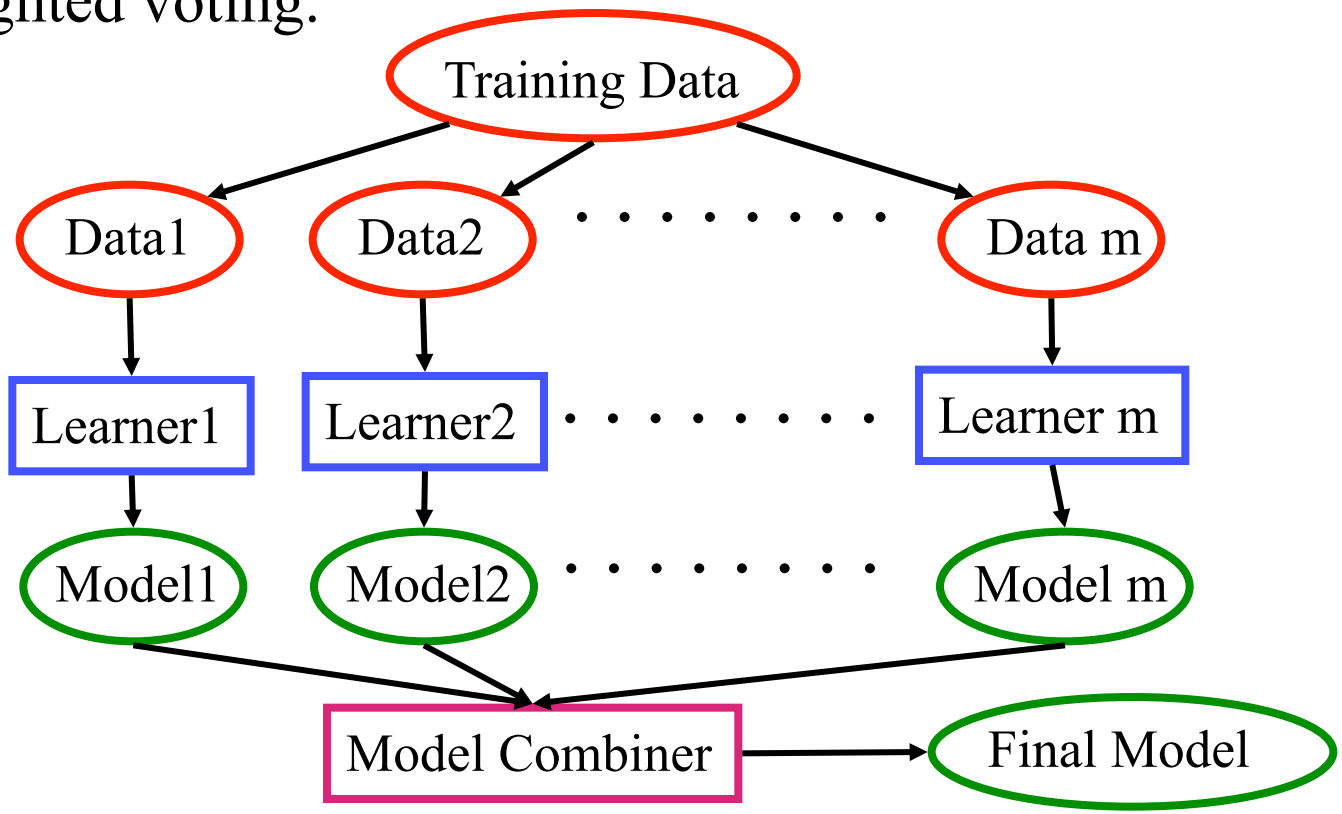
Machine Learning: Ensemble Methods

Learning Ensembles

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



Example: Weather Forecast

100% CORRECT!

GROUND TRUTH							
PREDICTOR1							
PREDICTOR2	X			X			X
PREDICTOR3			X		X	X	
PREDICTOR4			X		X		
PREDICTOR5		X				X	
Combine							

Why does it work?

- Suppose there are 25 “simple” classifiers
  - Each classifier has error rate, $\varepsilon = 0.35$ (which is a mid-high rate)
  - Assume classifiers are **independent**
  - Probability that the ensemble classifier makes a wrong prediction (it is wrong if at

IF CLASSIFIERS ARE INDEPENDENT, THE PROBABILITY THAT THE ENSAMBLE MAKES AN ERROR IS VERY LOW!!

