Machine Learning: Ensemble Methods (bagging and boosting)

Ensemble Methods

- An ensemble is a set of classifiers/regressors (either different algorithms or different settings of the same algorithm, or the same algorithm on different samples of the dataset) that learn a target function, and their individual predictions are combined to classify new examples.
- Ensembles generally improve the generalization performance of a set of individual models on a domain.
- Based on the following idea: ***A large number of relatively uncorrelated models operating as a committee will outperform any of the individual constituent models.***

Example:

Weather Forecast

100% CORRECT!

GROUND TRUTH							
PREDICTOR1							
PREDICTOR2							
PREDICTOR3							
PREDICTOR4							
PREDICTOR5							
Combine							

Why to use Ensemble Methods?

- Statistical reasons:
  - A set of classifiers with similar training performances may have different **generalization performances**
  - Combining outputs of several classifiers **reduces the risk of selecting a poorly performing (weak) classifier**.
- Too large volumes of data:
  - If the amount of data to be analyzed is too large, a single classifier may not be able to handle it; train **different classifiers on different partitions of data**.
- Too little data:
  - Ensemble systems can also be used when there is too little data by the use of **resampling techniques**.

Why to use Ensemble Methods?

- The the error of a learning algorithm (will see later evaluation) has three components: the noise, the bias, and the variance:

$$\textbf{Error}(x) = \textbf{Bias}(x)^2 + \textbf{Variance}(x) + \textbf{Noise}(X)$$

  - ➢ The **noise** is the **irreducible error** (random errors in the data that can't be eliminated, for example due to corrupted input, data entering errors..)
  - ➢ The **bias** is the **systematic error that the learning algorithm is expected to make** due to, e.g., architectural choices (e.g. if we use a Perceptron for data that are not lineraly separable) or to insufficient/unrepresentative training data
  - ➢ The **variance** measures the **sensitivity of the algorithm to the specific training set and/or hyper-parameters used** (algorithms can be more or less robust to such variations).

Ensembles may help reducing both bias and variance (except noise, which is irreducible error)

The relation between error, bias and variance

- $X, Y, \hat{Y}$ are *random variables* describing the distribution of values for instances x, and their ground truth and predicted values f(x) and h(x)
- $h(X)$ is and estimator (hypothesis) of the true (unknown) function $f(X)$, generated by some model M

  $Y = f(X) + \varepsilon$ y ∈ Y values are generated by the (unknown) $f(X)$ plus «some» random error ε

  $MSE = E\left[\left(h(X) - f(X)\right)^2\right] = E\left[\left(\hat{Y} - f(X)\right)^2\right]$ **mean square error** is is the expected (mean) value of the square error over the entire distribution

  Bias^2(h,f)=$\left(E[h(X)] - f(X)\right)^2 = E^2[h(X)] + f(X)^2 - 2E[h(X)]f(X)$

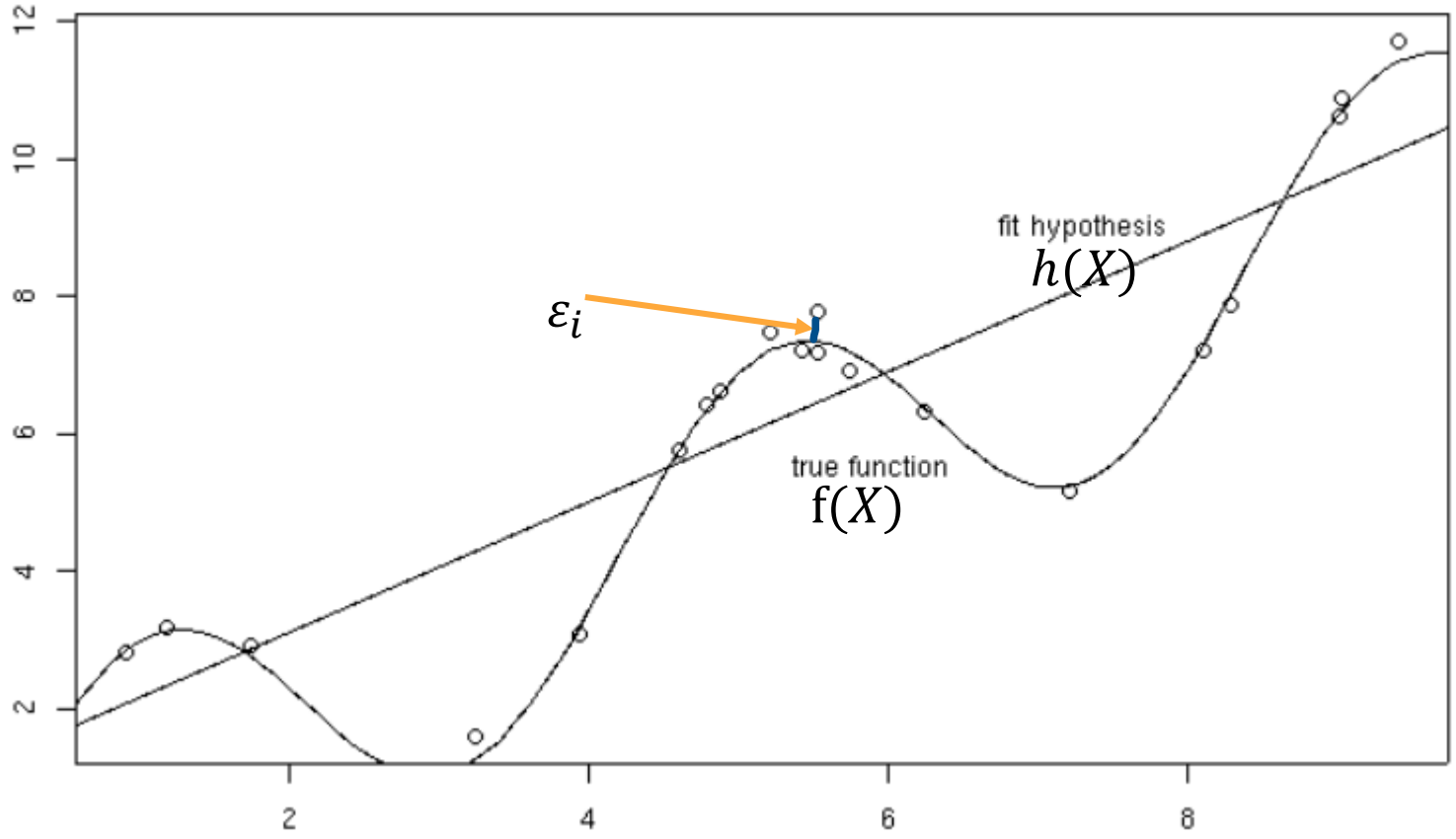
  Bias is the difference between the mean predicted values of our model and the correct values which we are trying to predict

  Var(h)=$E[(h(X) - E[h(X)])^2]$ variance is the expected variability of the model around its mean

$MSE = E\left[\left(h(X) - f(X)\right)^2\right] = E[(h(X) + E[h(x)] - E[h(X) - f(X)])^2]$=….= $\left(E[h(X)] - f(X)\right)^2$ + $E\left[\left(h(X) - f(X)\right)^2\right] =$
$\left(E\left(\hat{Y}\right) - Y\right)^2 + \mathrm{E}\left[\left(\hat{Y} - Y\right)^2\right] + \varepsilon = Bias^2 + \mathrm{Variance} + \mathrm{irreducible\ error}$

True function, model hypothesis, random error

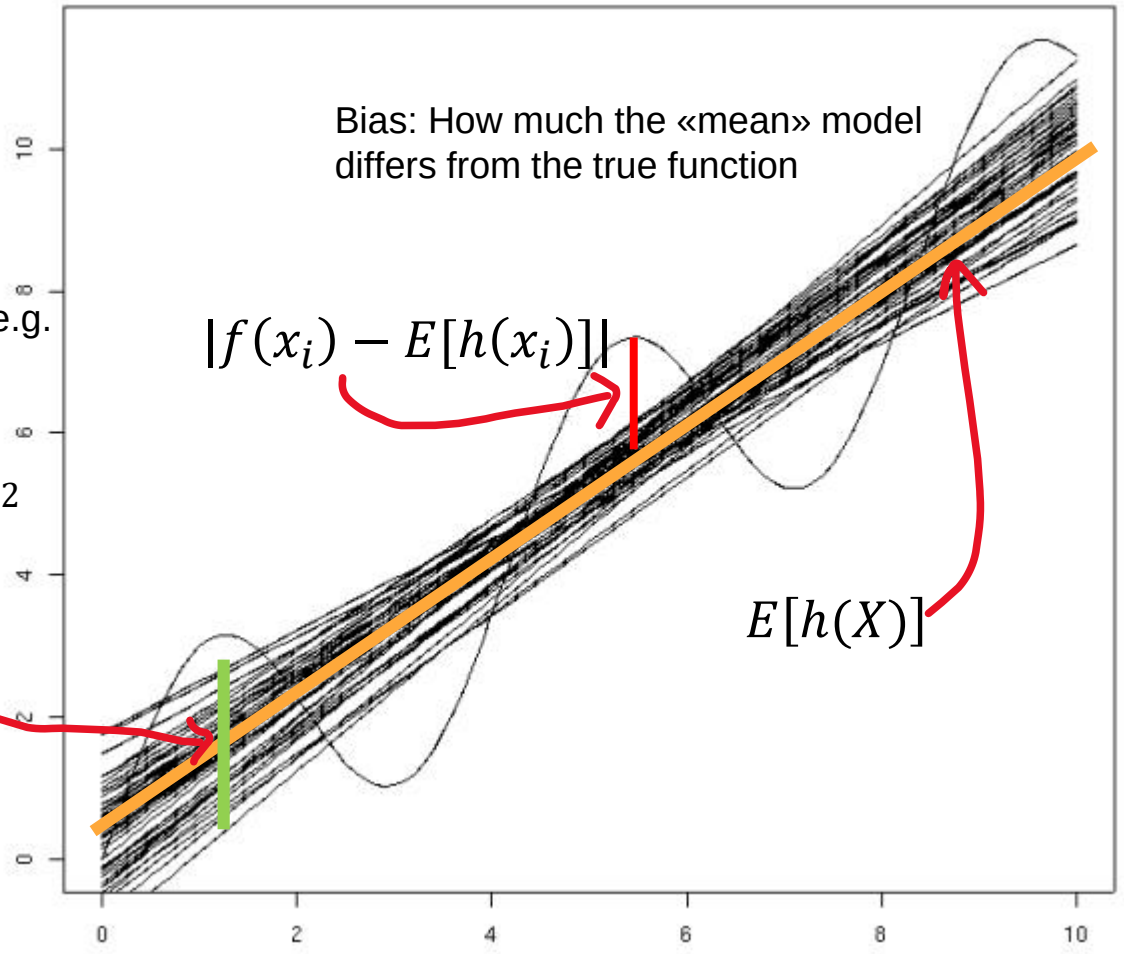
- $Y = f(X) + \varepsilon$



Bias and variance, the intuition

Variance: How much the model varies e.g. with different runs (hyperparameters) or data samples

$$\sum_i (h_i(x_k) - E[h(x_k)])^2$$



Why to use Ensemble Methods?

- When combining multiple **independent** and **diverse** decisions each of which is at least more accurate than *random guessing*, the variance is reduced and so does the error rate.

$$Var\left(Ensamble\left(h_i(x,D)\right)\right) = \frac{\sum_m Var(h_i(x,D_i))}{m}$$

- Where h_i(x,D) is the hypothesis generated by i-th learner of an ensamle.
- In practice, the idea is the following:

  If we use “simple” models trained on smaller samples, the **individual** variance of each hypothesis **can** be higher than for a more complex learner, but still in most cases the **ensamble variance** is lower than if we use one single complex learner (will discuss later about bias)

Example

- Suppose there are 25 “simple” classifiers
  - ➢Each classifier has an average error rate, ε = 0.35 (which is a **mid-high** rate)
  - ➢Assume classifiers-generated hypotheses $h_i(x)$ are statistically **independent, and final class is predicted with majority voting**
  - ➢The probability that the **ensemble classifier** makes a wrong prediction (it is wrong if at least 13 out of 25 make the wrong prediction):

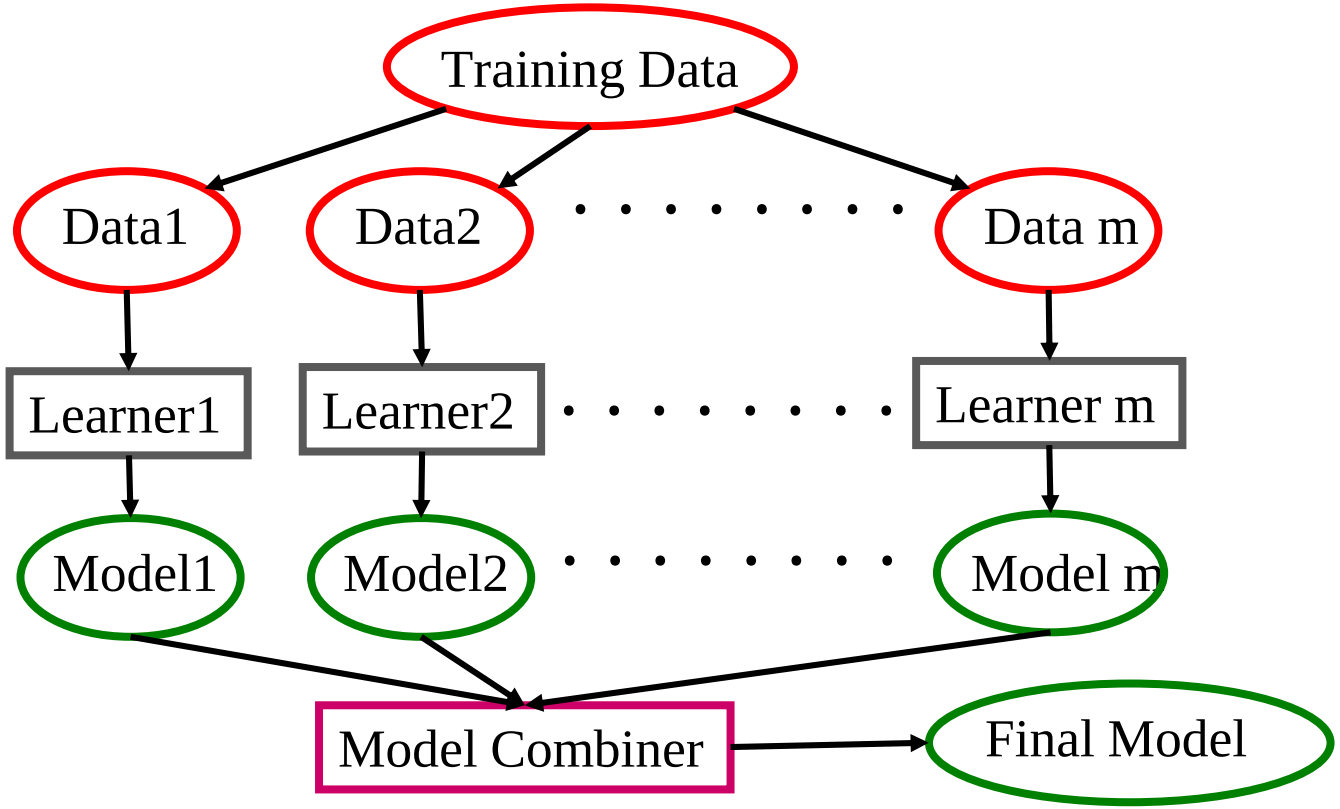
$$\sum_{i=13..25} \binom{25}{i} \varepsilon^i (1-\varepsilon)^{25\text{-}i} = 0{,}6$$

All possible ways, given a set of 25 classifiers, of having *i* (with *i*>12) classifiers that make an error, and 25-i producing a correct prediction

If classifiers are **independent**, the probability that the ensemble makes an error is very low!

Learning Ensembles

- Learn multiple alternative models using **different training data** or **different learning algorithms**.
- **Combine** decisions of multiple definitions, e.g. using weighted voting.



Methods for Constructing Ensembles

- **By manipulating the training set**: Create multiple training sets by resampling the original data according to some sampling distribution.
- **By manipulating the input features**: Choose a subset of input features to form each training set.
- **By manipulating the class labels**: Transform the training data into a binary class problem by randomly partitioning the class labels into two disjoint subsets (e.g. for 4 labels: (AB) (CD)).
- **By manipulating the learning algorithm**: Manipulate the learning algorithm to generate different models (e.g. different hyperparameters)

Homogeneous Ensembles

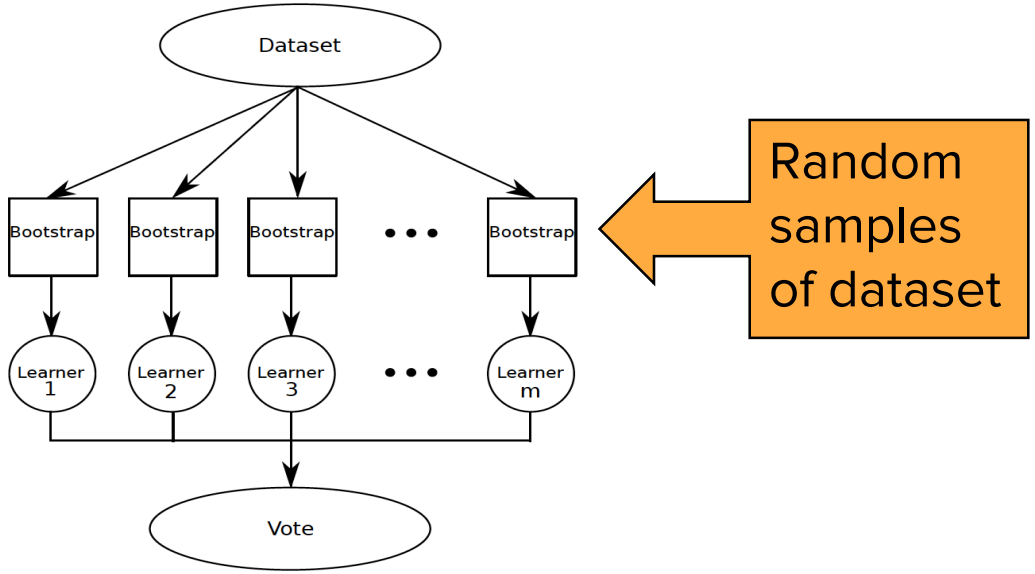
- Use a **single, arbitrary learning algorithm** but **manipulate training data** to make it learn multiple models $h_1(x)$ $h_2(x)$.. $h_m(x)$
  - Data1 ≠ Data2 ≠ … ≠ Data m
  - Learner1 = Learner2 = … = Learner m

- Different methods for **changing training data**:
  1. **Bagging**: Resample data (Useful to reduce the **variance**)
  2. **Boosting**: Re-weight data (Useful to reduce the **bias**)

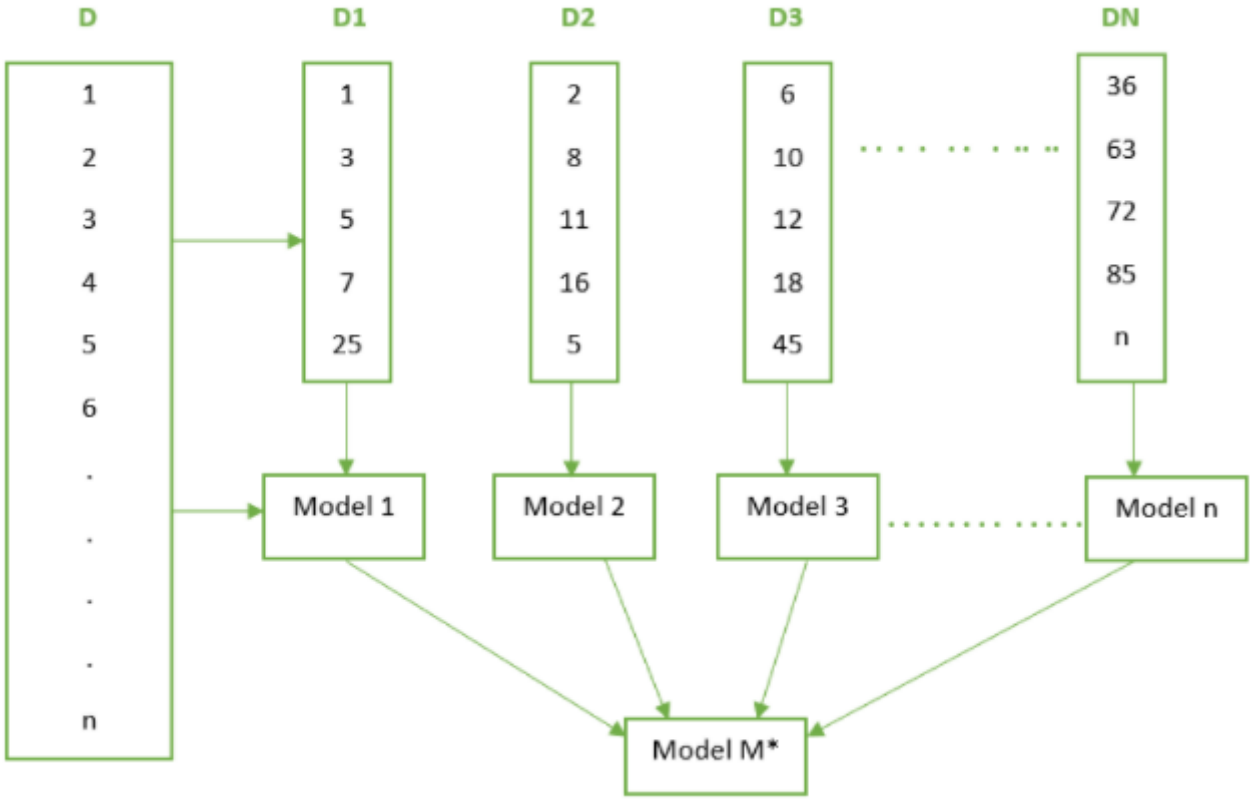
Bagging

A high variance for a model is not good, suggesting its performance **is sensitive to the training data provided** (low generalization power). So, even if more training data is provided, the model may still perform poorly. And, this may not even reduce the variance of our model.

