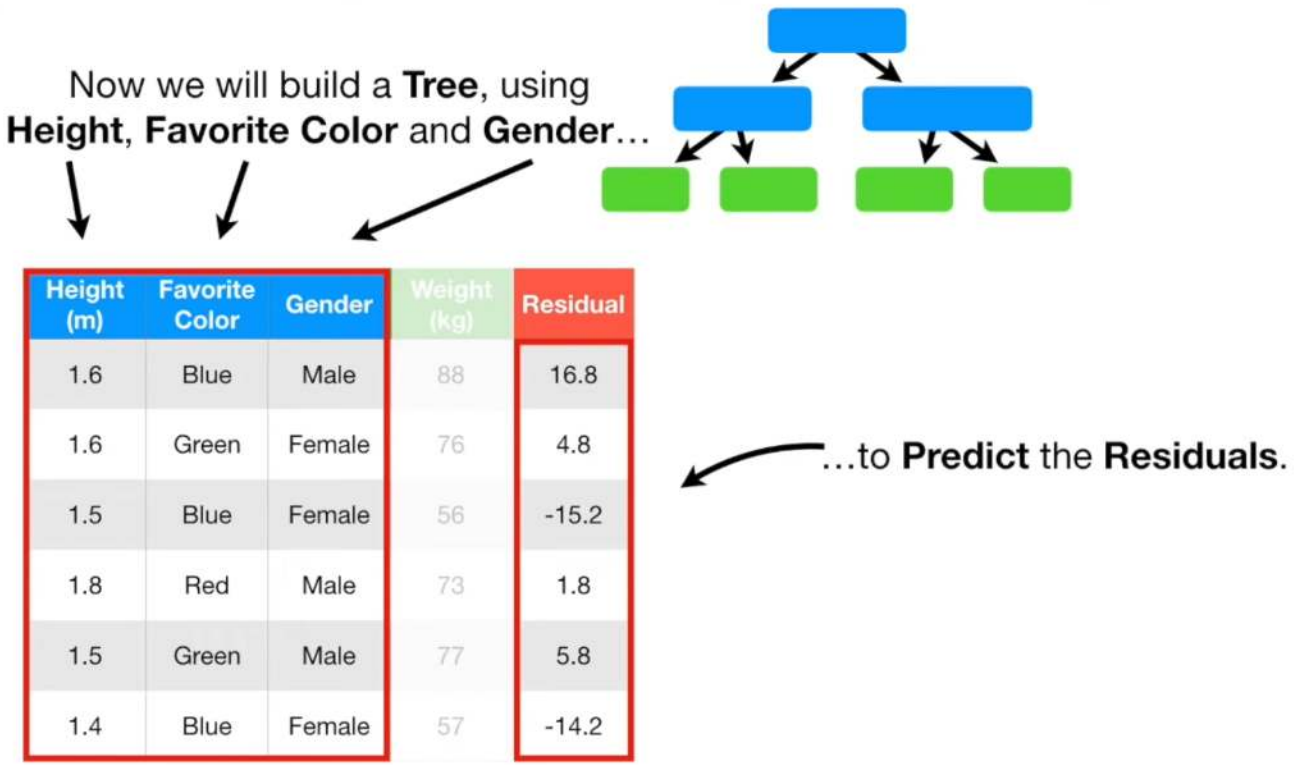
Then, we compute the so-called pseudo-residuals. For squared-error loss, these pseudo-residuals coincide with the ordinary residuals, i.e., the difference between the true target value and the predicted target value:

$$r^{(i)} = y^{(i)} - \hat{y}^{(i)}$$

After computing the residuals for all training examples, we can add them as a new column of our dataset.

2. New Decision Tree

We fit a new decision tree, h_1, to the residuals. Usually, we limit the depth of this tree, but it's not necessarily a stump as in AdaBoost.



Height (m)	Favorite Color	Gender	Weight (kg)	Residual
1.6	Blue	Male	88	16.8
1.6	Green	Female	76	4.8
1.5	Blue	Female	56	-15.2
1.8	Red	Male	73	1.8
1.5	Green	Male	77	5.8
1.4	Blue	Female	57	-14.2

3. Boosting the old predictor

The new prediction is the sum of the initial predictor and the new tree:

$$\hat{y}_{\text{new}} = h_0(\mathbf{x}) + \alpha h_1(\mathbf{x})$$

where α is the learning rate. We then repeat again from step 2, by computing the new pseudo-residuals, until a stopping criterion is met.

The final prediction is:

$$\hat{y}_{\text{final}} = h_0(\mathbf{x}) + \alpha \sum_{t=1}^{T} h_t(\mathbf{x})$$

XGBoost

XGBoost (Extreme Gradient Boosting) is one of the most popular implementations of gradient-boosted decision trees.

It extends standard gradient boosting with several practical improvements:

- **regularization** terms to reduce overfitting,
- efficient handling of sparse data,
- handling of missing values,
- parallelized and optimized training,
- shrinkage and subsampling for better generalization.

XGBoost is widely used in machine learning competitions and applied tabular learning because it often provides an excellent balance between **predictive accuracy**, **flexibility**, and **robustness**.

LightGBM

LightGBM is an implementation of **gradient boosting with decision trees** that mainly changes **how the boosted trees are grown**.

Its main peculiarities are:

- **histogram-based** split search for faster training,
- **leaf-wise (best-first)** tree growth instead of level-wise growth,
- good scalability to **large tabular datasets**,

LightGBM keeps the same boosting principle, but makes the **tree-building stage much faster and more aggressive**.

CatBoost

CatBoost is a **gradient boosting** method specialized for **tabular data with categorical features**.

Its main peculiarities are:

- direct handling of **categorical features**,
- **ordered boosting**, which reduces target leakage and prediction shift,
- default use of **symmetric (oblivious) trees**, where all nodes at the same depth use the same split.

So, compared with the original boosting idea, CatBoost keeps boosting intact but changes it to be **safer for categorical data** and uses a **more constrained tree structure**.

Feature Importance

Feature importance in tree-ensemble models can refer to one of the following:

- How often was a feature used? How many times? (frequency). Note that this will be biased towards high-cardinality features that can be picked in a greater number of splits, e.g. a variable with 20 categories instead of a binary one.
- How much of a “root” component is a feature? How many instances are influenced by a split on it? (cover)
- How effective is a feature when it is used? Even if it is used very few times at leaf-level, it may give an effective contribution to our classifications or predictions (gain)