$$\sum_{i=13}^{25} \binom{25}{i} \varepsilon^{i} (1-\varepsilon)^{25-i} = 0.06$$

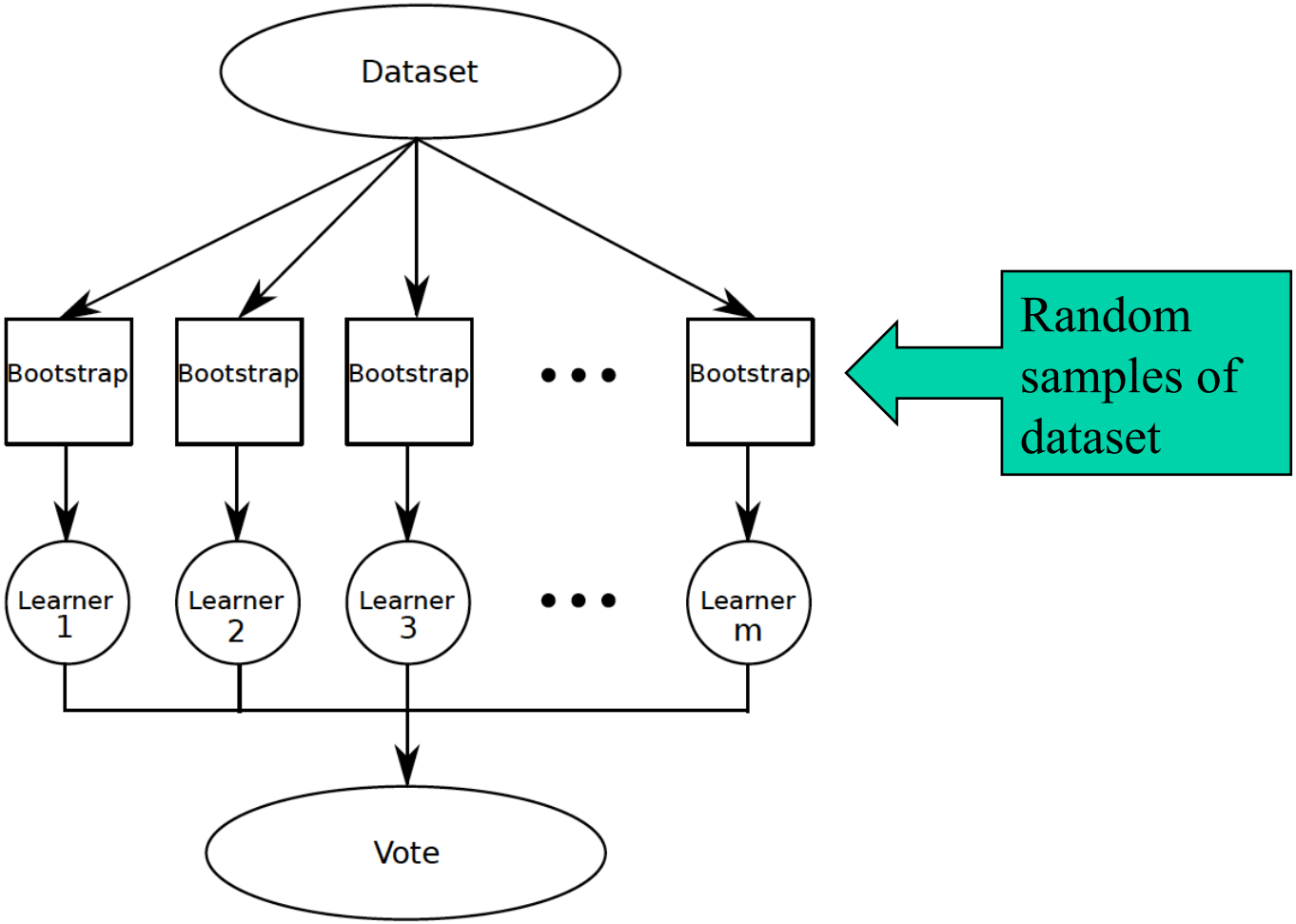
Value of Ensembles

- When combing multiple ***independent*** and ***diverse*** decisions each of which is at least <u>more accurate than random guessing</u>, random errors **cancel each other out**, correct decisions are reinforced.
- Human ensembles are demonstrably better
  - How many jelly beans in the jar?: Individual estimates vs. group average.
  - In information retrieval evaluation tasks, "ensamble" decision making is used

Homogenous Ensembles

- Use a **single, arbitrary learning algorithm** but **manipulate training data** to make it learn multiple models.
  - Data1 ≠ Data2 ≠ … ≠ Data m
  - Learner1 = Learner2 = … = Learner m
- Different methods for **changing training data**:
  - Bagging: Resample training data
  - Boosting: Reweight training data
  - DECORATE: Add additional artificial training data

I. Bagging (**BAGGING is acronym for Bootstrap AGGregatING**)



Bagging (BAGGING is short for Bootstrap AGGregatING)

- Create m samples of n data **with replacement (means same item can be resampled)**

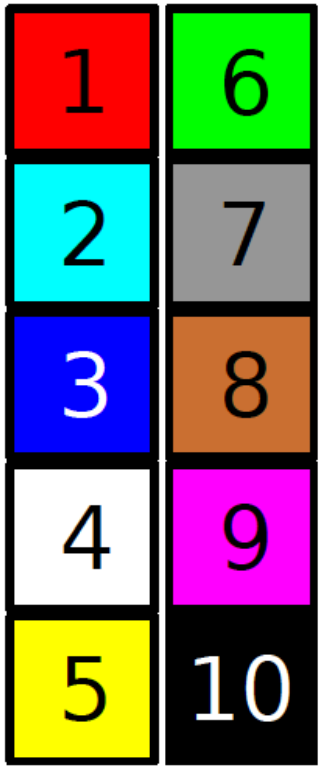
Data ID — Training Data

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 m=10 instances (#1, #2..#10)
- Each individual classifiers randomly extracts a sample of n instances (**n=m** in this example) **with replacement** (instances are put back in the urn, therefore they can be sampled more than one time)
- Each instance has probability of $1/m^n$ of being selected in a training sample and $(1 – 1/m)^n$ of being selected as test data, in each bagging round.