- **Solution**: BAGGING, a shorthand for the combination of **B**ootstrapping and **agg**regat**ing**



The basic idea of bagging with bootstrap

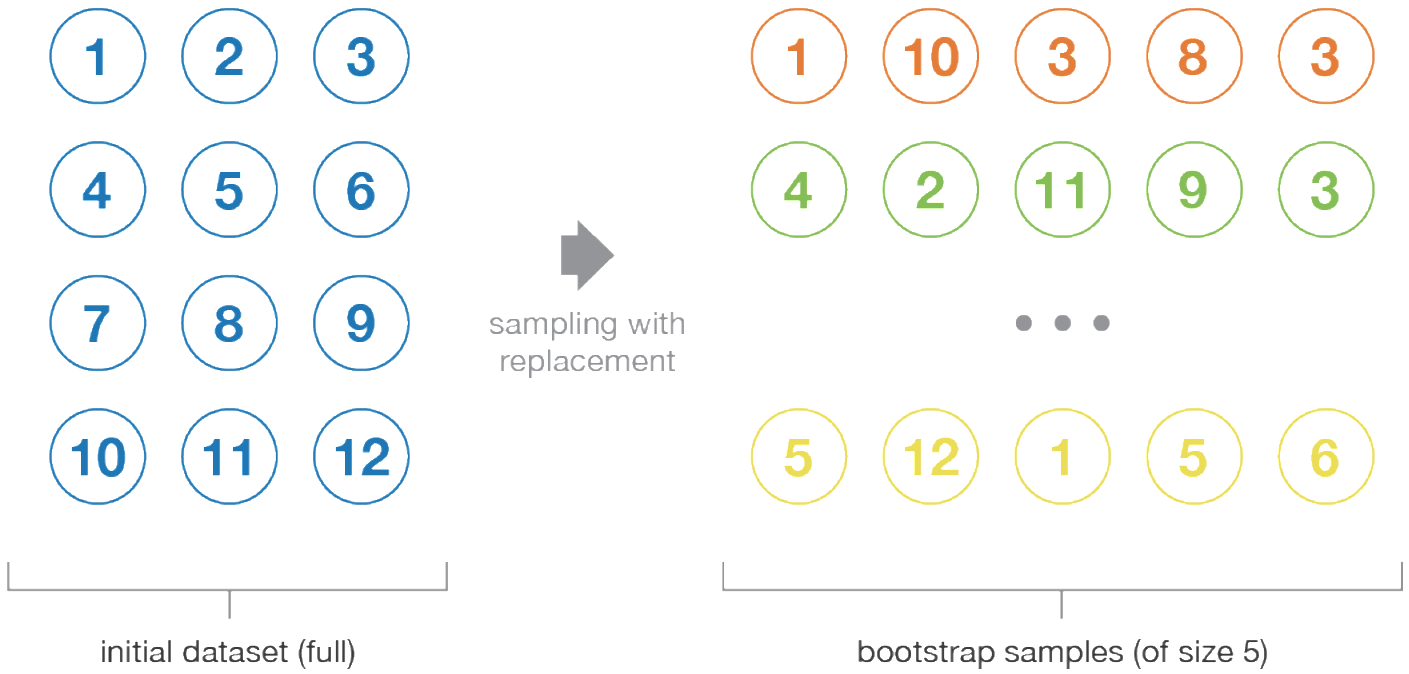


Bagging: Bootstrapping

Bootstrapping is a method to help **decrease the variance** of the classifier and reduce overfitting, by resampling data from the training set. The ensamble model created should be less overfitted than a single individual model.

- **How**: Each individual classifiers randomly extracts a sample of **m** instances over the training set of **n** instances with **replacement** (instances are put back in the urn, therefore **they can be sampled more than one time**). If the sample has the same cardinality as the original set **m=n**, is called «the 0.632 bootstrap».
- **When**: Effective method in limited data contexts (high variance, risk of overfitting).

Bootstrapping



Example with n=12 and m=5 (note the effect of replacement)

Bagging: Bootstrapping

- Instances that are not extracted during bagging, **are used for testing**
- Each instance x out of n has probability of $(1/n)^m$ of being selected in a training sample of m instances, and $(1 – 1/n)^m$ of being left as test data, in each bagging round.

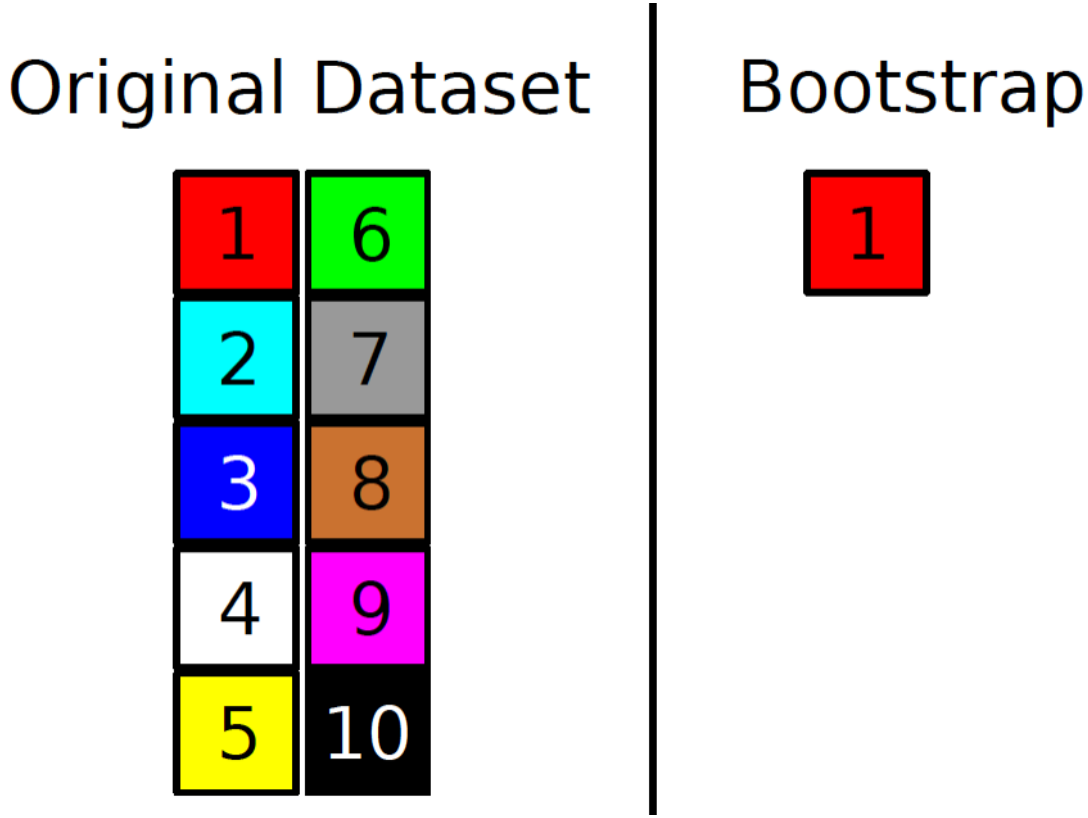
Data ID Training Data

Original Data	1	2	3	4	5	6	7	8	9	10
Bagging (Round 1)	7	8	10	8	2	5	10	10	5	9
Bagging (Round 2)	1	4	9	1	2	3	2	7	3	2
Bagging (Round 3)	1	8	5	10	5	5	9	6	3	7

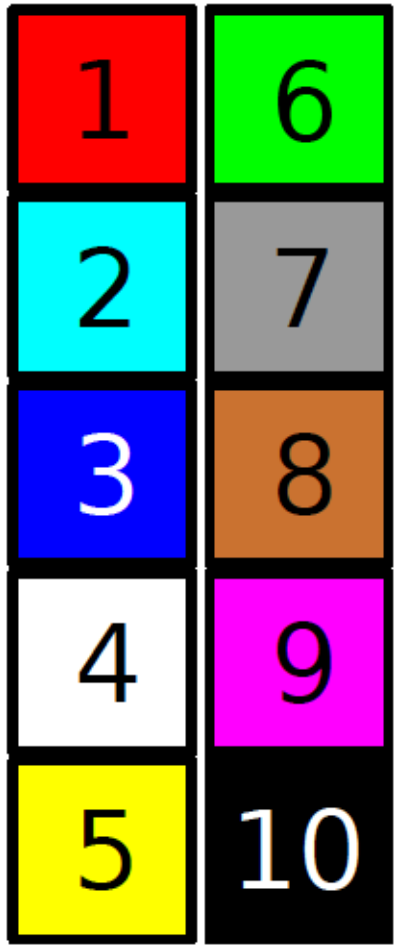
Original training dataset has n=10 instances. **n=m** in this example

Example (n=m)

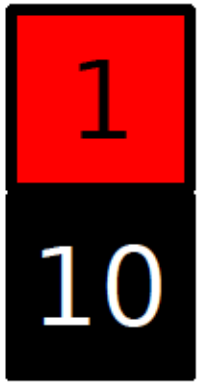
Each instance has a probability p=1/n of being extracted out of n instances at each extraction. Since extraction is “with **replacement**” (the instance is put back in the urn after having been extracted) the probability is always the same at each extraction.



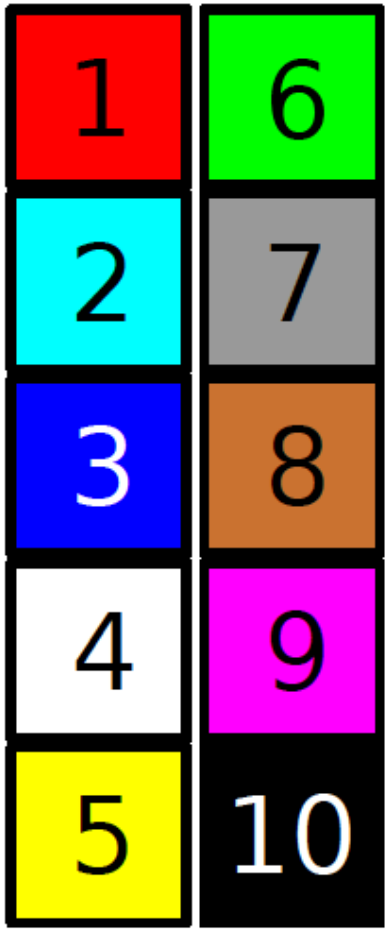
Original Dataset



Bootstrap

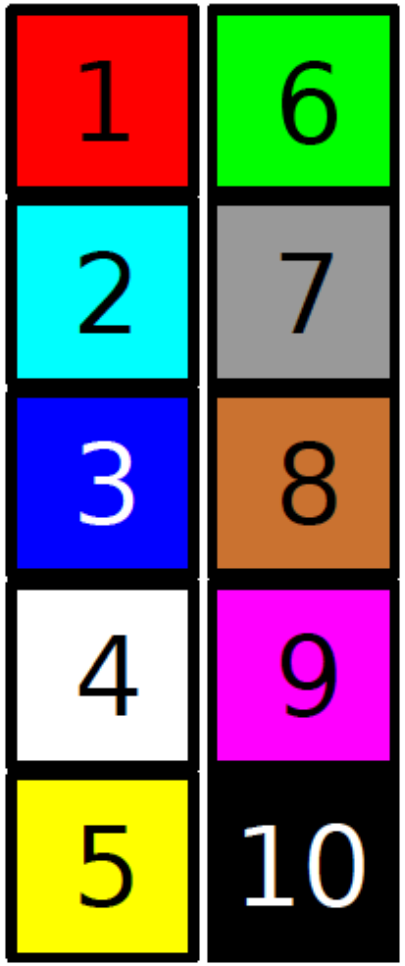


Original Dataset



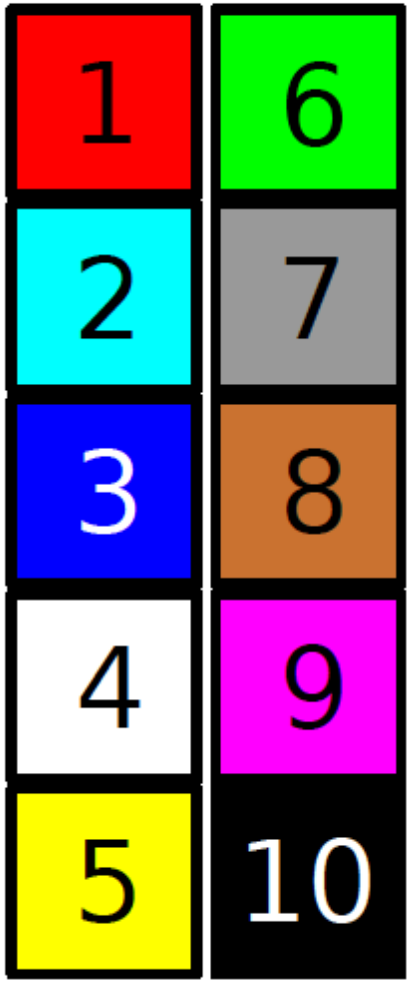
Bootstrap

Original Dataset



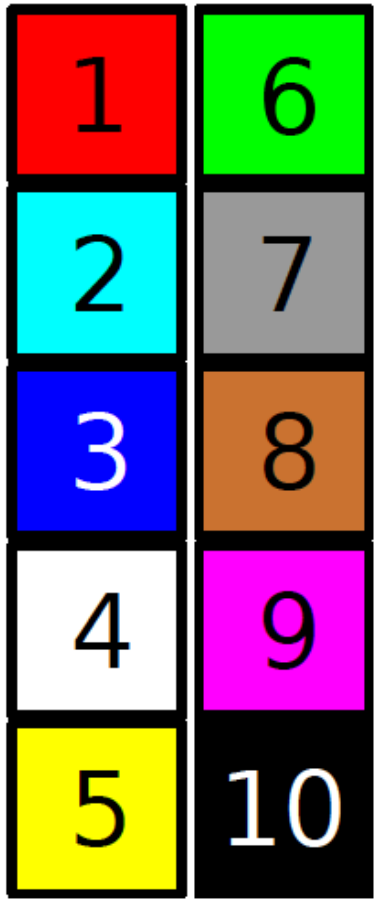
Bootstrap

Original Dataset

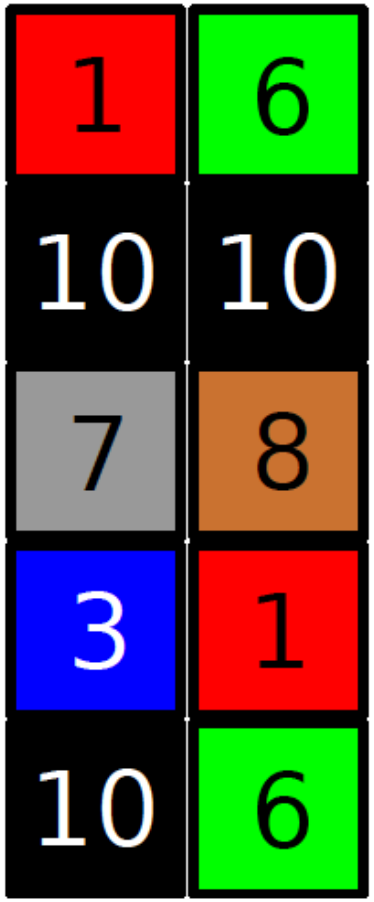


Bootstrap 5

Original Dataset



Bootstrap



7

Training set

Test set

Original Dataset	Bootstrap	Unselected
1	1	2
6	6	
2	10	4
7	10	
3	7	5
8	8	
4	3	9
9	1	
5	10	
10	6	

The 0.632 bootstrap

- Each example in the original dataset has a selection probability of 1/n on n samples
- If n=m, on average, 36.8% of the data-points are left unselected and can be used for testing

Why?

The 0.632 bootstrap

This method is also called the *0.632 bootstrap*

- If we have **n** instances, each instance has a probability **1/n** of being **picked** and (**1-1/*n)*** of ***not*** being **picked** at each extraction
- If **m=n**, its probability of ending up in the test data (= not being selected in any of n extractions) is:

$$\left(1-\frac{1}{n}\right)^{n} \approx e^{-1} = 0.368$$

- This means the training data will contain approximately 63.2% of the instances

Example of Bagging

We aim to learn a classifier C(x) in $\Re^1$ (x is a *continuous* variable). Assume that the "real" (unknown) classification function f(x) is:

- In this example data is not linearly separable, a classifier for this data must learn a **decision region**, e.g.:

  IF T_1<= x <=T_2 then C
  else not(C)

  ➢ In our example, the "true" (unknown) values for decision boundaries are $\mathbf{T_1}$=0.3 and $\mathbf{T_2}$=0.8.

Goal: find a collection of 10 **simple thresholding (=linear)** classifiers that can **collectively** classify correctly the data.

i.e. , each classifier $\mathbf{C_i}$ learn a single **threshold $\mathbf{T_i}$** such that:

IF x<=T_i then C
else not(C)

Training set

This is the training set: we have 10 pairs (x, C(x))

x	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Y=C(x)	1	1	1	0	0	0	0	1	1	1

We now create 10 samples of this data with bootstrapping, and on any sample, we train a “simple” threshold classifier.