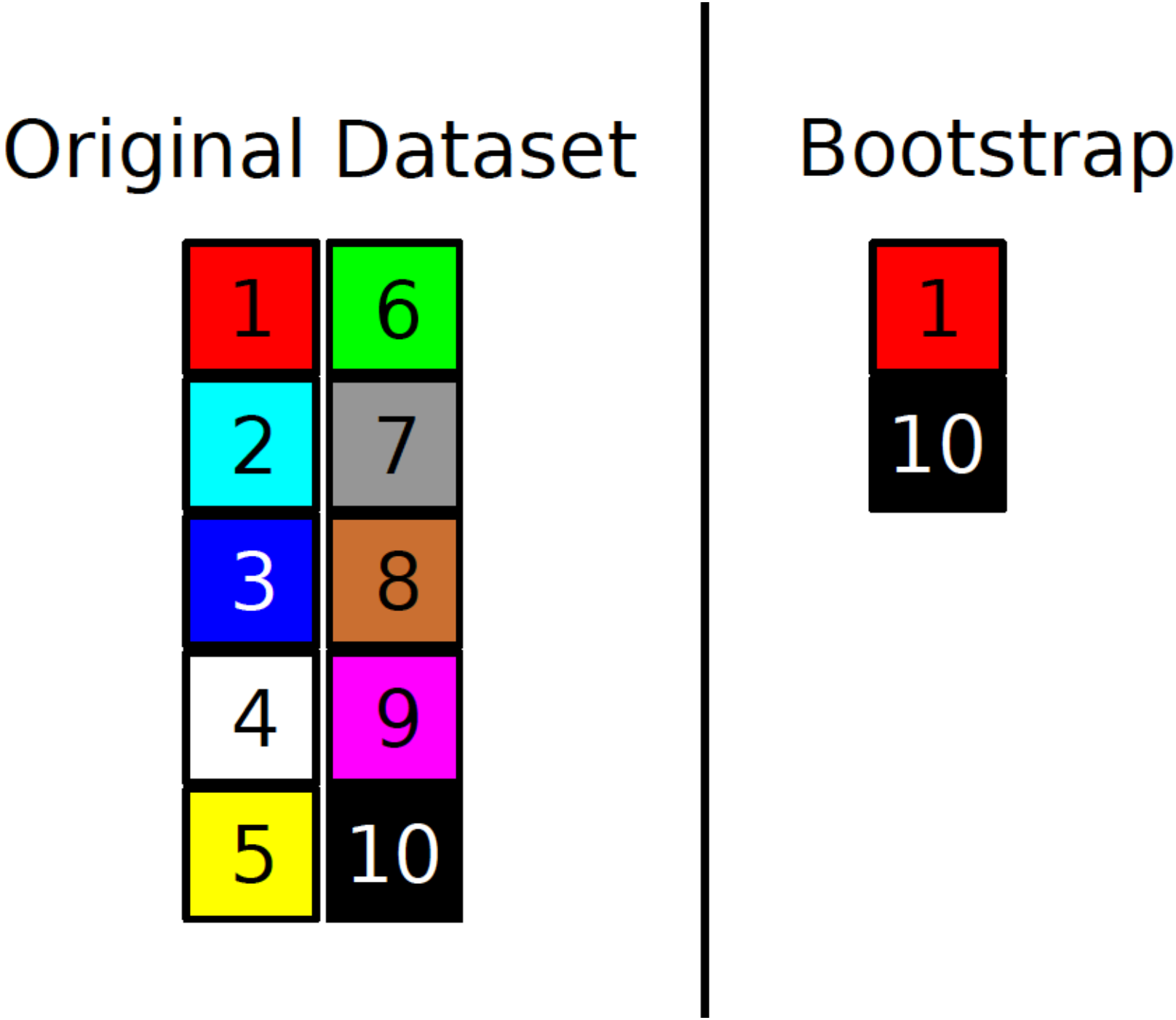
Example

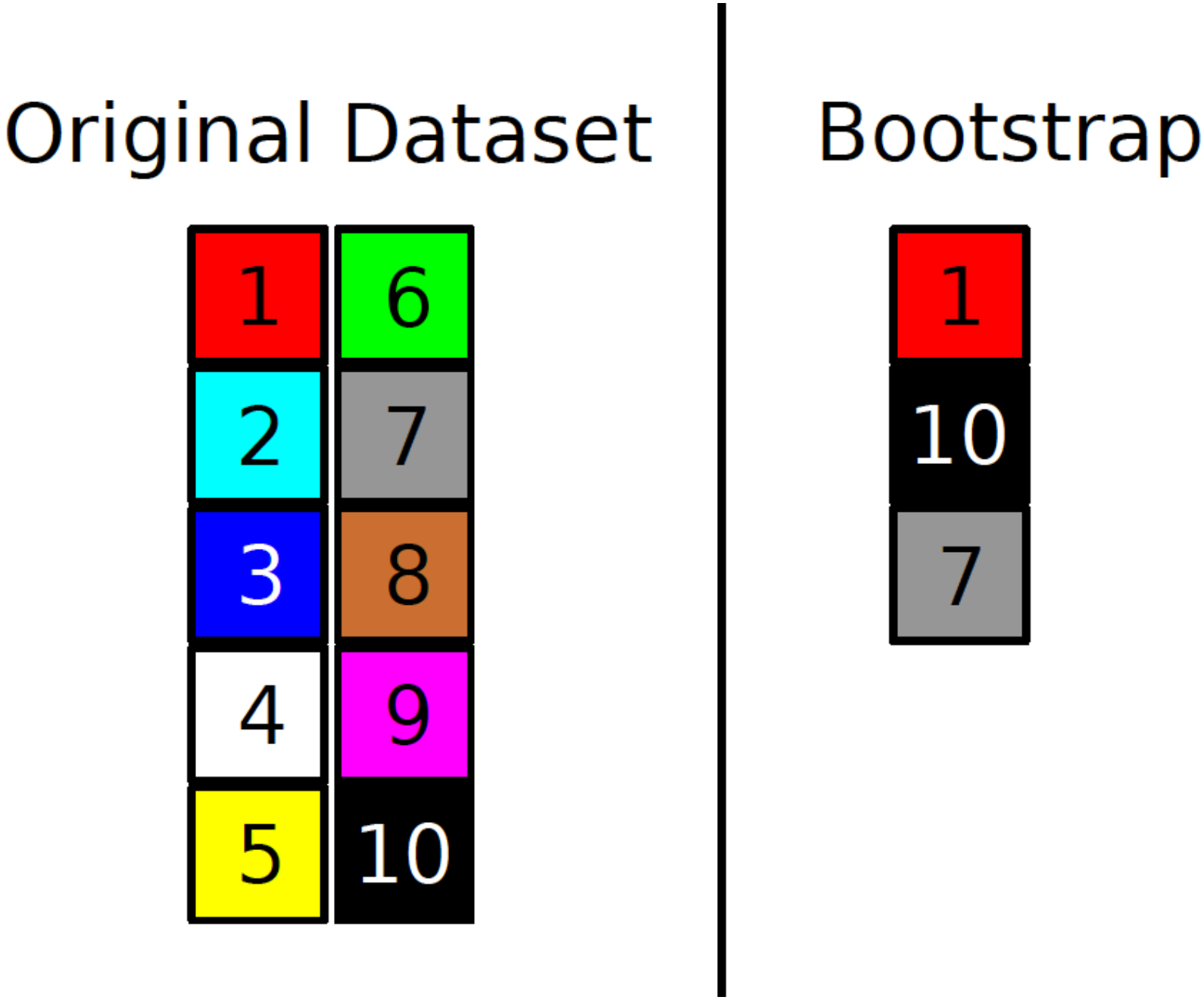
Original Dataset

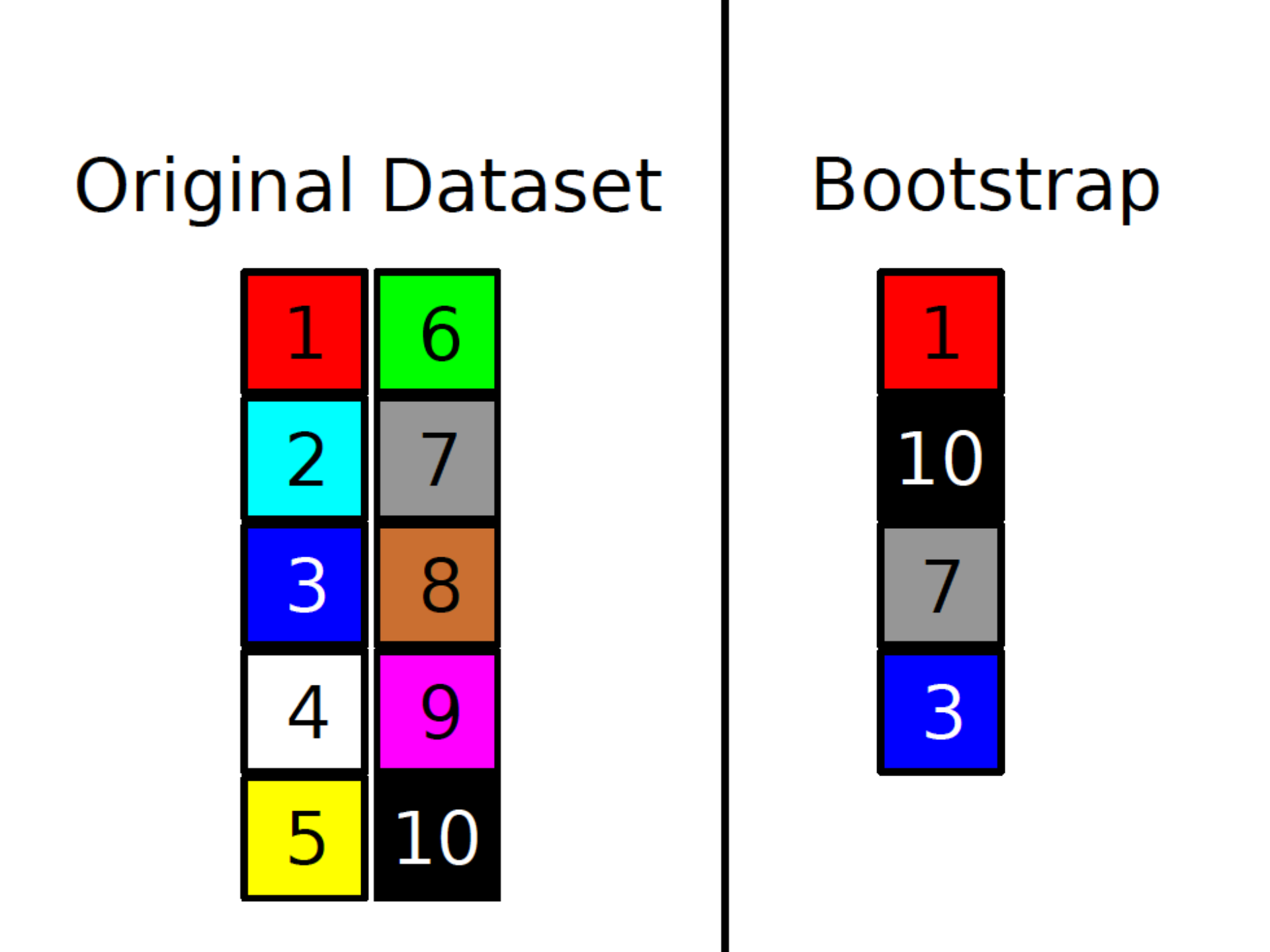