Remember: Each sample is one training set (bagging round) of size 10. This means that for 10 times (bagging rounds) we sample 10 instances from the original dataset **with replacement**. The extracted instances in a round(i) are used to train the i-th learner h_i, and the non-extracted instances are used for testing.

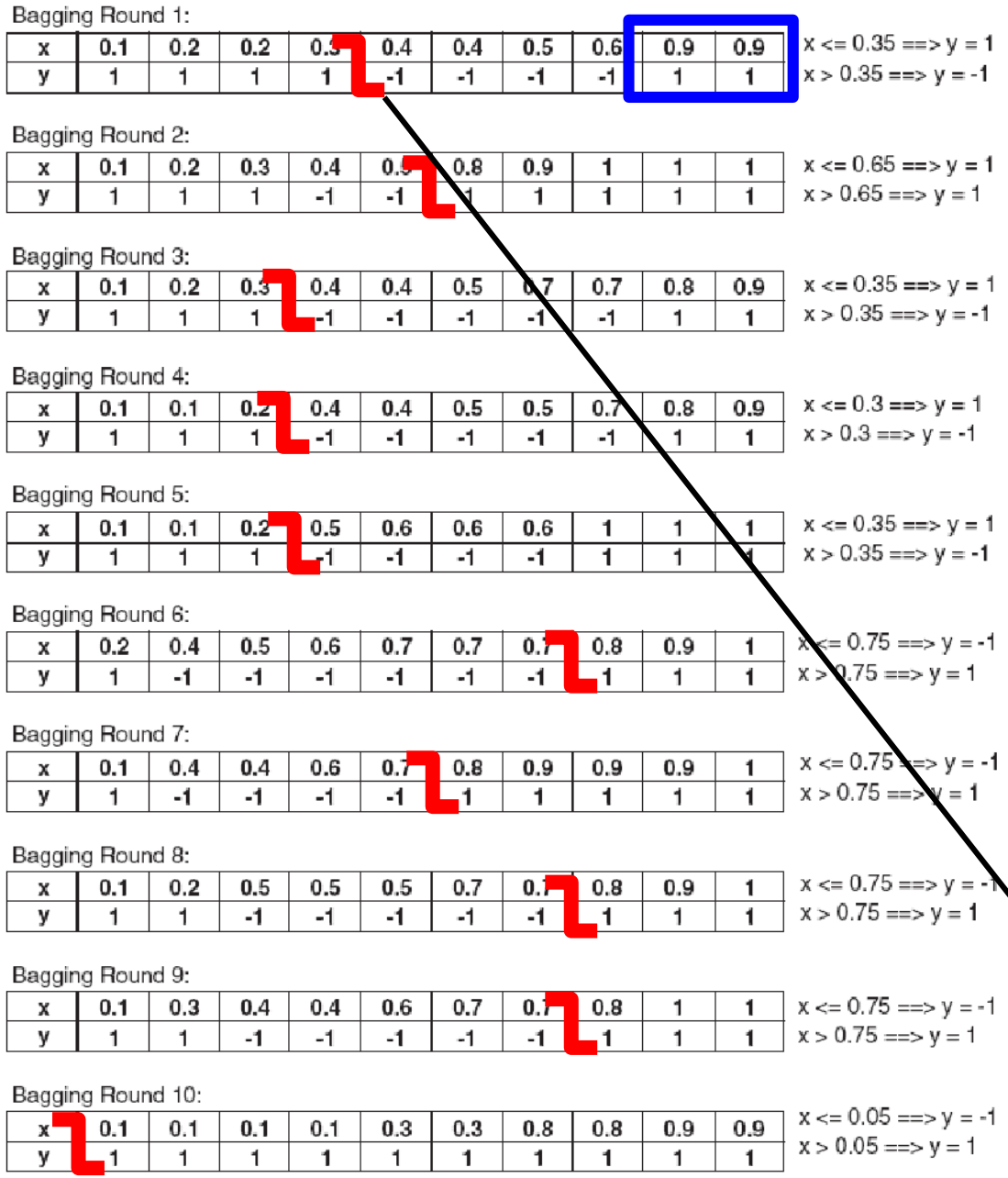


Figure 5.35. Example of bagging.

Note: In each round the same training example can be extracted more than one time, and some examples are not extracted.

Note that each classifier is **does not correctly classifies** some of the training data!
E.g. classifier 1 is wrong on the last two items of the "sampled" training set:
c(0.9) = -1 (and is instead 1)

For each bagging, we show the threshold learned by each classifier

Bagging: Aggregation

- In the previous example, given an initial training set of 10 examples, we bag the data 10 times and we learn **10 threshold classifiers C_i (i=1,..., 10),** (each of our $h_i(x)$) has an expected error rate $E[h_i(x) - f(x)] = \varepsilon_i$
- We then need to **combine** these results (**ensemble method**)
- There exists several methods to determine the final score, e.g.:

1. Majority voting
2. Combining (averaging) classification functions

Bagging: Aggregation

Majority voting

Majority voting (Hard voting): we just need a majority of classifiers to determine what the result could be.

A simple version of Majority voting for **binary classifiers** (with values +1, -1):

IF *sign*($\Sigma_i C_i(x_j)$) = +, then $C(x_j) = 1$

➢ This means: if majority says "1" (if $\sum(+1) > \sum(-1)$) then, predicted class is 1 (the sign of the sum of prediction is positive)

Bagging with Majority Voting

IF applied to training data

Accuracy of ensemble classifier: 100% 😊

$$If\ sign(\sum C_i(x)) = 1\ then\ C(x) = 1$$

Round	x=0.1	x=0.2	x=0.3	x=0.4	x=0.5	x=0.6	x=0.7	x=0.8	x=0.9	x=1.0
1	1	1	1	-1	-1	-1	-1	-1	-1	-1
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	-1	-1	-1	-1	-1	-1	-1
4	1	1	1	-1	-1	-1	-1	-1	-1	-1
5	1	1	1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	-1	-1	1	1	1
7	-1	-1	-1	-1	-1	-1	-1	1	1	1
8	-1	-1	-1	-1	-1	-1	-1	1	1	1
9	-1	-1	-1	-1	-1	-1	-1	1	1	1
10	1	1	1	1	1	1	1	1	1	1
Sum	2	2	2	-6	-6	-6	-6	2	2	2
Sign	1	1	1	-1	-1	-1	-1	1	1	1
True Class	1	1	1	-1	-1	-1	-1	1	1	1

Figure 5.36. Example of combining classifiers constructed using the bagging approach.

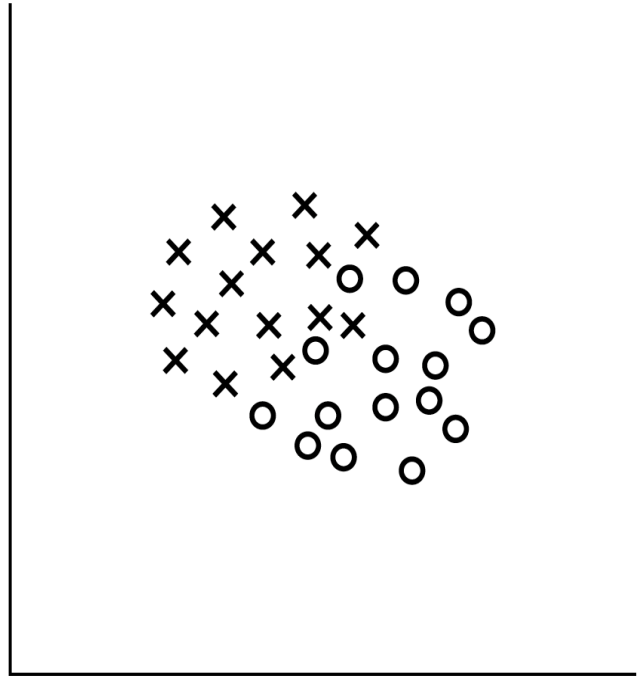
Bagging with weighted Aggregation

Weighted voting (Soft voting): Every classifier *i* output a class based on the learned model $h_i(\mathbf{x})$ for each instance **x**, and these predictions are multiplied by each classifier's weight (relevance), and finally averaged by the number of classifiers. The final class label is then derived from the class label with the highest average probability.

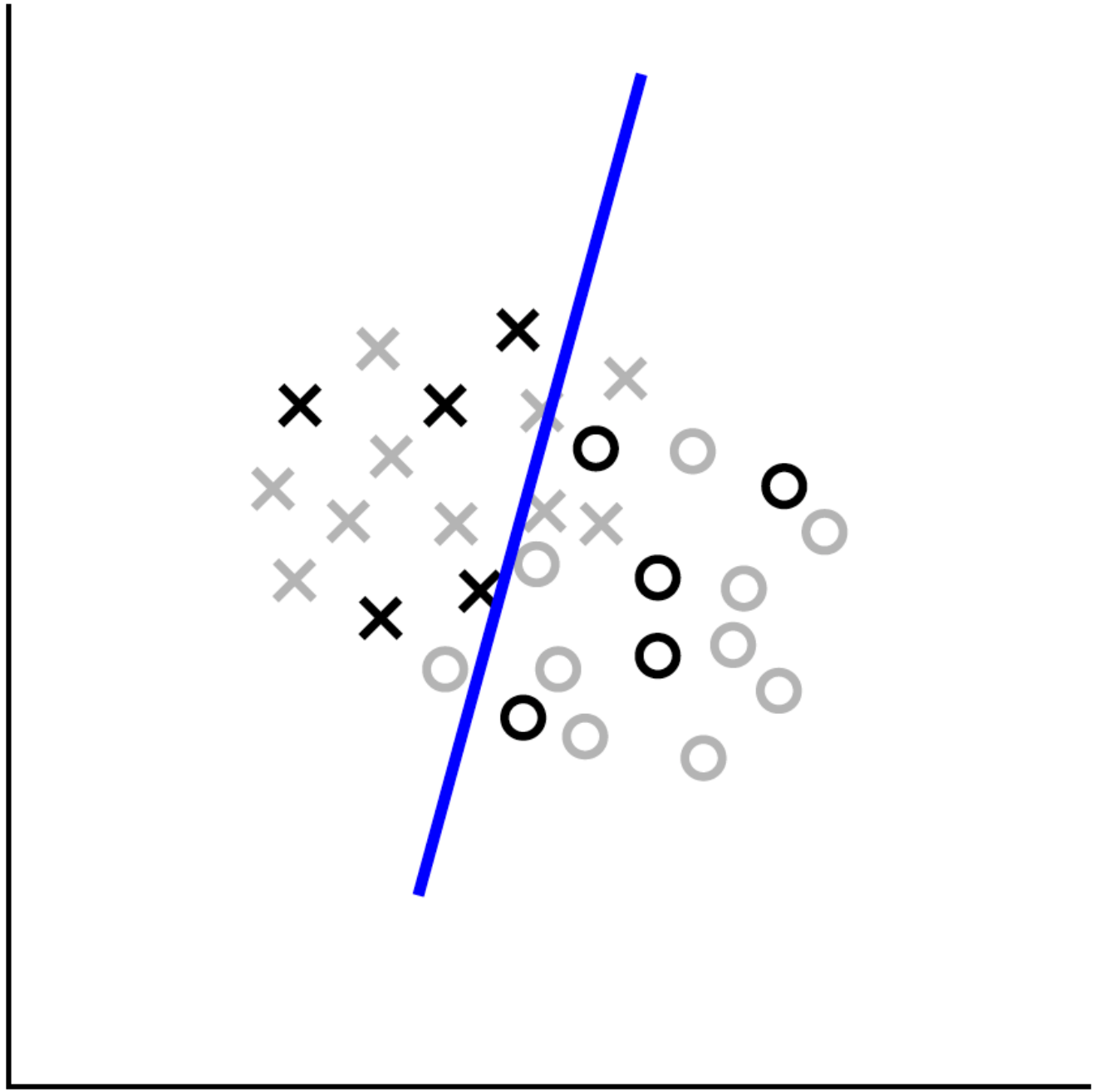
Tip: In reality, weights are hard to find using intuition. To counter this subjective process, a **linear optimization equation** or a **neural network** could be used to learn the optimal weighting for each of the models to optimize the accuracy of the ensemble.

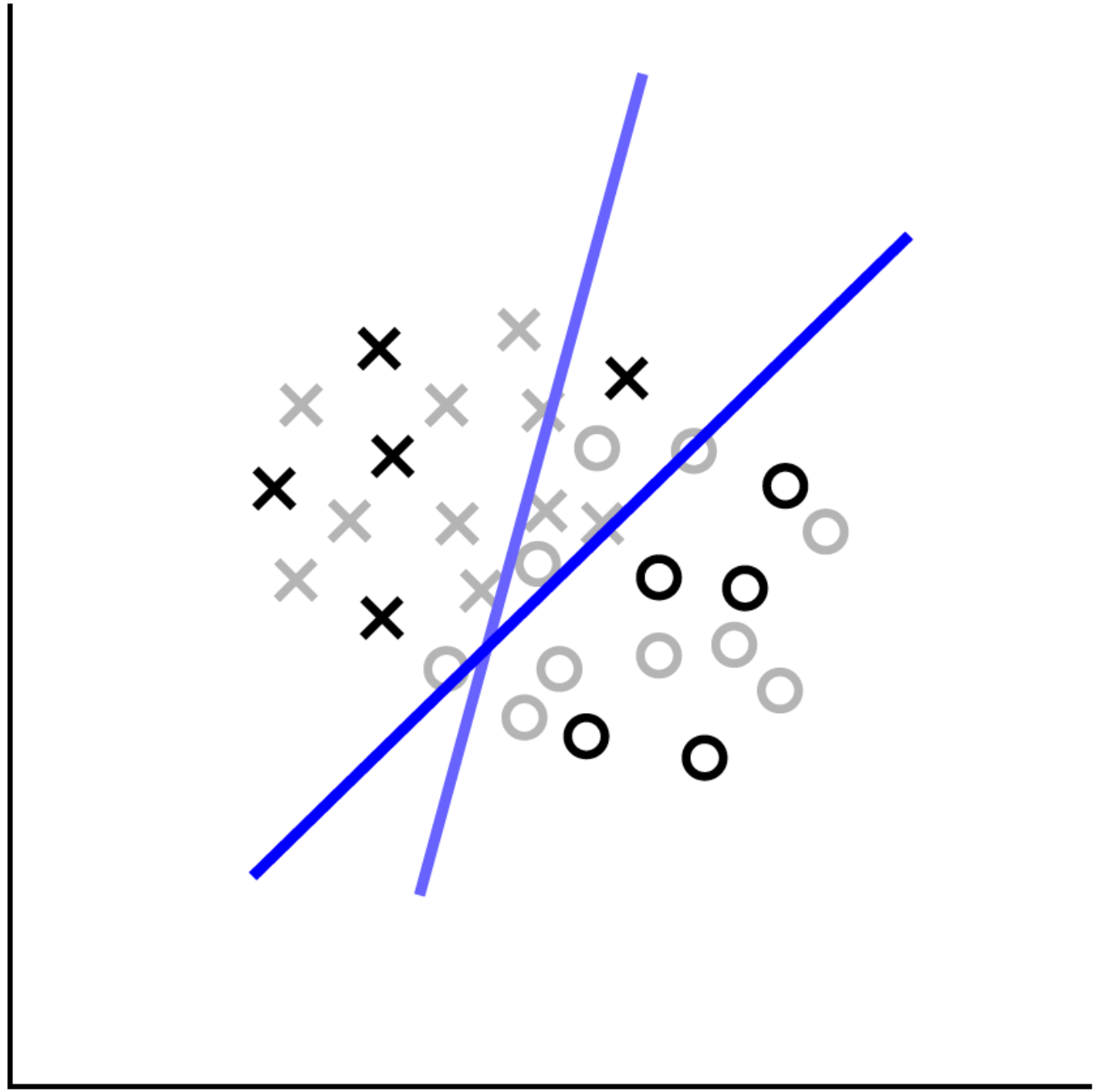
Bagging with aggregation by averaging different learned functions

Example: creating a non-linear classifier out of many linear classifiers (e.g perceptrons)



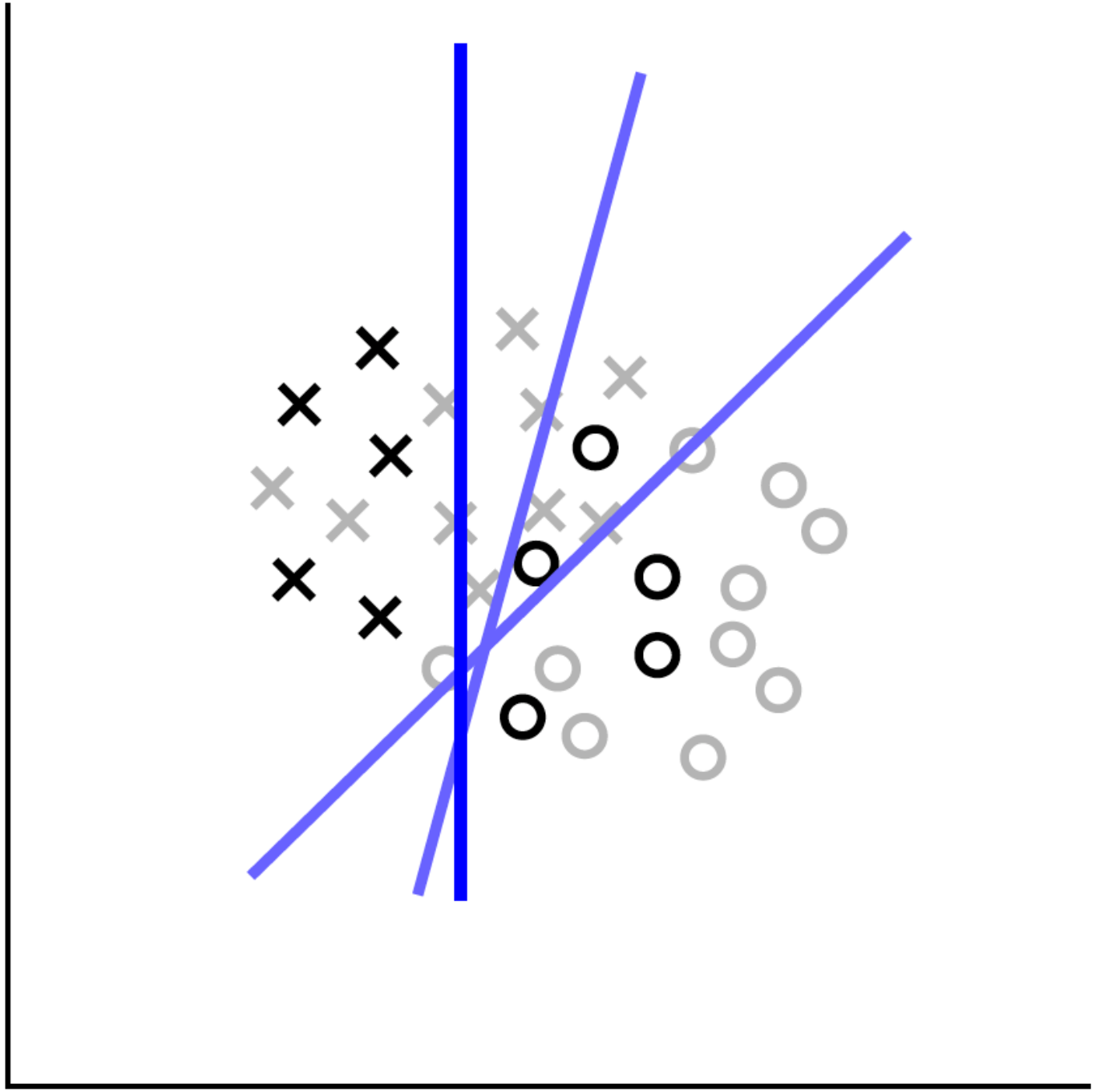
2

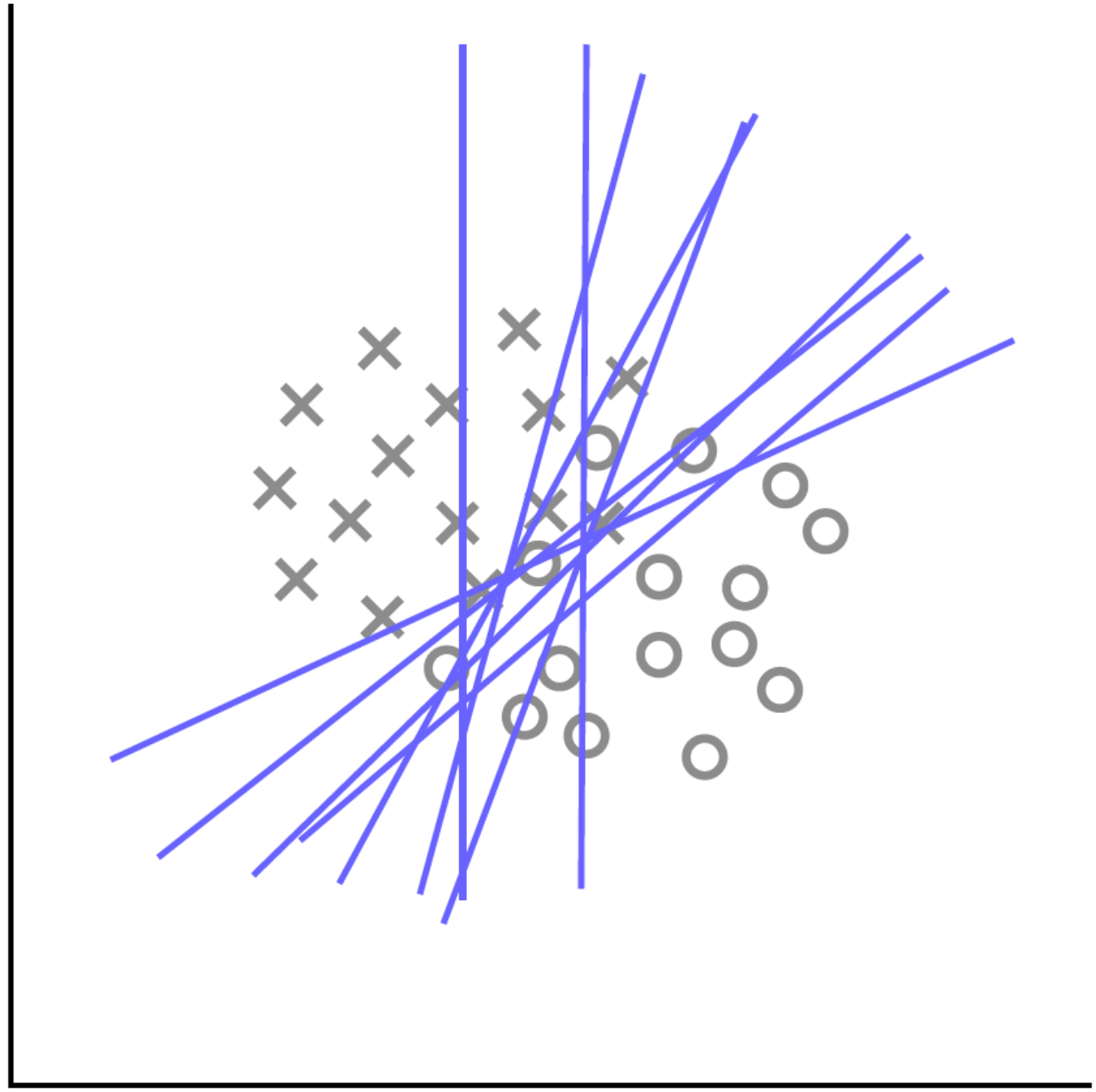




3

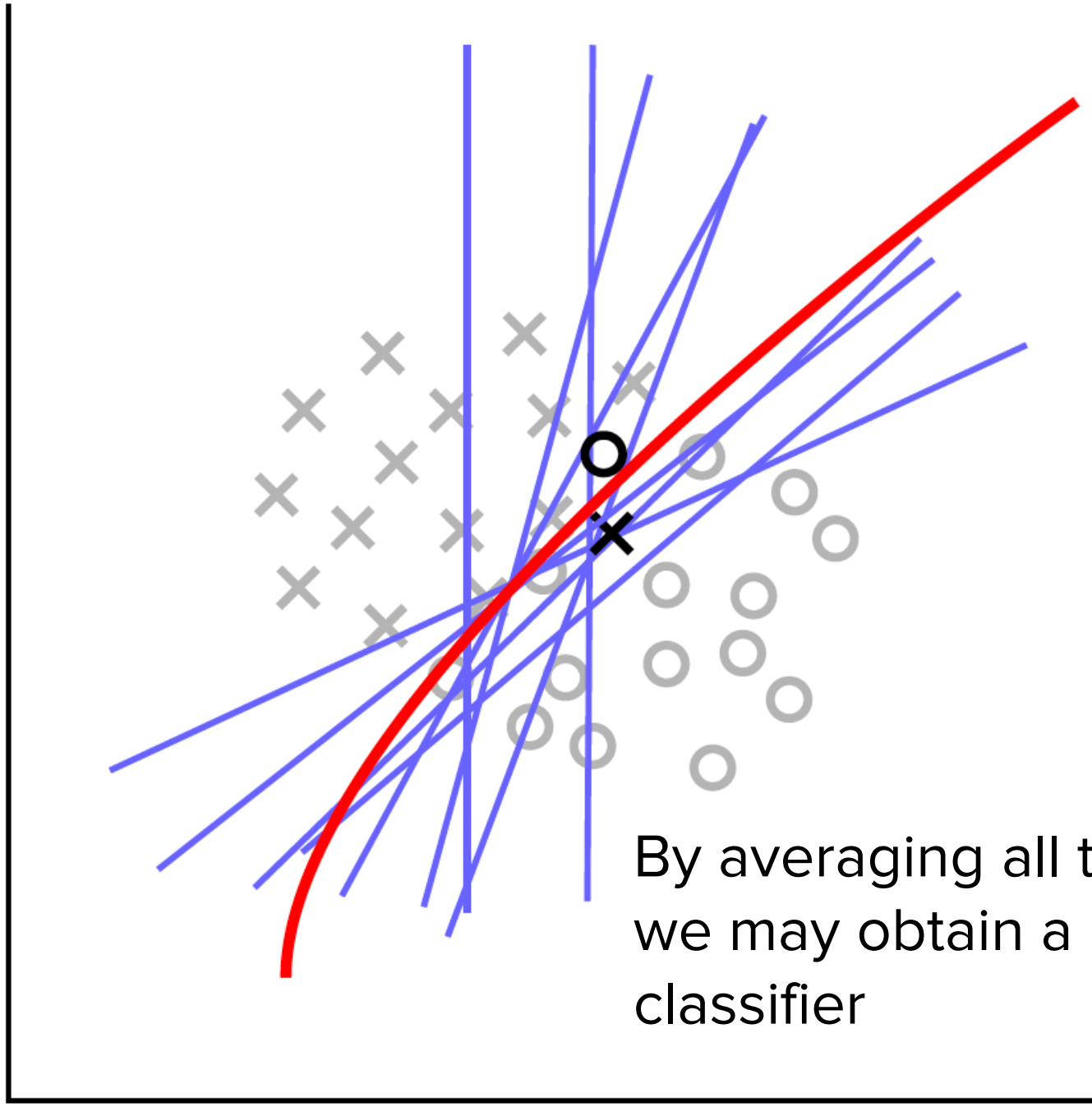
4





5

6



By averaging all the lines we may obtain a perfect classifier

Bagging:
Notable Benefits

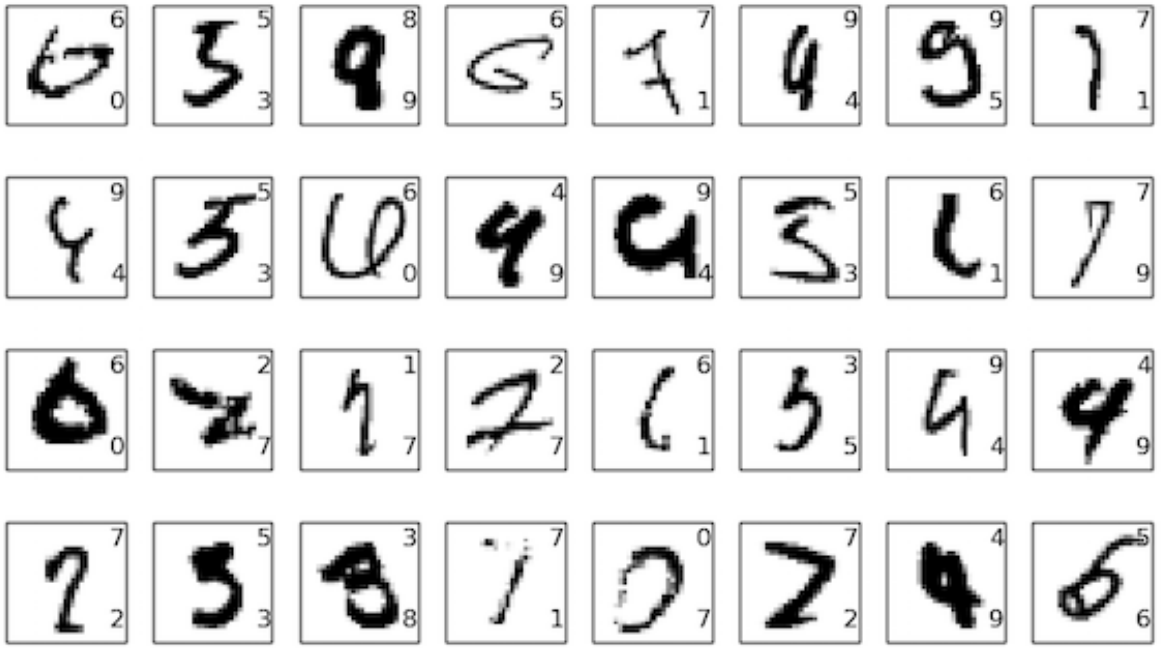
- Robust to overfitting
- Main goal is to reduce **variance** of combined models
- Improves ability to ignore irrelevant features
- Works well if all instances **have an equal probability p of being classified correctly or wrongly** (means: *Pr(f(x)≠h(x))=p* **for all x in X**) (this means that, more or less, all instances have the same complexity)

Bagging:
Issues

- Does not focus on any particular instance of the training data - **assumption is that all instances have the same probability of misclassification** (often wrong assumption, e.g. image recognition)

- What if we want to focus on a particular instance of training data?
  - ➢ E.g. some instance can be more difficult to classify than others (**and on these difficult instances most "simple" classifiers may be all wrong, so majority voting won't work**)

Example 1: Handwriting Recognition



Some signs are more difficult than others, even for humans

Example 2:
Face Recognition

Not all faces are easily recognized here..

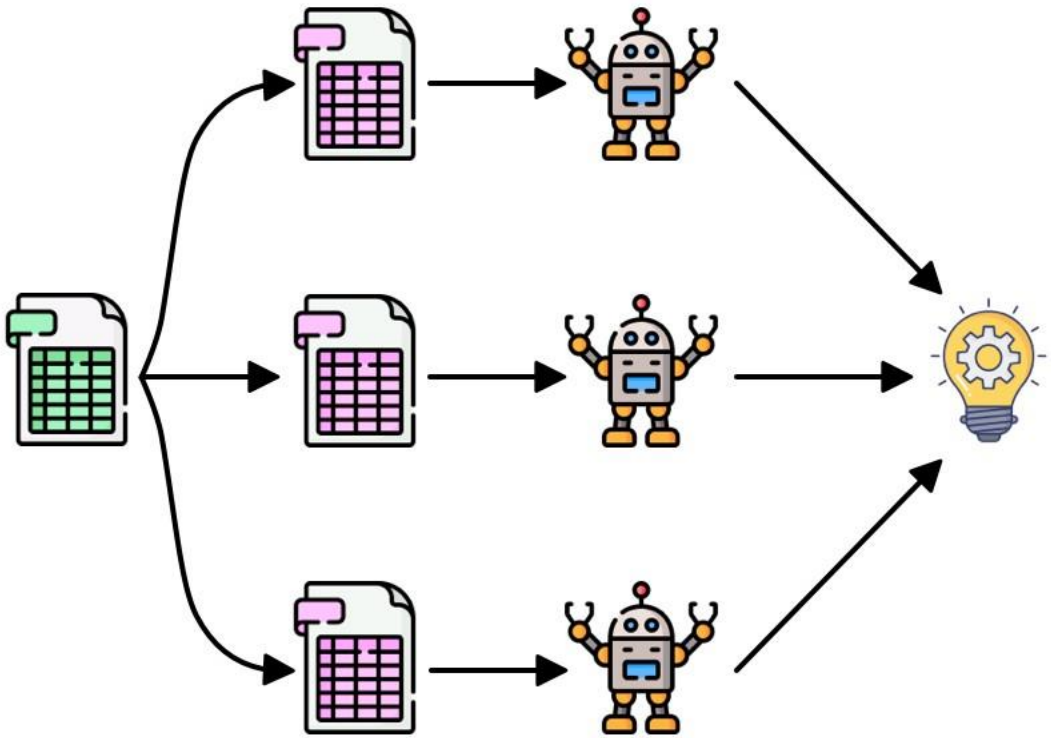
Boosting

Idea: The main idea of boosting is to add models to the overall ensemble **sequentially**. At each iteration, a new model is created and the new base-learner model is forced to «pay attention» to the errors of the previous learners.

As for bagging, the algorithm creates multiple weak models whose output is finally combined to get an overall prediction.

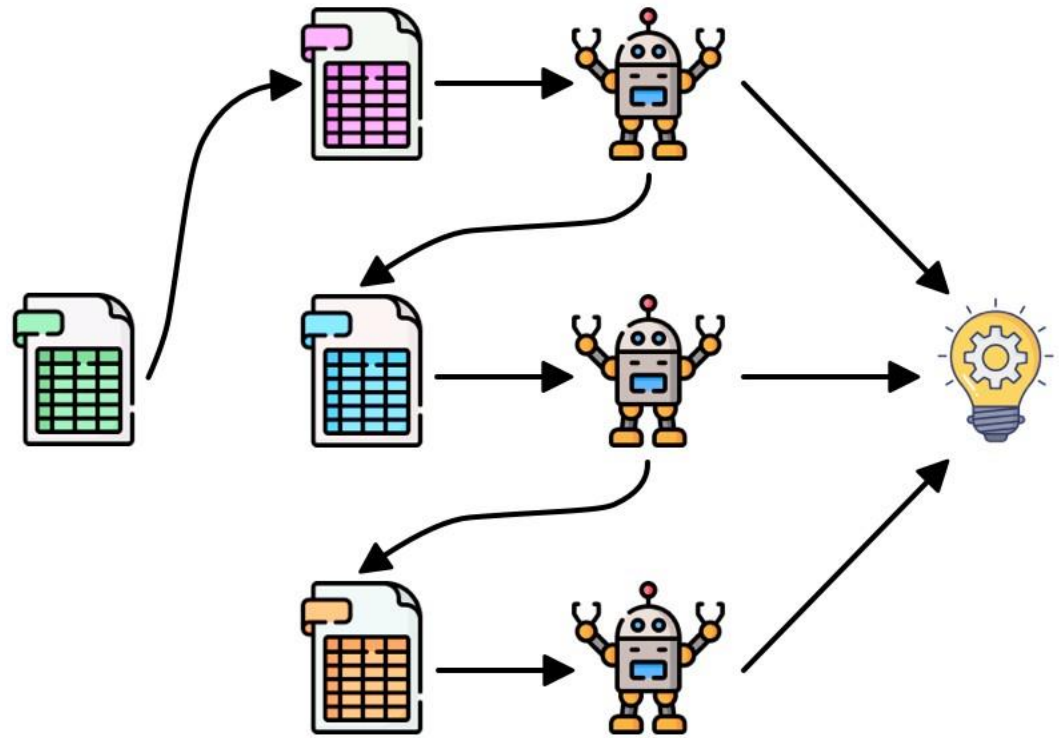
- The main goal of boosting is to reduce the **bias** (errors due to the model itself, not to its sensitivity to data variations)

Bagging



Parallel

Boosting



Sequential

Boosting

How: An iterative procedure to adaptively change the distribution of training data by **focusing (in each iteration) on previously misclassified examples.**

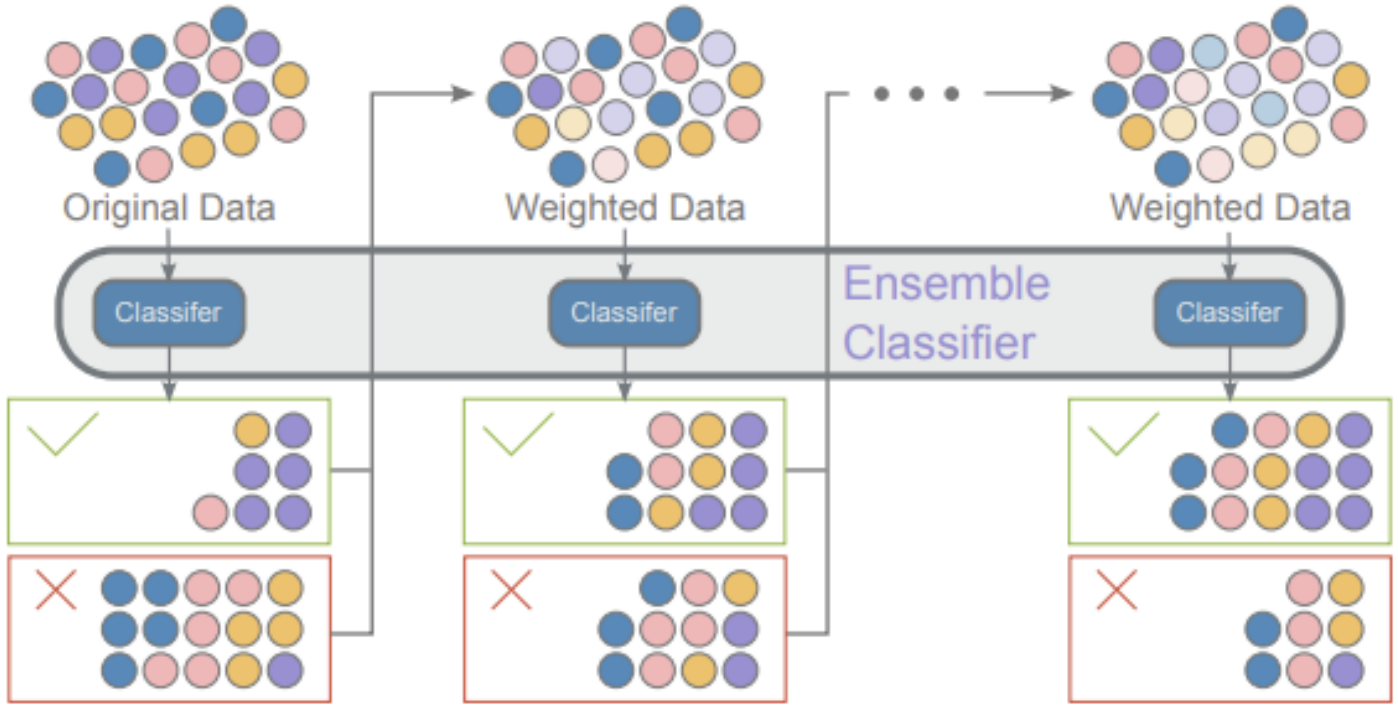
1. Get a dataset
2. Take a bootstrap (as for bagging), and train a model on it
3. See on which examples the model is wrong
4. **Upweight** those ‘hard ‘ examples, **downweight** the ‘easy’ ones,
5. Go back to step 2, but with a **weighted bootstrap,** so that in next iteration harder examples have a higher probability of being extracted

Each new member of the ensemble «focuses» on the instances that the previous ones got wrong!