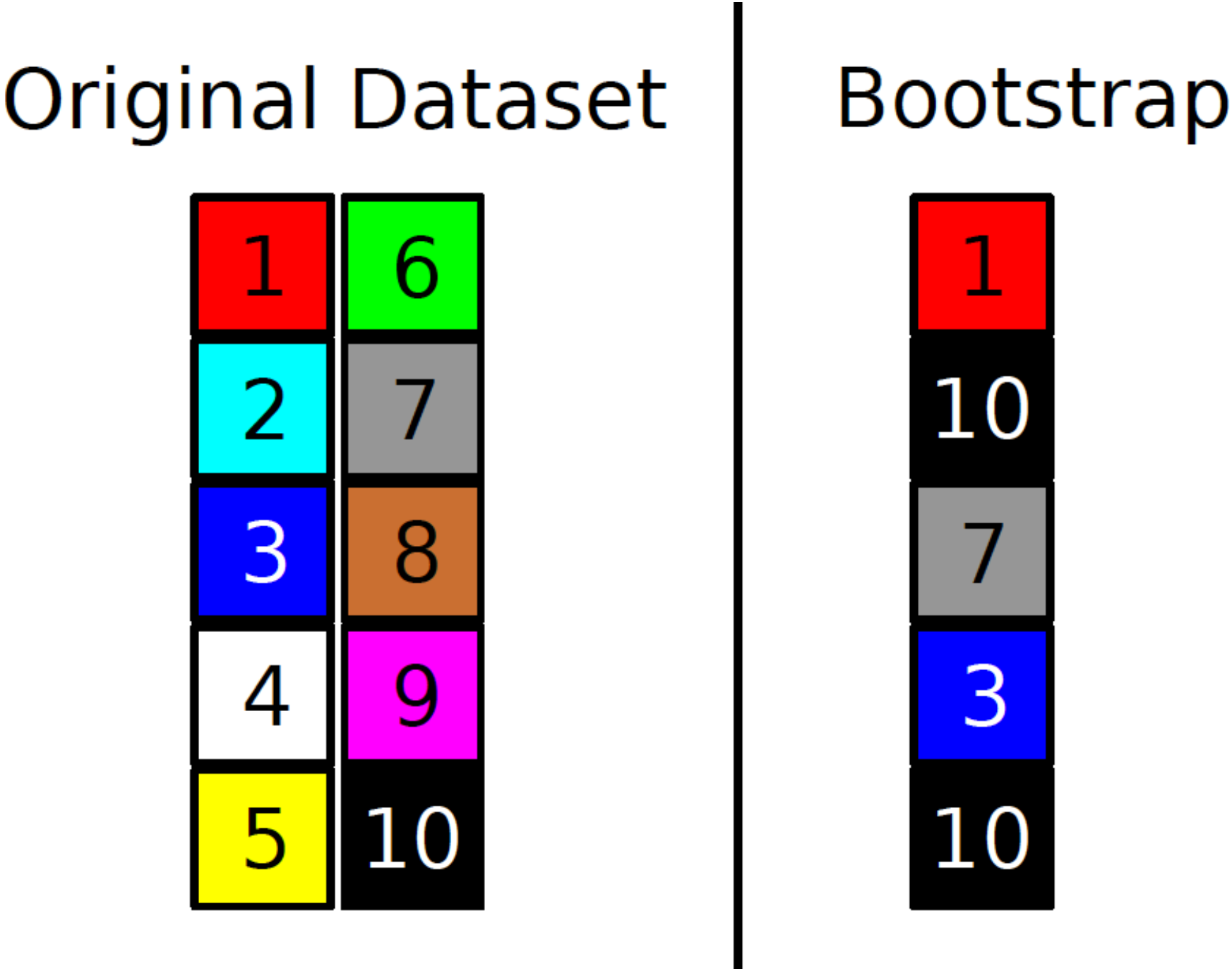
Bootstrap

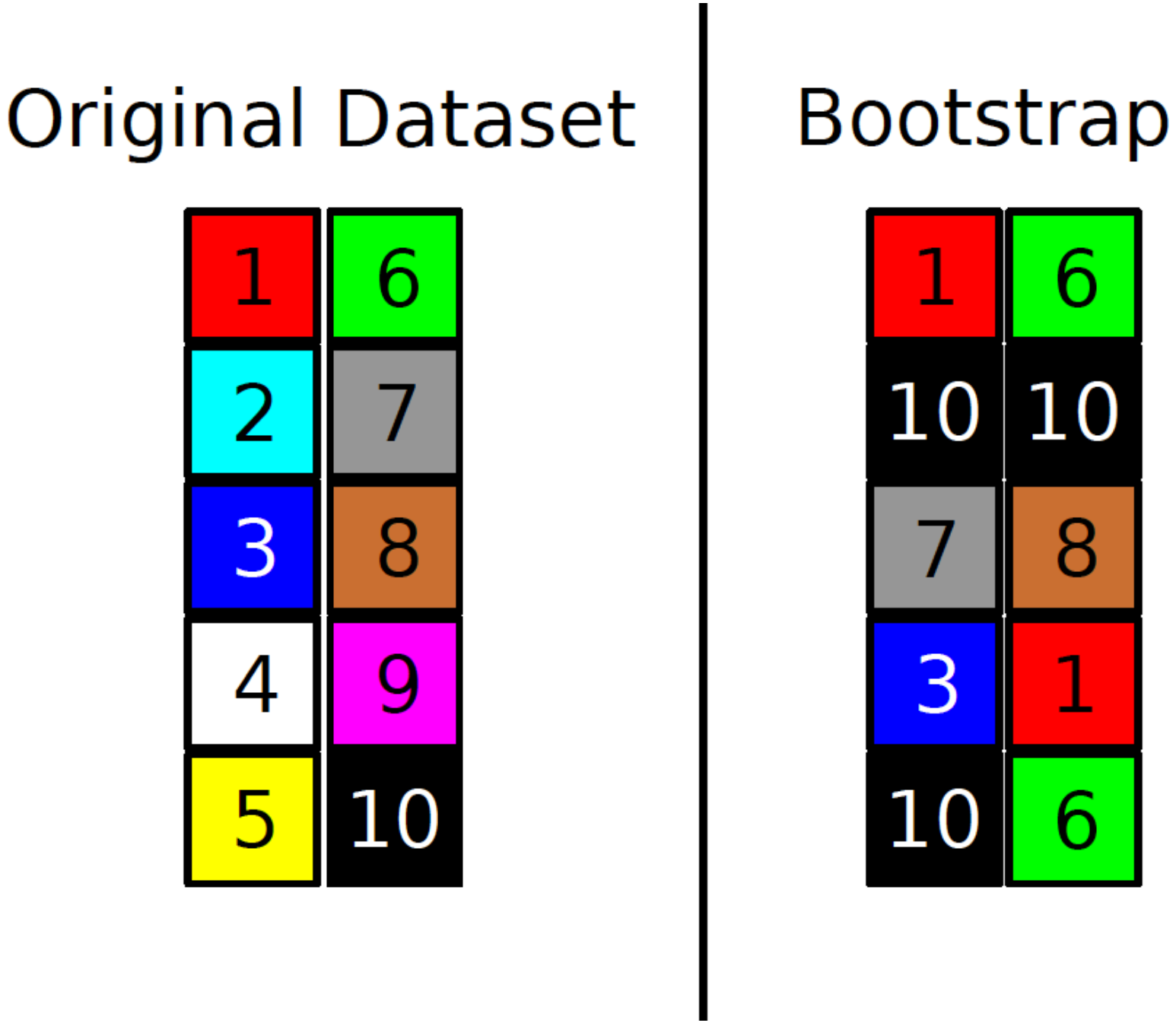
Each instance has a probability p=1/m of being extracted out of m instances. Since extraction is "with replacement" (the instance is put back in the urn after having been extracted) the probability is always the same at each extraction.