Boosting

- Instances x_i in the training set D are sampled with a probability that depends on their weight w_i^j ($P(X=x_i)=w_i$ in iteration j). Initially, instances are equally weighted ($P(X=x_i)=1/n$) .
- In iteration (round) j, instances that are wrongly classified by current learner **will have their weights increased** (so that their probability of being sampled in next boosting round will grow)
- Those that are classified correctly will have their weights **decreased**

The workflow of boosting algorithms

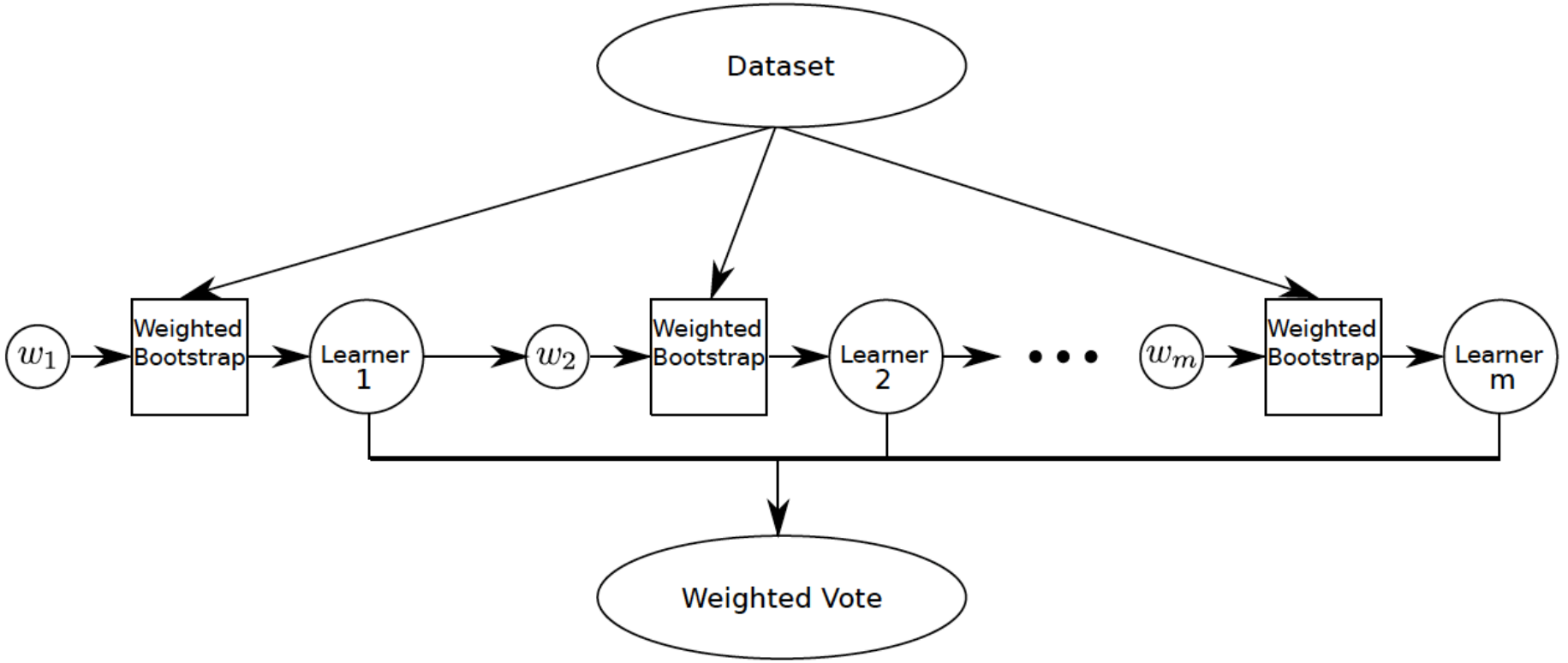


Boosting Example

- Suppose example 4 is hard to classify (round 1 classifier is wrong on example 4)
- Its weight is increased, therefore it is more likely to be extracted again in subsequent weighted sampling rounds (2)
- Therefore, in the subsequent round, the learner is forced to pay more attention to the misclassified example (in the attempt of reducing the error on the learning set)
- If round 2 classifier is again wrong on example 4, it probability of being extracted increases again

Original Data	1	2	3	4	5	6	7	8	9	10
Boosting (Round 1)	7	3	2	8	7	9	4	10	6	3
Boosting (Round 2)	5	4	9	4	2	5	1	7	4	2
Boosting (Round 3)	4	4	8	10	4	5	4	6	3	4

Boosting flow diagram



Adaboost

Adaptive Boost

- Input:
  - ➢Training set D containing **n** instances
  - ➢A classification algorithm (e.g., NN or a Tree-based algorithm)
  - ➢T iterations (i.e. rounds) (i=1,…,T). A classifier $\mathbf{C_i}$ is learned at each round.
- Output:
  - ➢A composite classifier **C***

Adaboost:

Training Phase

- Training data D contains **n** labeled data
  - (x_1,y_1), (x_2,y_2), (x_3,y_3),....(x_n,y_n)
  - where y_i are the correct classifications
- Initially assign equal weight **1/n** to each example
- To generate *T* "base" classifiers, we need *T* rounds
- Round ***i***:
  - instances from D are sampled **with replacement**, to generate the dataset $\mathbf{D_i}$ ($|\mathbf{D_i}| = n$)
- Each instance's chance of being selected in the next rounds **depends on its weight**
  - the new sample is generated directly from the training data D with **different sampling probability according to the weights**;

AdaBoost:

Testing Phase

- Testing occurs on individual classifiers C_i at the end of each round.
- The performance of each classifier on training set is used to assess the “importance” or **authority** of C_i
- **Final testing is performed on unseen data (i.e. a validation set)**. To combine individual classifications by each C_i, the decision of each classifier is taken into consideration **proportionally to its importance**

AdaBoost:

Testing Phase

Training phase of $\mathbf{C_i}$ depends on previous testing phase on $\mathbf{C_{i-1}}$

- Base classifier $\mathbf{C_i}$, is learned from training data of set $\mathbf{D_i}$
- Error of $\mathbf{C_i}$ is tested **always using** $\mathbf{D_i}$ (test set = training set)
- Weights of training data are **adjusted** depending on how they were classified

AdaBoost:

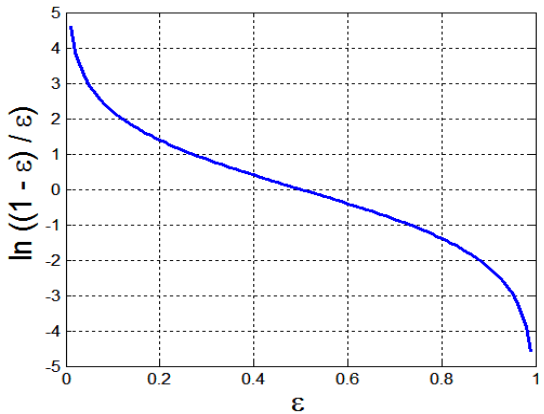
Testing Phases for individual classifier

- "Base" learned classifiers: C_1, C_2, …, C_T
- **Loss Function (Error rate of C_i on sample D_i)**:

$$error(Ci) = \varepsilon_i = \sum_{j=1}^{n} w_j \delta\left(C_i(x_j) \neq y_j\right)$$

  - *i* = index of boosting round
  - *j*=index of instance in training data
  - $\delta(c)\ is\ 1\ if\ the\ condition\ c\ is\ true$, else is zero
  - w_j = *current weight of xj (1/n in round 1)*
- **Importance of a classifier: the weight of a classifier C_i in final vote is**:

$$\alpha_i = \frac{1}{2}\ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$$



AdaBoost:

Weight updating rule (before (i+1)th round)

- Weight updating rule **on all training data:**

$$w_j^{(i+1)} = \frac{w_j^{(i)}}{Z_i} \times \begin{cases} \exp^{-\alpha_i} & \textbf{if } C_i(x_j) = y_j \\ \exp^{\alpha_i} & \textbf{if } C_i(x_j) \neq y_j \end{cases} \qquad \alpha_i = \frac{1}{2}\ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$$

where Z_i is *a* normalization factor

α_i is the "importance" of classifier C_i, as previously computed
Z_i is used to obtain that weights w_j are **sampling probabilities** for x_j

If classification of x_j is correct, decrease weight (divide by $\exp^{\alpha_i}$)
else increase (multiply by $\exp^{\alpha_i}$)

AdaBoost:
Final Testing Phase

- The lower a classifier's C_i error estimate ε_i is, the more accurate it is, and therefore, **the higher its importance α_i when final voting is performed**
- Final Testing (on unseen data, validation set):
  - For each class value y_j, sum the weights of each classifier that assigned class y_j to the instance x_{test}.
  - The class with the highest sum is the WINNER, the one the model will predict

$$y = C^*(x_{test}) = \underset{y}{\operatorname{argmax}} \sum_{i=1}^{T} \alpha_i \delta\left(C_i(x_{test}) = y\right)$$

- $\delta(x) = 1$ if $C(x)=y$. For any possible classification value y of x_{test}, the weighted sum of classifiers with output y is computed. The y value that maximised this sum is taken as the predicted vale.

Example

$$y = C^*(x_{test}) = \underset{y}{\operatorname{argmax}} \sum_{i=1}^{T} \alpha_i \delta\left(C_i(x_{test}) = y\right)$$

Example:

- 3 class values: A, B and C
- 6 classifiers C1..C6
- C1,C2 and C6 classify x_{test} =A ; C3 and C4 classify x_{test} =B; C5 classifies x_{test} =C
- weight of classifiers: C1, C2, C3 =0,1; C4=0,35; C5= 0,15 C6=0,2 (total is 1)
- $\delta(*)$ =1 iff * is true, else is 0

$C(x_{test})$=A➔ 0,1 +0,1+0,2=0,4

$C(x_{test})$=B➔ 0,1 +0,35=0,45

$C(x_{test})$=C➔ 0,15

$Argmax_{(A,B,C)}$ =A

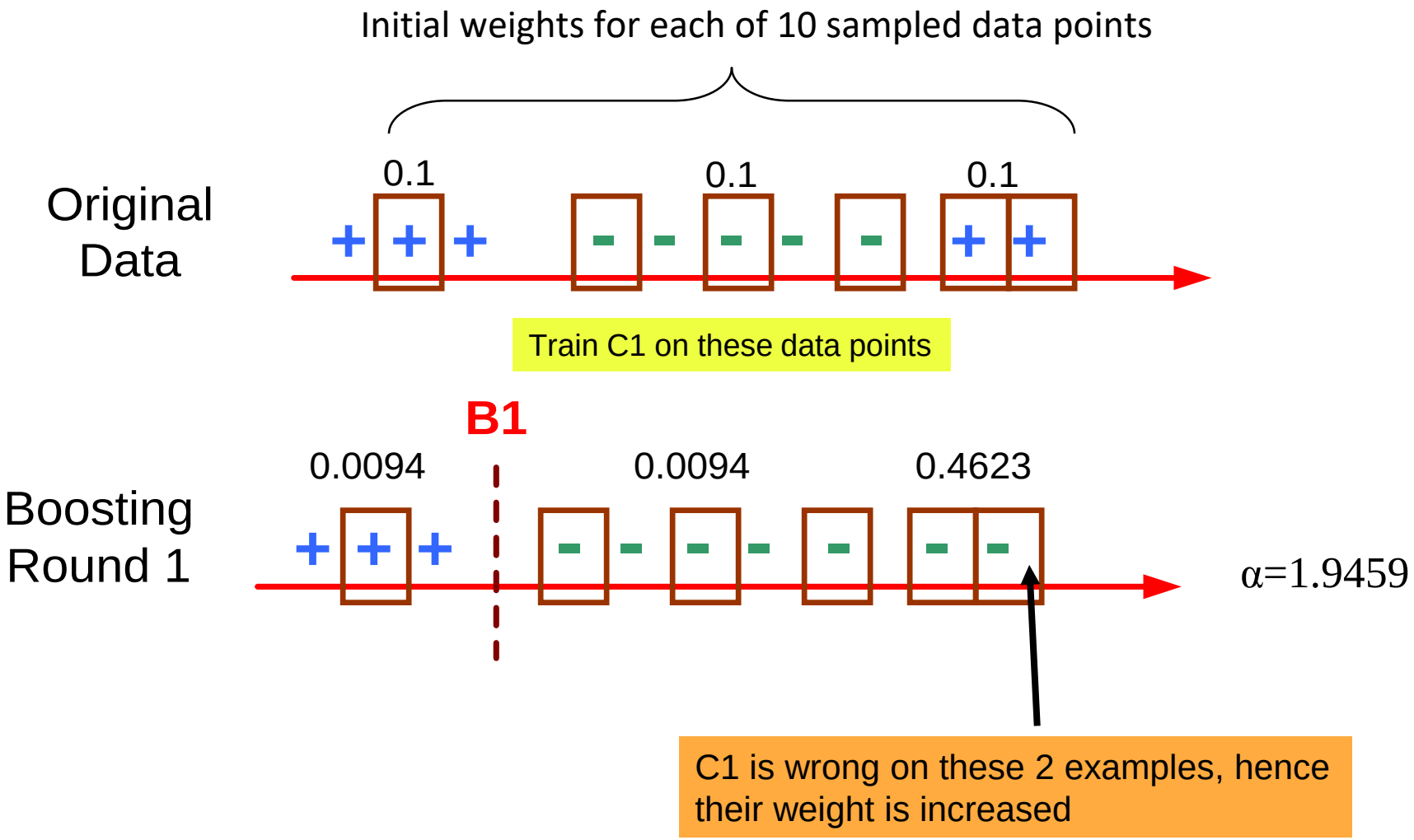
AdaBoost:
Pseudo-code

Given D:<**x**i,yi> |D|=n

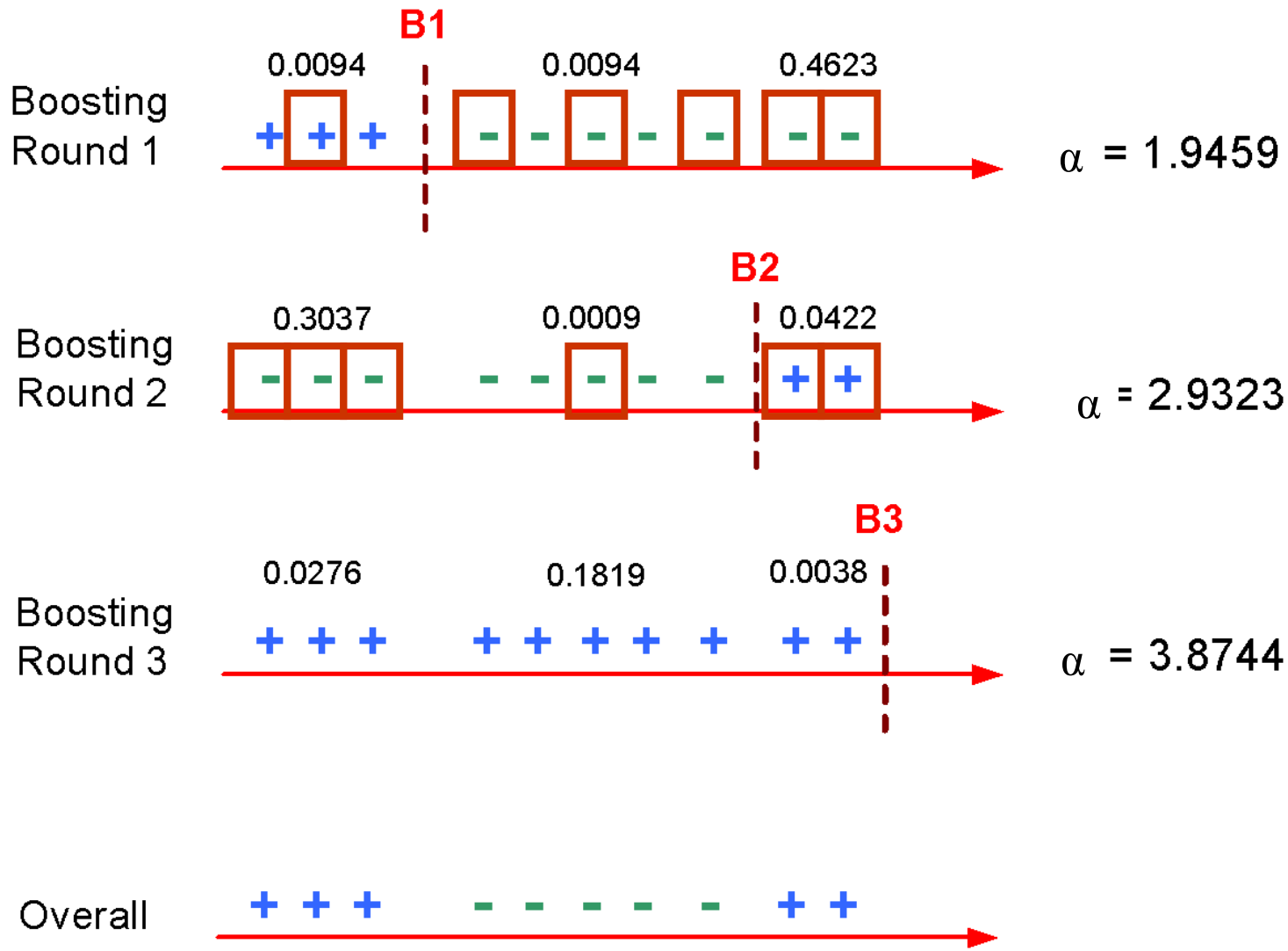
1. **Set** weights w_j=1/n
2. **For** i=1..T
   a. **Bootstrap** D_i from D using P(X=x_j)=w_j, and train C_i
   b. **Test** C_i and **compute error** rate on D_i, ε_i
3. **IF** ε_i>1/2 then T=t-1 **abort** loop
   a. **Compute** αi
   b. **Update** wj
4. Test: for any unseen x_{test}

$$C^*(x_{test}) = \underset{y}{\operatorname{argmax}} \sum_{i=1}^{T} \alpha_i \delta\left(C_i(x_{test}) = y\right)$$