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


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


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


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


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

Training set

Test set

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

The 0.632 bootstrap

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

The 0.632 bootstrap

- This method is also called the *0.632 bootstrap*
  - If I make n extraction on n instances, each instance has a probability 1/n of being picked and 1-1/*n* of *not* being picked at each extraction
  - Thus its probability of ending up in the test data (=not being selected n times) 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 $\mathcal{R}^1$. Assume that the "real" (unknown) classification is:

Data is not linearly separable, a classifier for these data must learn a **range**, e.g.:
IF t1<=x<=t2 then C else not(C) In our example, "true" values are t1=0.3 and t2=0.8.

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

E.g. each classifier **ci** learn a single **threshold ti** such that:
If x<=ti then C else not(C)

Training set

So this is the learning 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 sample these data 10 times (thus we obtain 10 datasets), and on any sample we train a "simple" threshold classifier

Remember: "sampling" the dataset 10 times means that for 10 times ("bagging rounds") we extract 10 instances from the original dataset with replacement. The extracted instances in Round i are used to train the i-th learner, and non extracted instances are used for testing

Bagging Round 1:

x	0.1	0.2	0.2	0.3	0.4	0.4	0.5	0.6	0.9	0.9
y	1	1	1	1	-1	-1	-1	-1	1	1

<= 0.35 ==> y = 1
> 0.35 ==> y = -1

Bagging Round 2:

x	0.1	0.2	0.3	0.4	0.5	0.8	0.9	1	1	1
y	1	1	1	-1	-1	1	1	1	1	1

x <= 0.65 ==> y = 1
x > 0.65 ==> y = 1

Bagging Round 3:

x	0.1	0.2	0.3	0.4	0.4	0.5	0.7	0.7	0.8	0.9
y	1	1	1	-1	-1	-1	-1	-1	1	1

x <= 0.35 ==> y = 1
x > 0.35 ==> y = -1

Bagging Round 4:

x	0.1	0.1	0.2	0.4	0.4	0.5	0.5	0.7	0.8	0.9
y	1	1	1	-1	-1	-1	-1	-1	1	1

x <= 0.3 ==> y = 1
x > 0.3 ==> y = -1

Bagging Round 5:

x	0.1	0.1	0.2	0.5	0.6	0.6	0.6	1	1	1
y	1	1	1	-1	-1	-1	-1	1	1	1

x <= 0.35 ==> y = 1
x > 0.35 ==> y = -1

Bagging Round 6:

x	0.2	0.4	0.5	0.6	0.7	0.7	0.7	0.8	0.9	1
y	1	-1	-1	-1	-1	-1	-1	1	1	1

x <= 0.75 ==> y = -1
x > 0.75 ==> y = 1

Bagging Round 7:

x	0.1	0.4	0.4	0.6	0.7	0.8	0.9	0.9	0.9	1

x <= 0.75 ==> y = -1

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

Note: in each round the same training example can be extracted more than one time, and some examples are not extracted. Furthermore, each classifier is inconsistent! E.g. classifier 1 is wrong on last two items of "sampled" learning set: c(0.9)=-1

Bagging Round 10:

x	0.1	0.1	0.1	0.1	0.3	0.3	0.8	0.8	0.9	0.9
y	1	1	1	1	1	1	1	1	1	1

x <= 0.05 ==> y = -1
x > 0.05 ==> y = 1

Figure 5.35. Example of bagging.