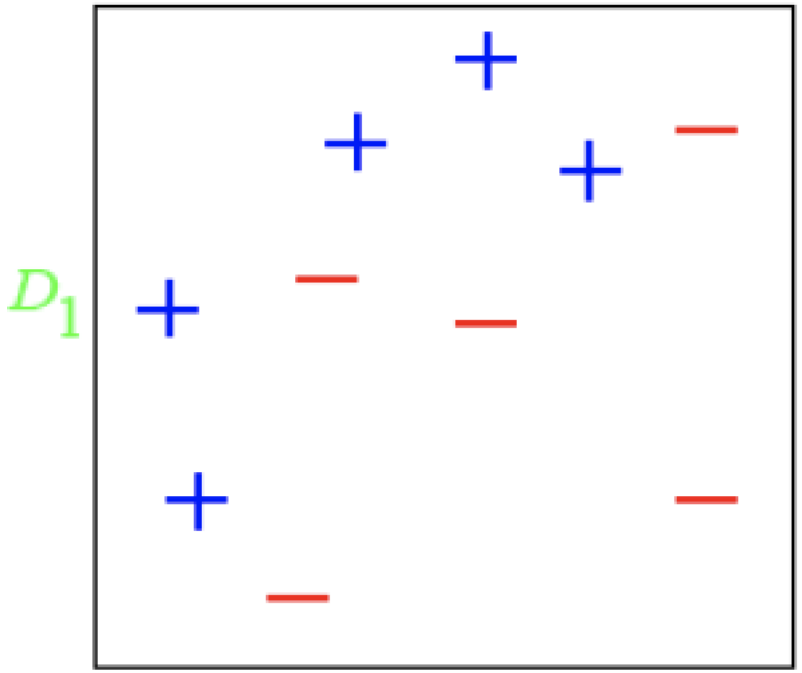
Illustrating AdaBoost



AdaBoost

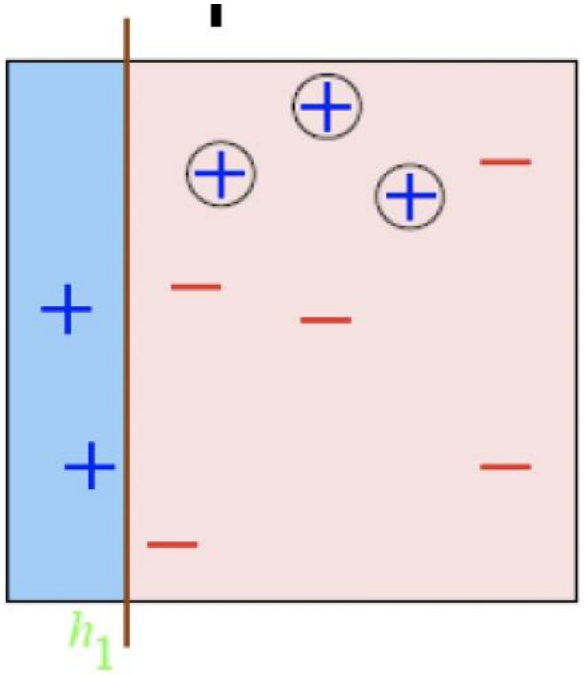

B1
0.0094
0.0094
0.4623
Boosting Round 1
α = 1.9459
B2
0.3037
0.0009
0.0422
Boosting Round 2
α = 2.9323
B3
0.0276
0.1819
0.0038
Boosting Round 3
α = 3.8744
Overall

AdaBoost (another example): Example Round 1



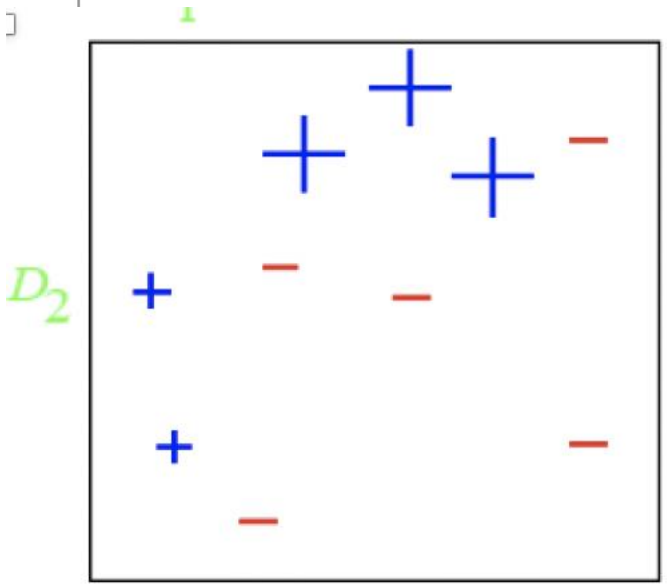
AdaBoost: Example Round 2

2



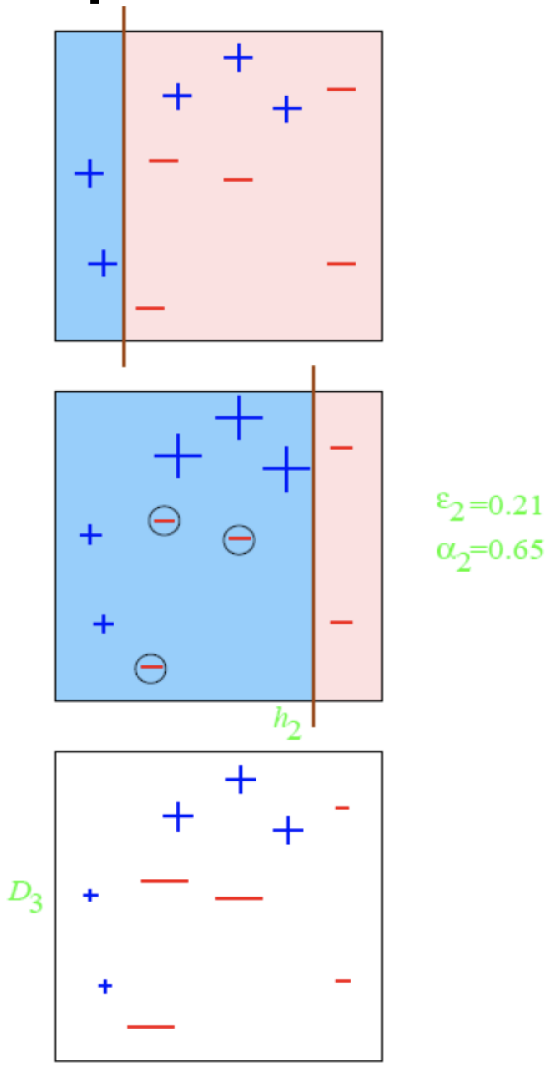
$\varepsilon_1 = 0.30$

$\alpha_1 = 0.42$



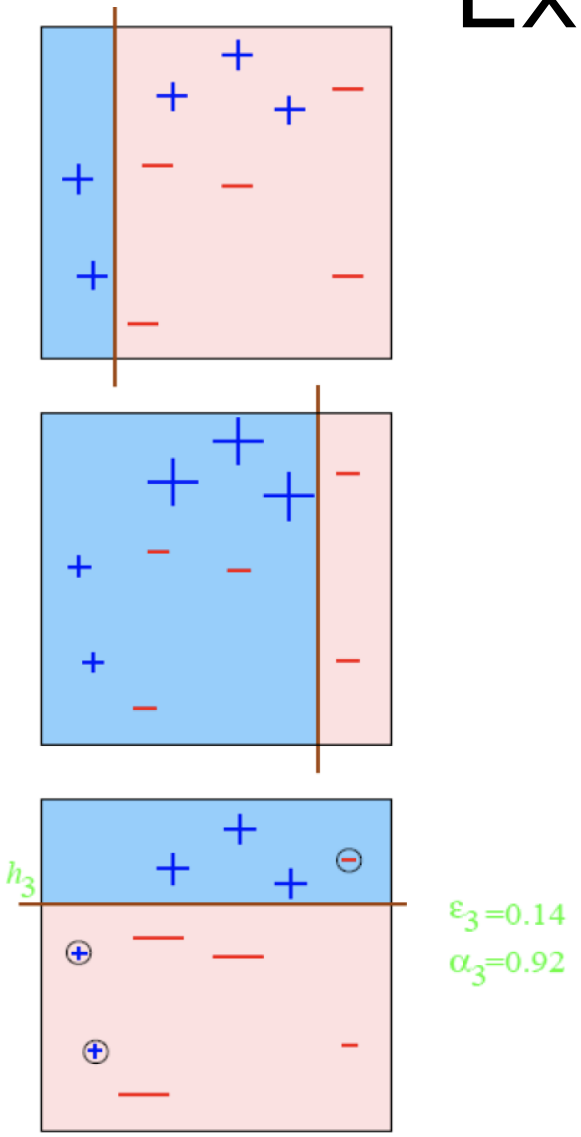
AdaBoost: Example Round 3

3



AdaBoost: Example

4



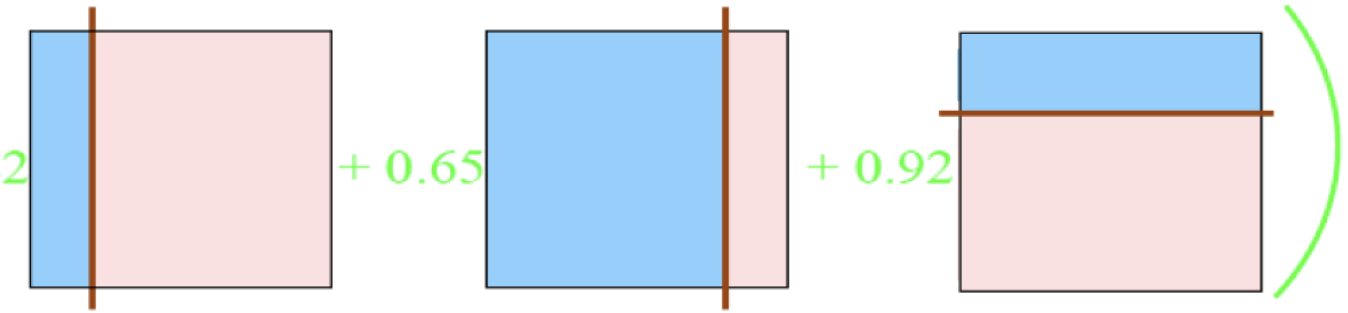
- The 3 learned classifiers

AdaBoost: The Final Hypothesis

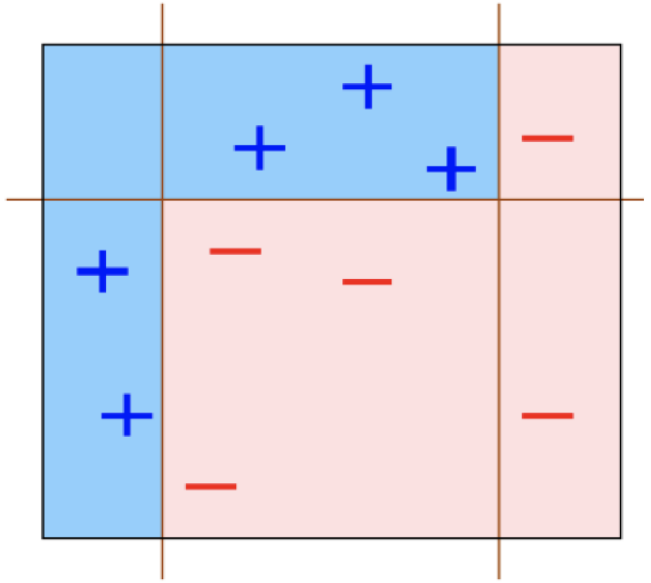
5

H_{final}

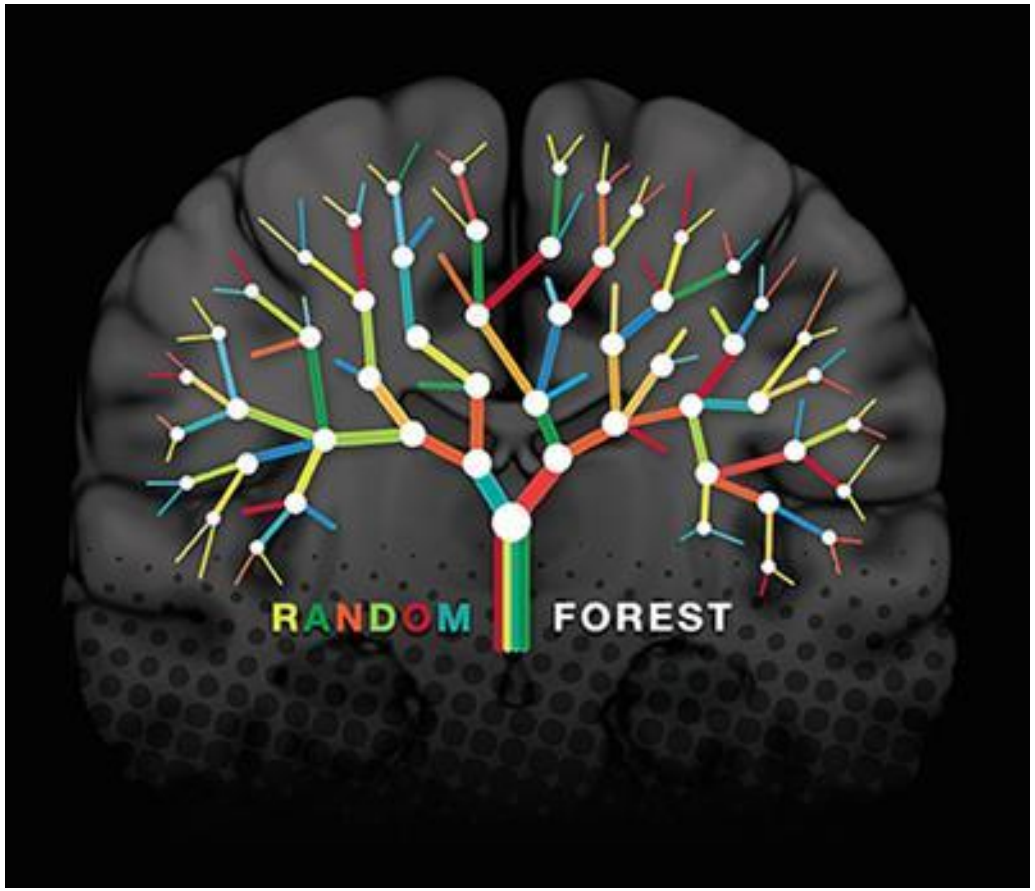
$= \text{sign}\left(0.42 \; + 0.65 \; + 0.92 \right)$



$=$



Random Forest



Random Forests

- Ensemble method designed for **decision/regression tree classifiers:**
- Combines predictions made by many **unpruned** d-trees.
- Each tree is generated based on a bootstrap sample of training data and a vector of **randomly chosen attributes** (i.e. random vector)
- The random vectors are generated from a fixed probability distribution.
- Final classification is chosen with a **voting method** like Majority Voting (Forest chooses the classification result having the majority of votes over all the trees in the forest)

Random Forests

Introduce **two sources of randomness**:

- **Bagging method**: each tree is grown using a bootstrap sample of training data (as in Bagging and Boosting)
- **Random vector method**: **At each decision node**, the best feature to test is chosen from a **random sample of *m*** attributes out of the total numer of features **d** rather than from all features
- So, if we have a dataset of dimension |D|xd=nxd, we randomly choose a subset D_k of D for training, and, at each split, a subset m of the d features

Random Forests:
Random Vector Method

The random vector method add the following rule to the D-tree (same for both regression and categorical trees) training algorithm.

For each node of the tree:

a. **Choose *m*** features **randomly (out of d)** on which to base the decision at that node.
b. Calculate **the best split** based on these ***m*** variables in the training set (e.g., test on best attribute in **m** based on Infogain).

Why random vectors? if one or a few features are very strong predictors for the output class (or value for regressors), these features will be selected **in many nodes of the trees**, causing them to become correlated. Instead, the idea is to generate as much as possible independent models! Accordingly, the set of features from which to find the best split are selected randomly at each iteration.

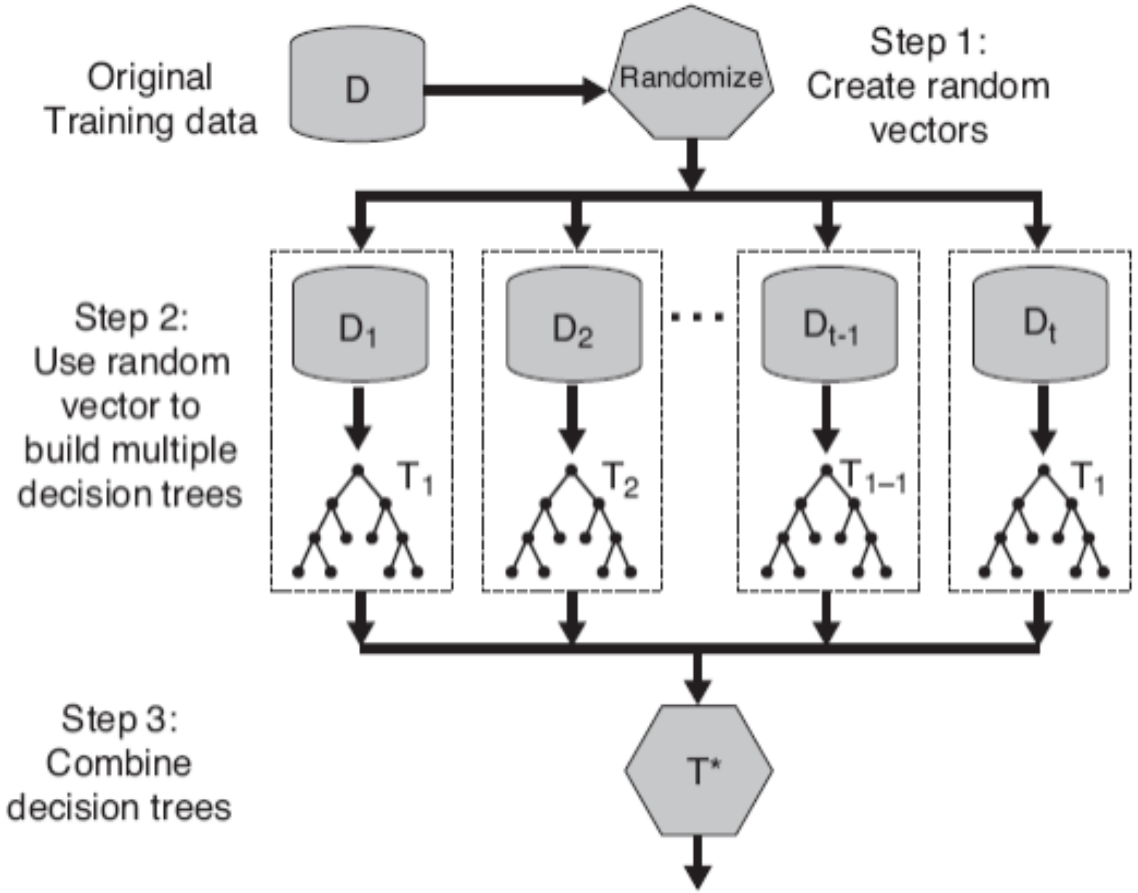
Random Forests:

Algorithm

1. **Divide** training examples into multiple training sets D_k using Bootstrap (i.e. choosing p times **with replacement** from all ***n*** available training cases)
2. **For each** training set D_k :
    a. **Train a decision tree** with the **random vector method**
    b. **Estimate the error** of the decision tree using the rest of the examples.
3. **Aggregate** the predictions of each tree to make classification decisions using a voting methods (usually Majority Voting)

Random Forests:

Visual Explanation



Why use Random Vectors?