Combining the different learned classifiers

- 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 with an error rate ε_i
- We then need to <u>combine</u> the results (**ensemble method**)
- A simple method (for binary classifiers with values +1, -1): if $sign(\Sigma_i C_i(x_j))=1$, then $C(x_j)=1$
- This means: if majority says "1" then, predicted class is 1.
- More in general, we can use **majority voting**

Bagging (if applied to training data)

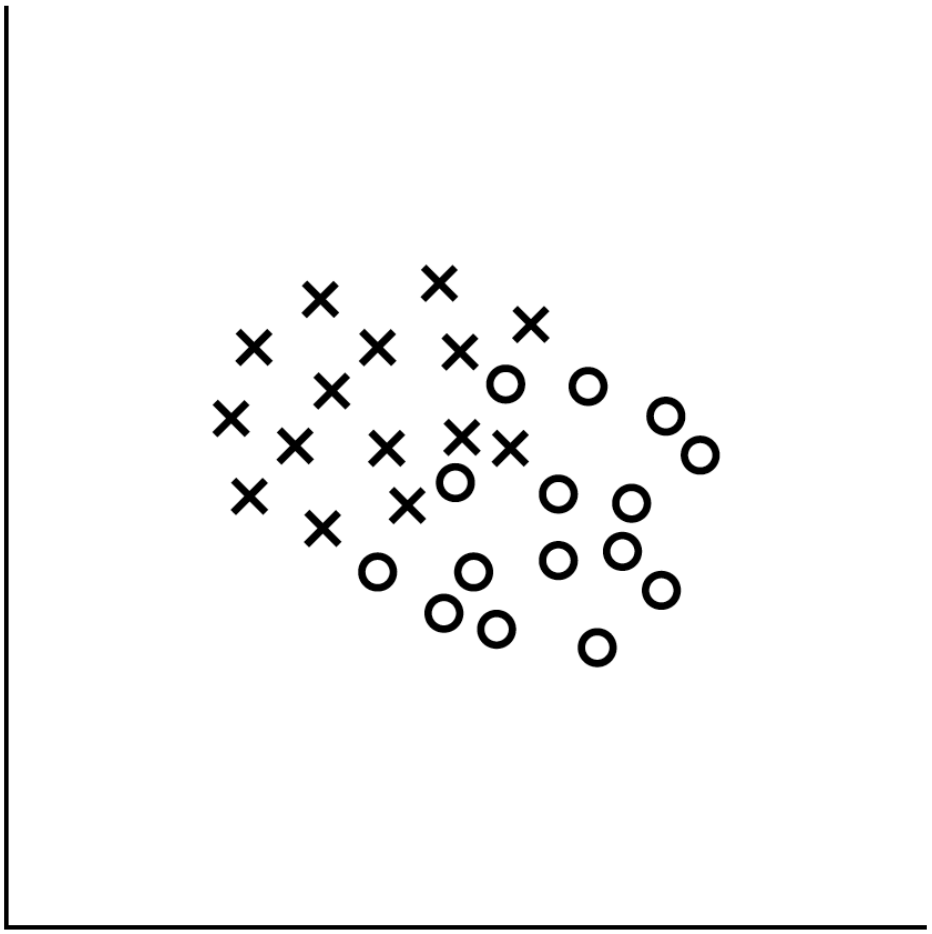
Round	x=0.1	x=0.2	x=0.3	x=0.4	x=0.5	x=0.6	x=0.7	x=0.8	x=0.9	x=1.0
1	1	1	1	-1	-1	-1	-1	-1	-1	-1
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	-1	-1	-1	-1	-1	-1	-1
4	1	1	1	-1	-1	-1	-1	-1	-1	-1
5	1	1	1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	-1	-1	1	1	1
7	-1	-1	-1	-1	-1	-1	-1	1	1	1
8	-1	-1	-1	-1	-1	-1	-1	1	1	1
9	-1	-1	-1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

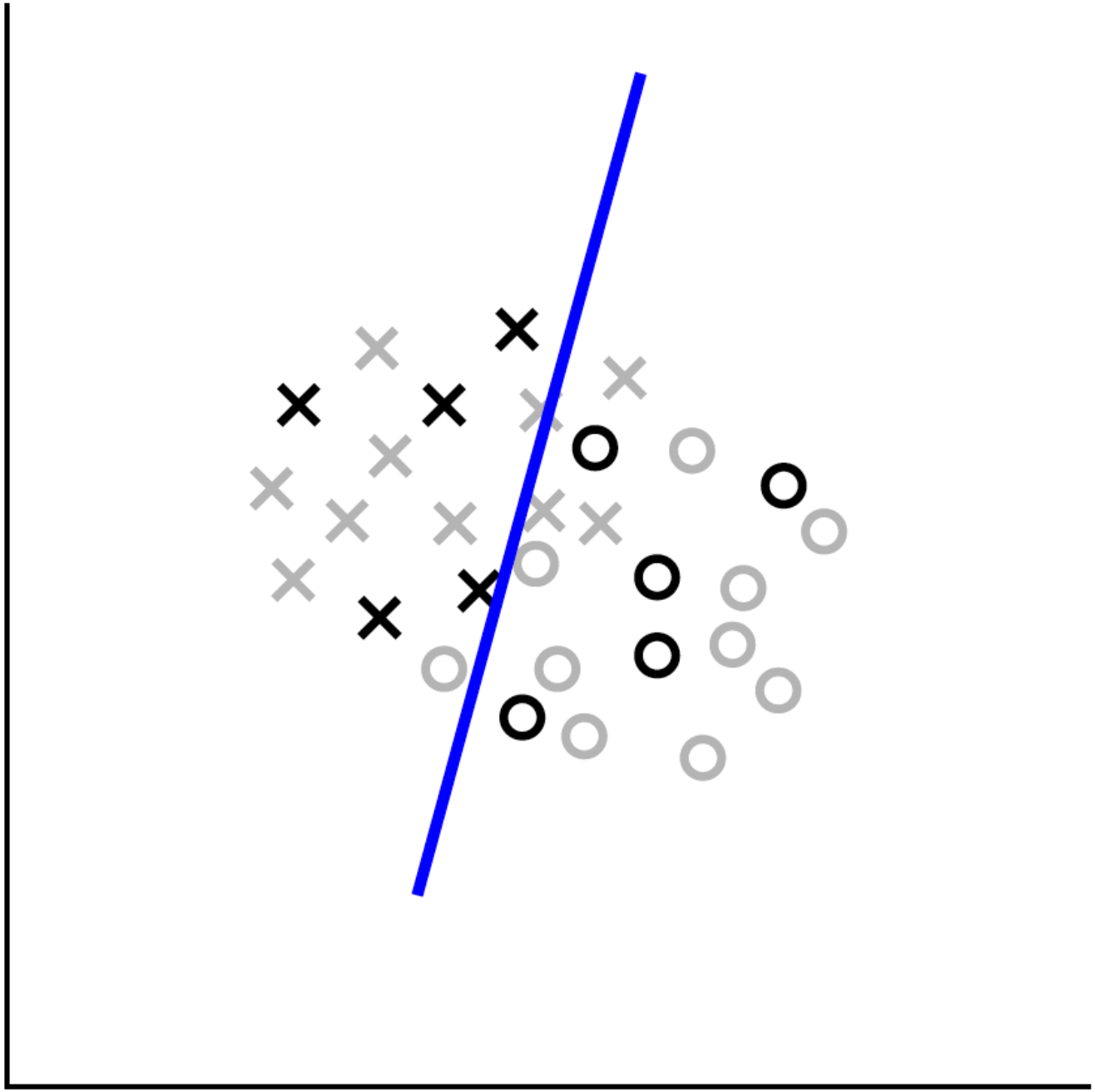
Figure 5.36. Example of combining classifiers constructed using the bagging approach.