- Because in an ensemble, we want **independent** base learner
- Averaging over trees with different training samples reduces the dependence of the predictions on a particular training sample (variance).
- Using random vectors of features further reduce this dependency. Typically, for a classification problem with d features, m=$\sqrt{d}$ (rounded down) features are used in each split.
- **Increasing the number of trees does not increase the risk of overfitting the data** .
- In fact the main advantage of RF is that they are very robust to overfitting

Advantage of Random Forest

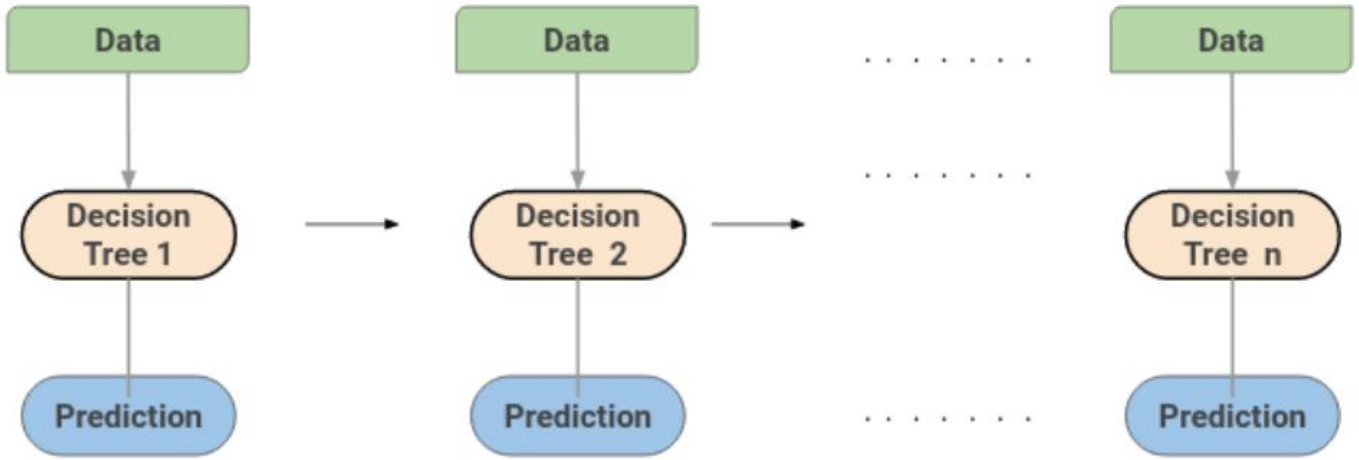
- Since each tree only handles a subset of features, this can be considered a good choice when instances are described by very many features
- It is also considered a good “dimensionality reduction” method (since it concentrates on subsets of features)

Other Popular (and more recent) Ensambles

GRADIENT BOOSTING MACHINES

A hybrid between Adaboost and Random forest

- Every Decision Tree DT_i (either categorical or regression tree) tries to improve over the previous model by “paying more attention” on instances where the previous model fails



How do we «pay more attention» to errors?

- Every DT_i (either categorical or regression tree) tries to improve over the previous model by paying more attention on instances where the previous model fails
- Adaboost does this by increasing the weight of misclassified instances
- GBM applies to regression trees, and does so using **gradient descent**
- How?

Gradient boosting (1)

- Let's say we start with a very simple model $h_0(x)$
- We have set of training instances D: $(x_1,y_1)..(x_n,y_n)$ and for any x_i we define the *residual* as follows:
- $r_j^0 = y_j - h_0(x_j)$ (=the error of h_0 on instance x_j)
- We want to improve $h_0(x)$ (reduce its residuals) by adding a <u>new model $h_1(x)$</u> such that:

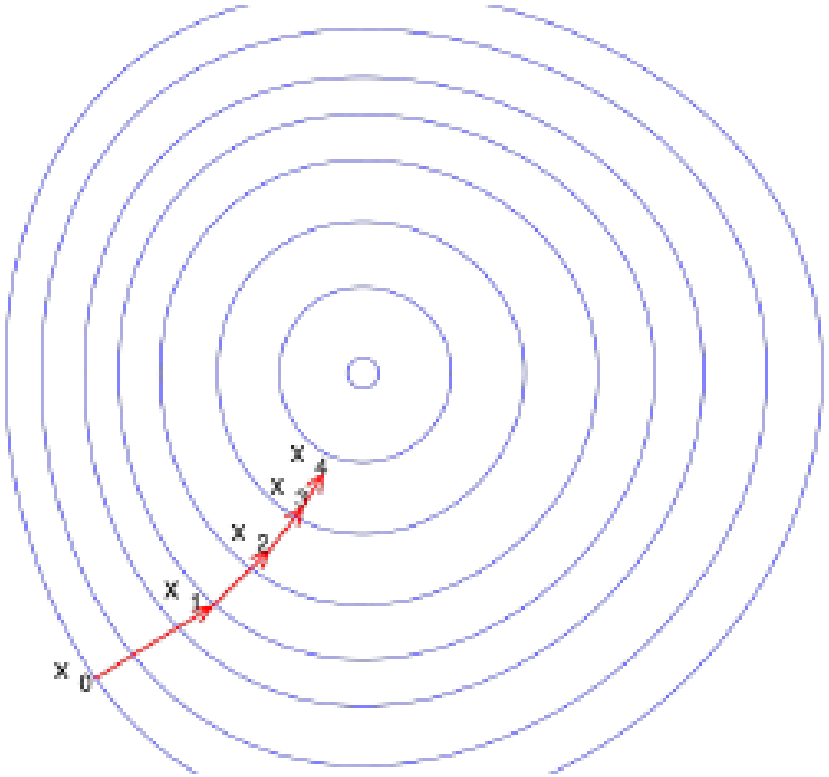
  $h_0(x)+h_1(x)=y$ for any (x,y) in D
- Or equivalently:

  $h_1(x)=y-h_0(x)$

Two questions:

1. How does this relate with gradient descent????
2. What if $h_1(x)$ does not achieve its task perfectly?

How do residuals relate with gradient?



- Let's consider regressors. As we have seen, a common loss function is the Square Loss function (note: MSE is the mean of square losses), defined as:

$$L=\sum L(x_i, h(x_i))= \sum(y_i - \mathrm{h}(x_i))^2/2 = \sum(y_i - \hat{y}_i)^2/2$$

(also called squared sum of errors)

- **To fit a model using this loss**, we must **optimize** h(x) by changing its parameters in a way that h(x) "moves" in the opposite direction of the Loss gradient:

$$h(x_i) = h(x_i) - \frac{\partial L}{\partial h(x_i)}$$

But $\frac{\partial L}{\partial h(x_i)}=\frac{\partial(\sum(y_i-\mathrm{h}(x_i))^2/2)}{\partial h(x_i)} = y_i\text{-}h(x_i) = y_i\ \text{-}\hat{y}_i = -r_i=g(x_i)$

The gradient is the opposite of the residual of h(x) on x_i !

Residuals can be interpreted as gradients!

- Back to GBM, let $h_0(x)$ be an initial weak model (a regression tree) in the pipeline
- The residuals r_j^0 of $h_0(x)$ on the dataset D are: $(y_1 - h_0(x_1))..(y_n - h_0(x_n))$
- Our aim is to «augment» $h_0(x)$ with $h_1(x)$ such that $h_1(x)$ is a new model «fitted» on the residuals of the previous model.
- The role of $h_1(x)$ is to «compensate» the shortcomings of the previous model (to generate an output that, added to previous function $h_0(x)$, cancel or at least reduces the difference between the ground truth output and the generated output
- If the new model $(h_0(x) + h_1(x))$ still performs poorly (=has high residuals), we can add another model $h_2(x)$ fitted on the residuals of $h_1(x)$ and **iterate**

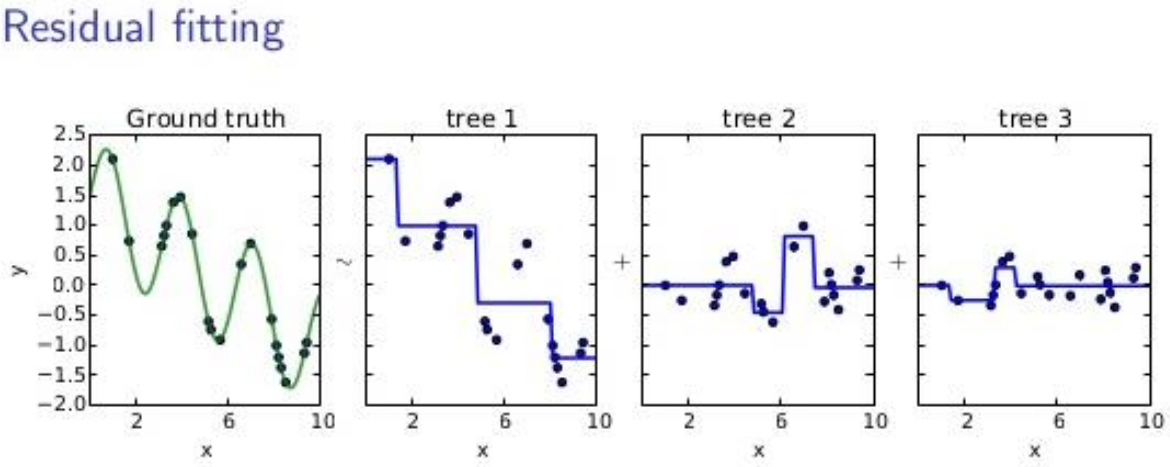
How do we «fit a new model» $h_{i+1}(x)$ on the residuals of the previous model $h_i(x)$?

- Just grow a model $h_{i+1}(x)$ (e.g., a tree) with the following training set:
- D:$\{[x_j, y_j - h_i(x_j)]\} = \{[x_j, r_j^i]\}$

- We train with current residuals (those of previous model) rather than with the «ground truth» output!
- The current model learns to output a quantity $\hat{r}$ which «compensates» the residual of the previous model
- Let's say that model «0» predicts h0(x)=1 for some x in D, but ground truth is 1.5; we would like model «1» to output a quantity that, if summed with the prediction of model «0», produces the exact value 1.5 (or at least «closer» to the ground truth value 1.5)!!

The intuition behing GBM

- What does it means «fit a regression tree with negative gradients»?
- Remember: error= variance+bias^2+noise (slide 5)
- A basic assumption of regression is that sum of its residuals should ideally be 0, i.e. the residuals should be spread randomly (which means that the residual error is only "noise", the random error ε that cannot be eliminated)
- ***If, instead, we are able to see some "pattern of residuals" around 0, we can leverage those pattern to fit a model****. (pattern of residuals mean that errors are somehow correlated, they are not random)*
- ***GBM therefore reduce both variance and bias!***



Each tree is the «sum» of previous trees with the current one

Gradient Boosting with Regression

Algorithm (link):

Start with an initial weak learner, say: $h_0(x) = \frac{\sum y_j}{n}$

$(output\ is\ the\ average\ value\ of\ observed\ values\ in\ the\ learning\ set)$

Iterate until convergence:

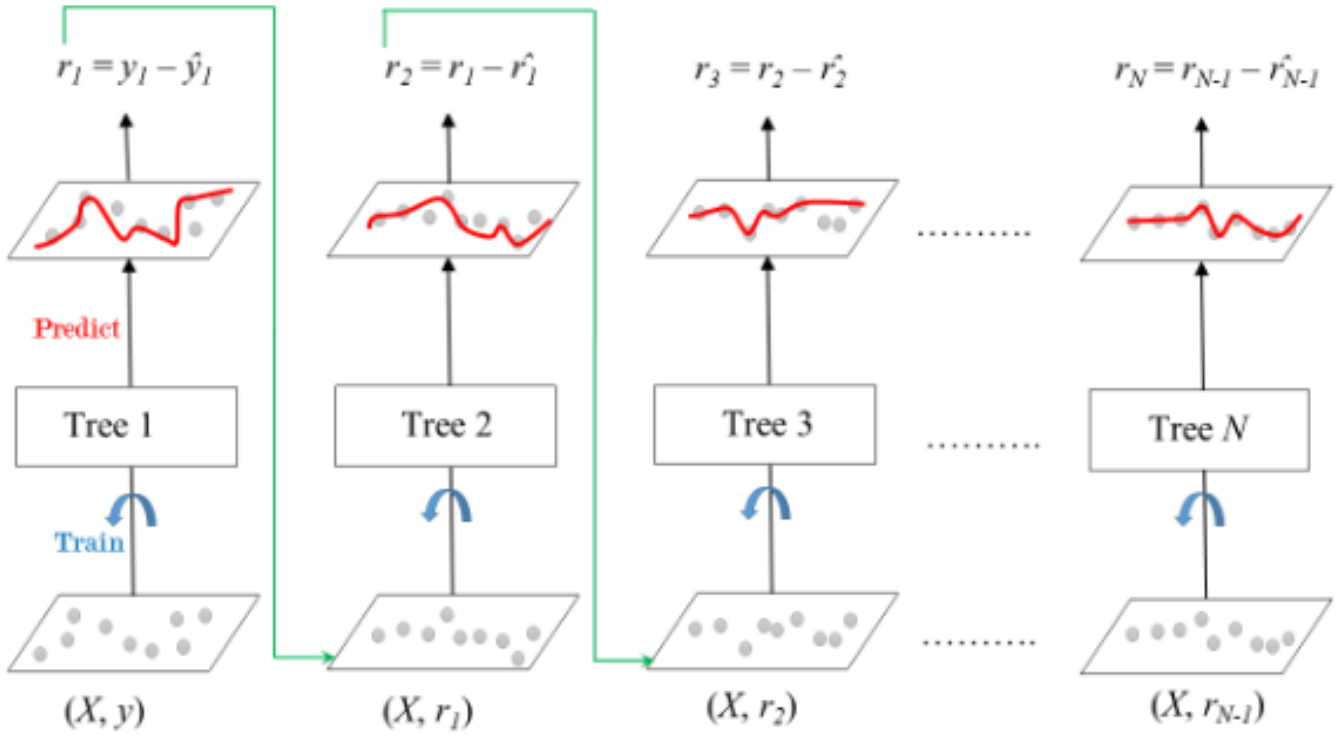
1. Calculate negative gradients: - g_j^i (residuals) of current model $h_i(x)$
2. Fit a regression tree $h_{r^i}(x)$ to negative gradients - g_j^i of $h_i(x)$ (i.e., train on pairs (x_j, r_j^i))
3. $\left(h_{i+1}(x) = h_i(x) + h_{r^i}(x)\right)$

Note that the role of $h_{r^i}(x)$ is to COMPENSATE the shortcomings of $h_i(x)$.

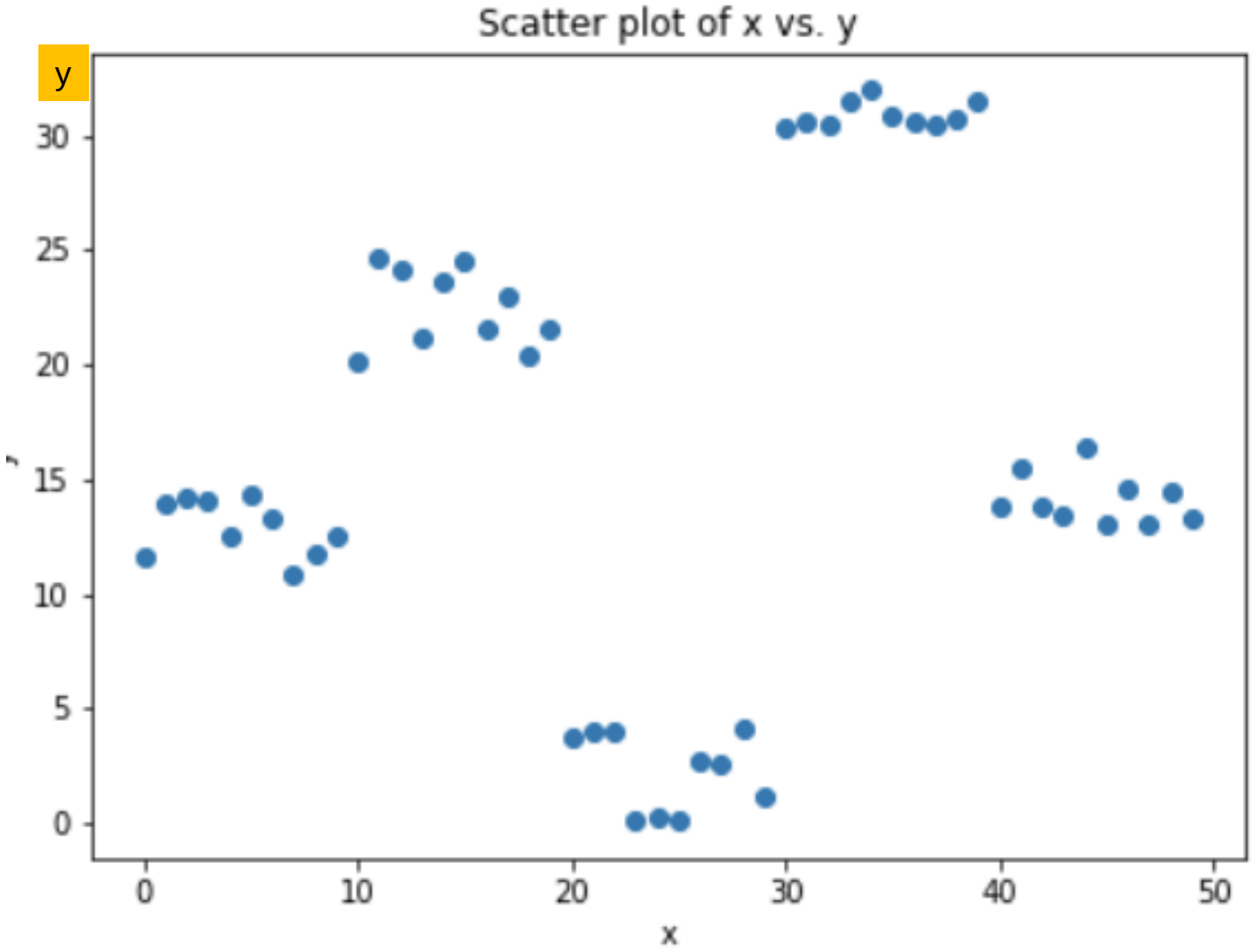
Final prediction is: $\left(\hat{y} = \sum h_i(x) = y^0 + r_x^1 + r_x^2 + \cdots r_x^k\right)$ since all subsequent predictors $h_{r^i}(x)$ after h_0 are trained to predict residuals of their previous model.

Residuals can be weighted in some implementation of the «basic» algorithm $\left(h_{i+1}(x) = h_i(x) + \gamma h_{r^i}(x)\right)$.

Flowchart of GB

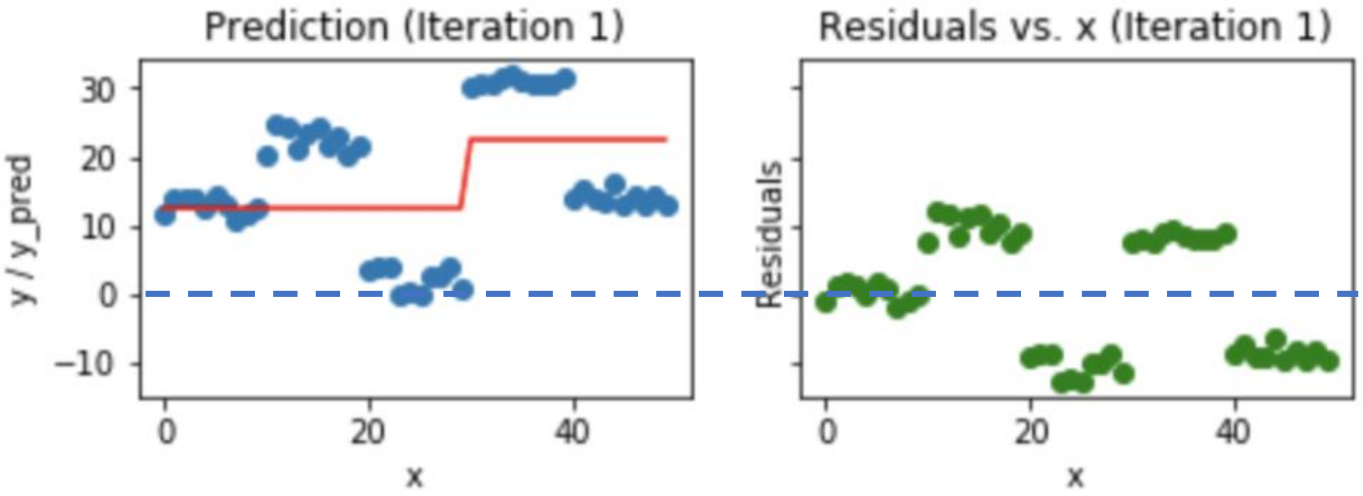


Example (1)



Example: first iteration

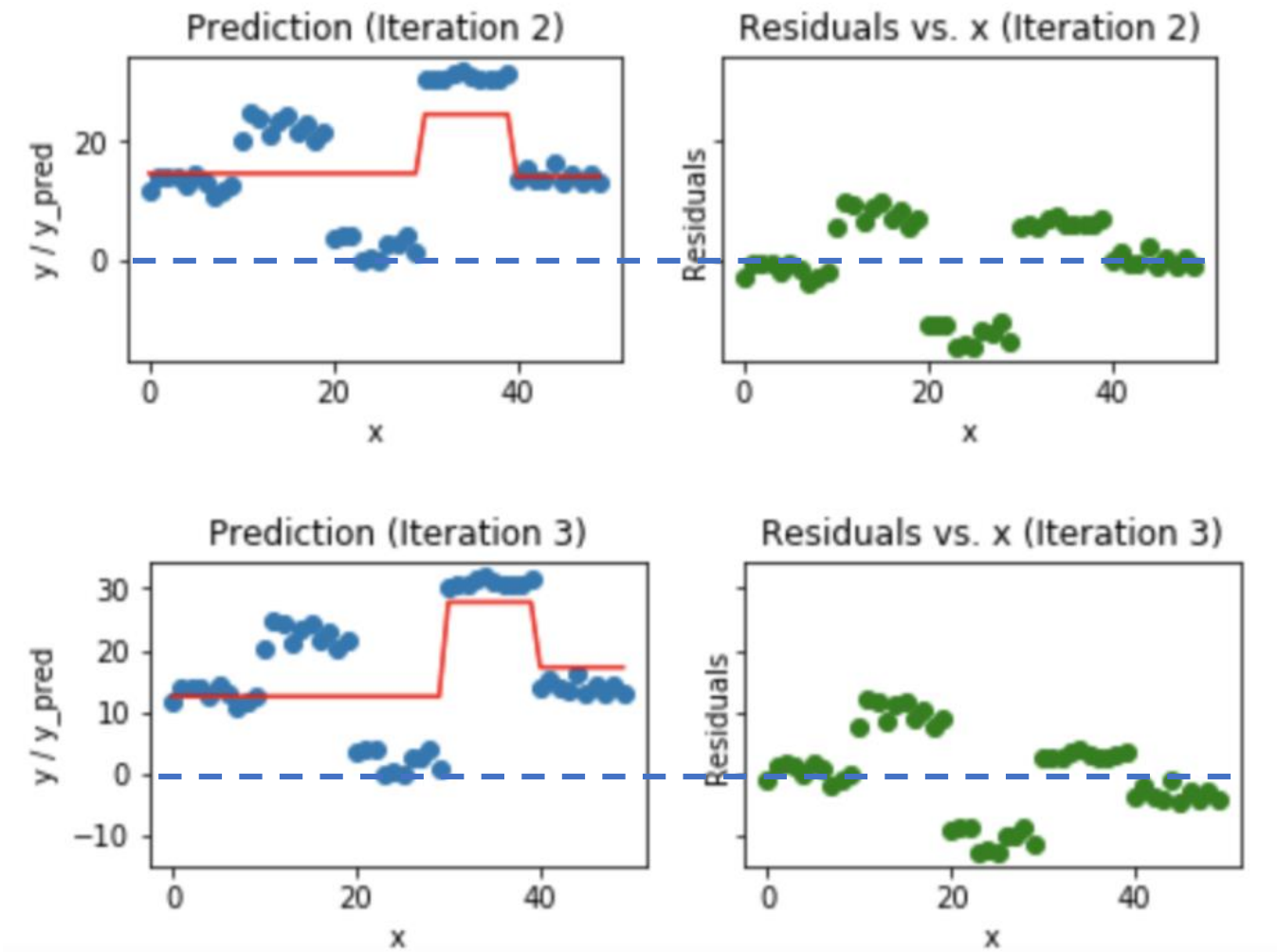
Red line is a simple predictor, green dots are the residuals



https://blog.mlreview.com/gradient-boosting-from-scratch-1e317ae4587d

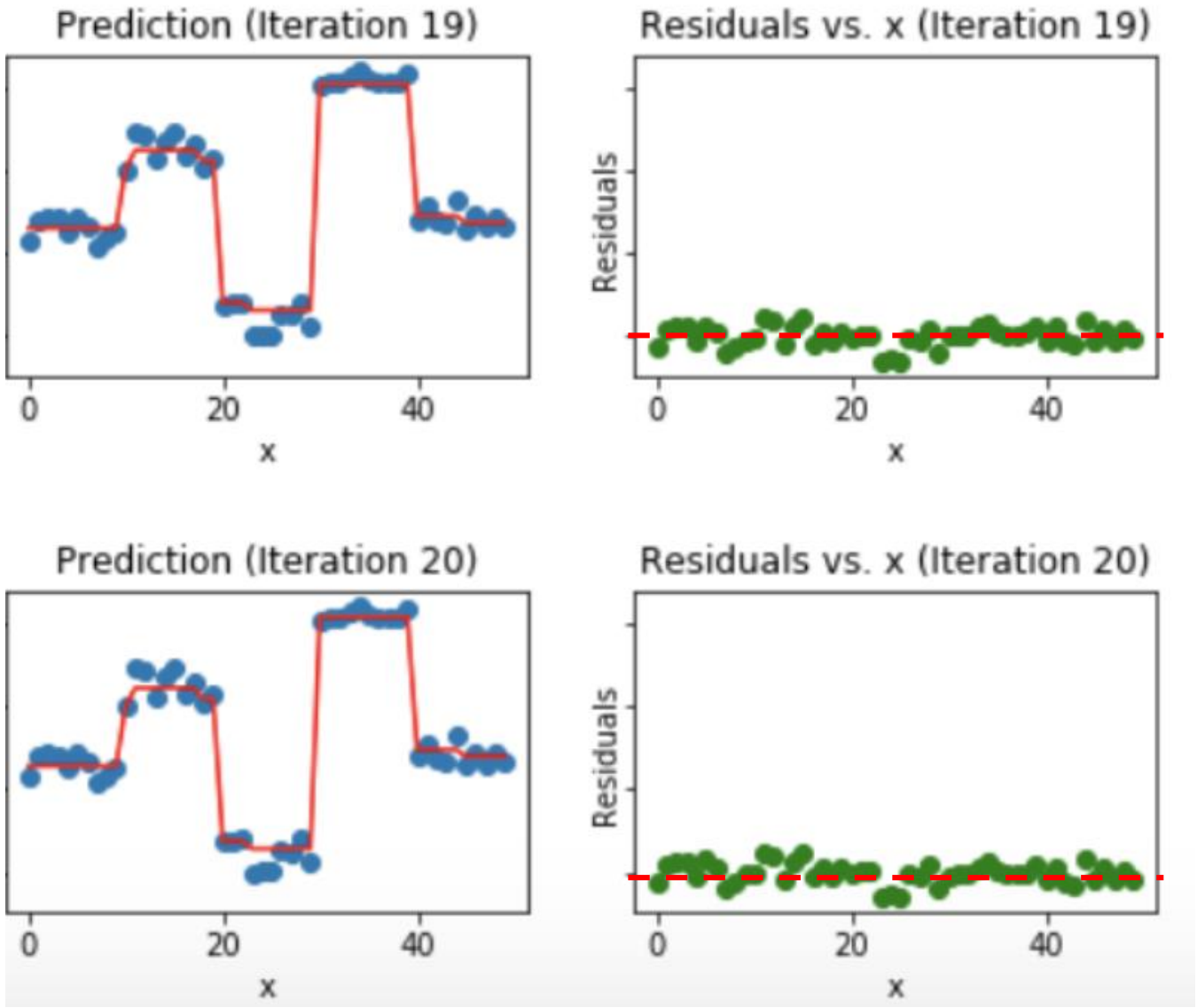
Two more iterations

We train on residuals and then add to previous model

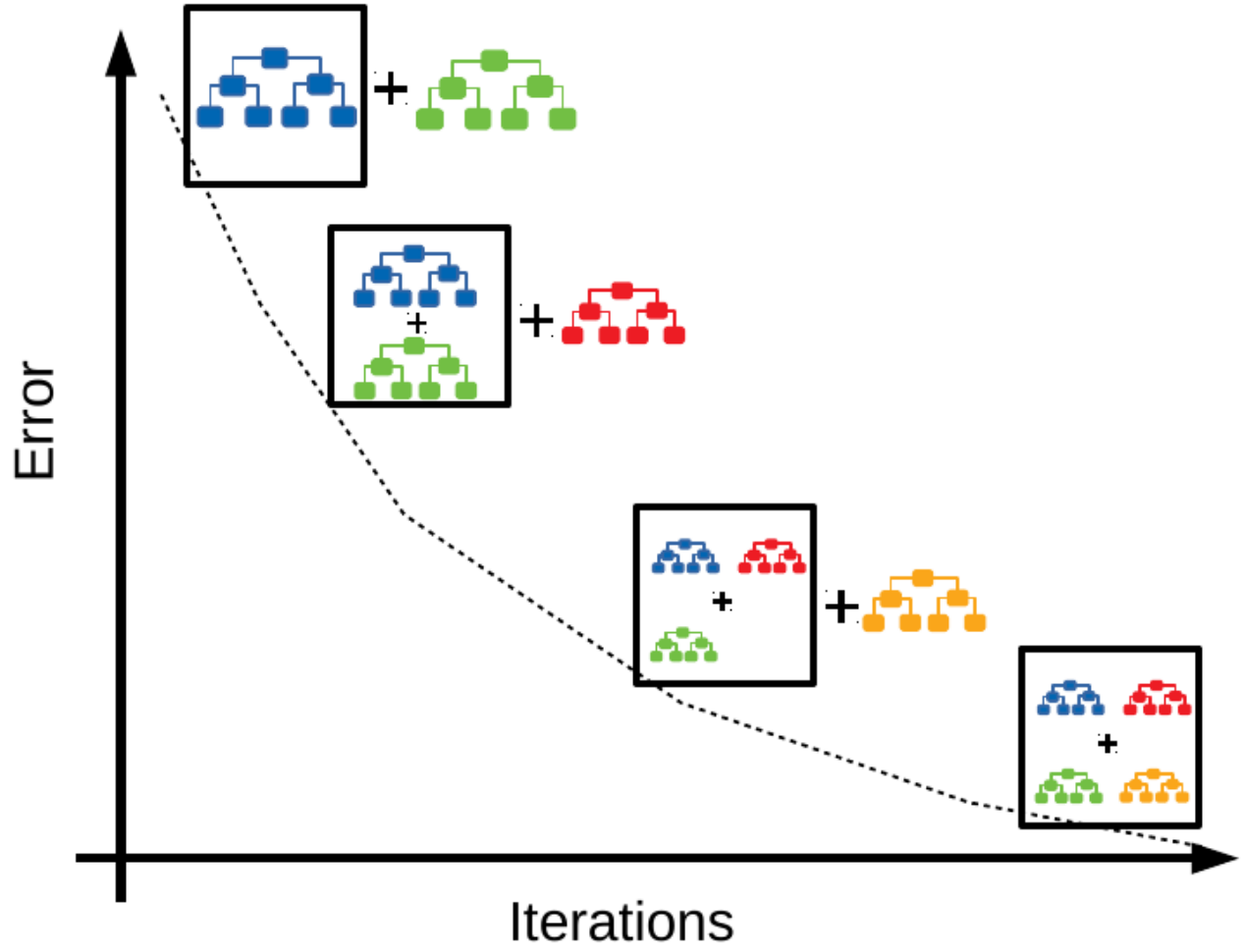


After several iterations

- Note that now residuals are randomly distributed around 0
- **There are no detectable patterns in errors**
- Note if we don’t stop the model may overfit!



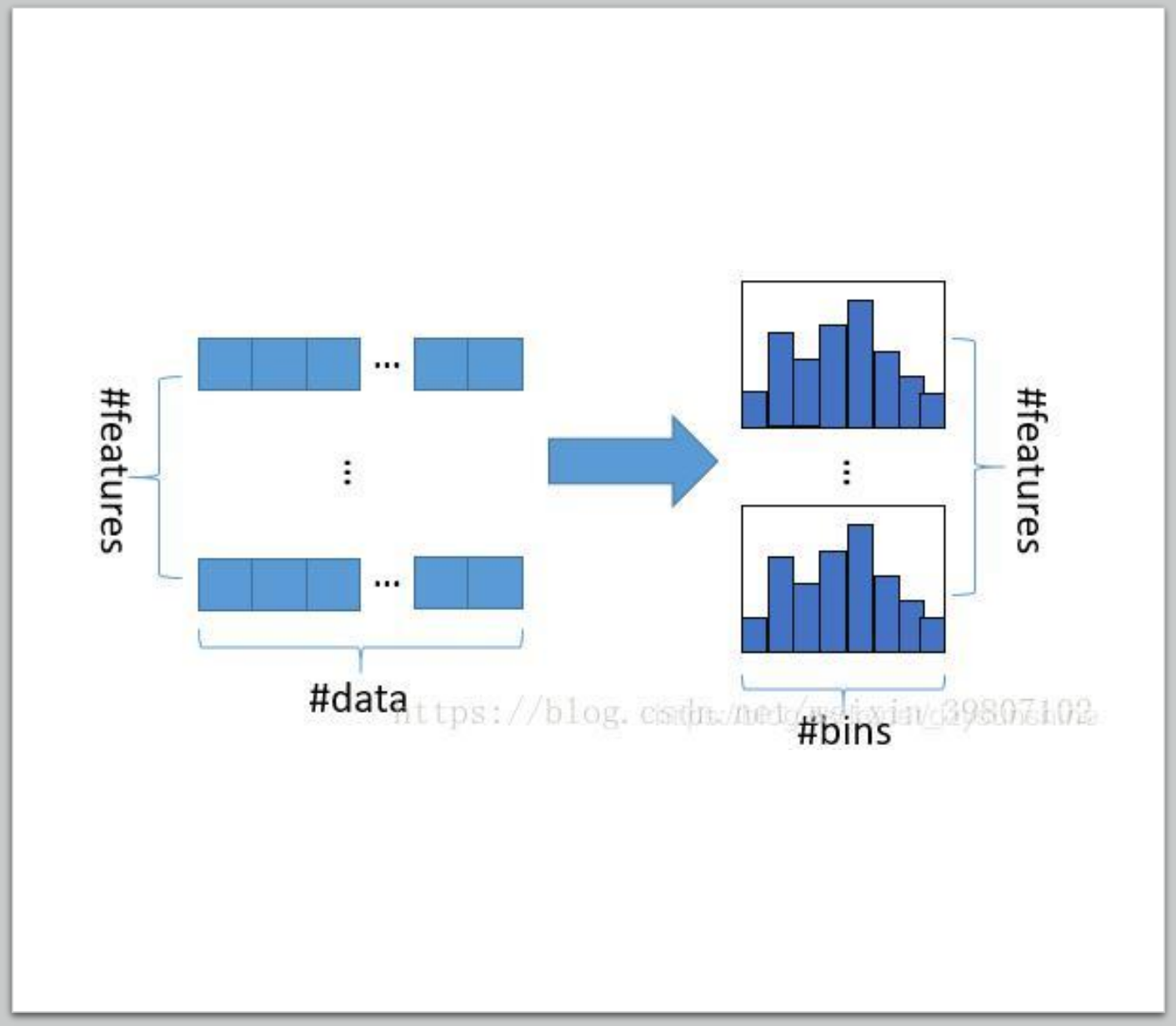
This visually shows what happens during gradient discent



https://morioh.com/p/e108a4521555

Popular variants of GBM: LGBM

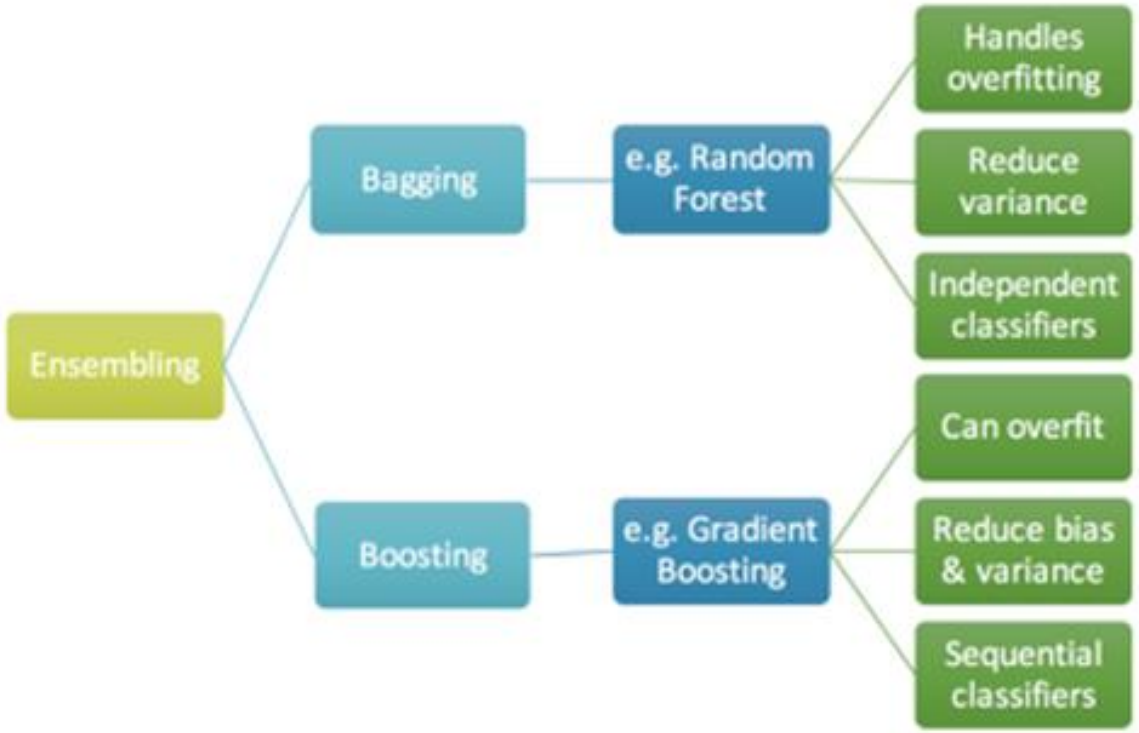
- Remember: «base» model in GBM is a regression tree (see lesson of regression trees and splitting method)
- LightGBM: rather than sorting all possible splits as in classic regression trees, LGBM uses ***Histogram based algorithm*** which buckets continuous features into discrete bins to construct feature histograms during training
- First, continuous data are discretized into k bins (e.g., 256), next, statistics are accumulated for each bin (frequency histogram of bins)
- Best split is built exploiting k bins statistics
- Obvious reduction of memory consumption, LGBM is advantageous with large datasets



Other variants of GBM

- XGB link
- CATBoost (for categorical features) link

Conclusion on Ensambles



Other Suggested Lectures

- General Overview: LINK and LINK
- Boosting and Bagging: LINK, LINK
- Random Trees and Extra Trees: LINK

- And the **heterogeneous** ensemble?
- Stacking and Bleeding: LINK