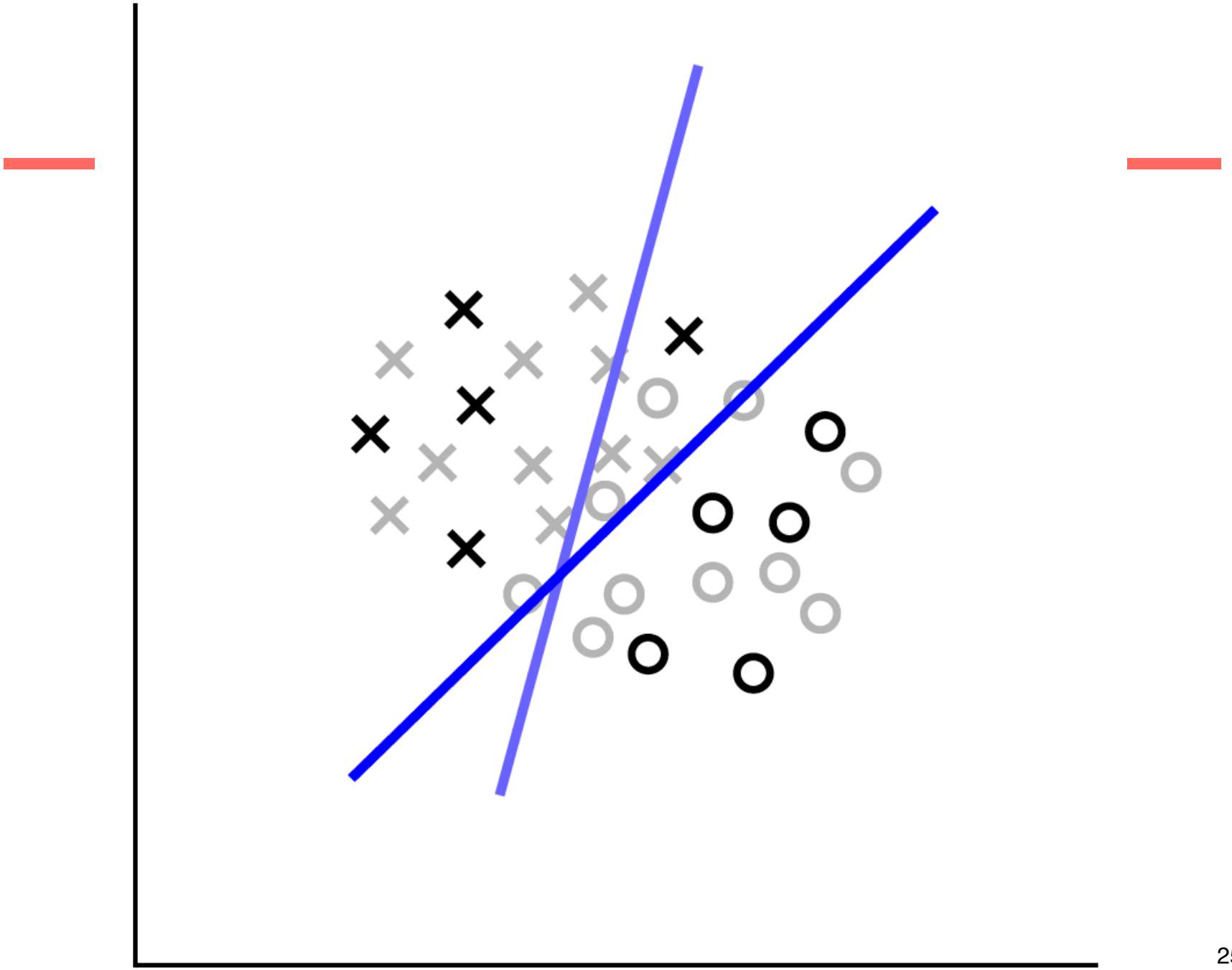
$$If \;\; sign(\sum C_i(x)) = 1 \;\; then \;\; C(x) = 1$$

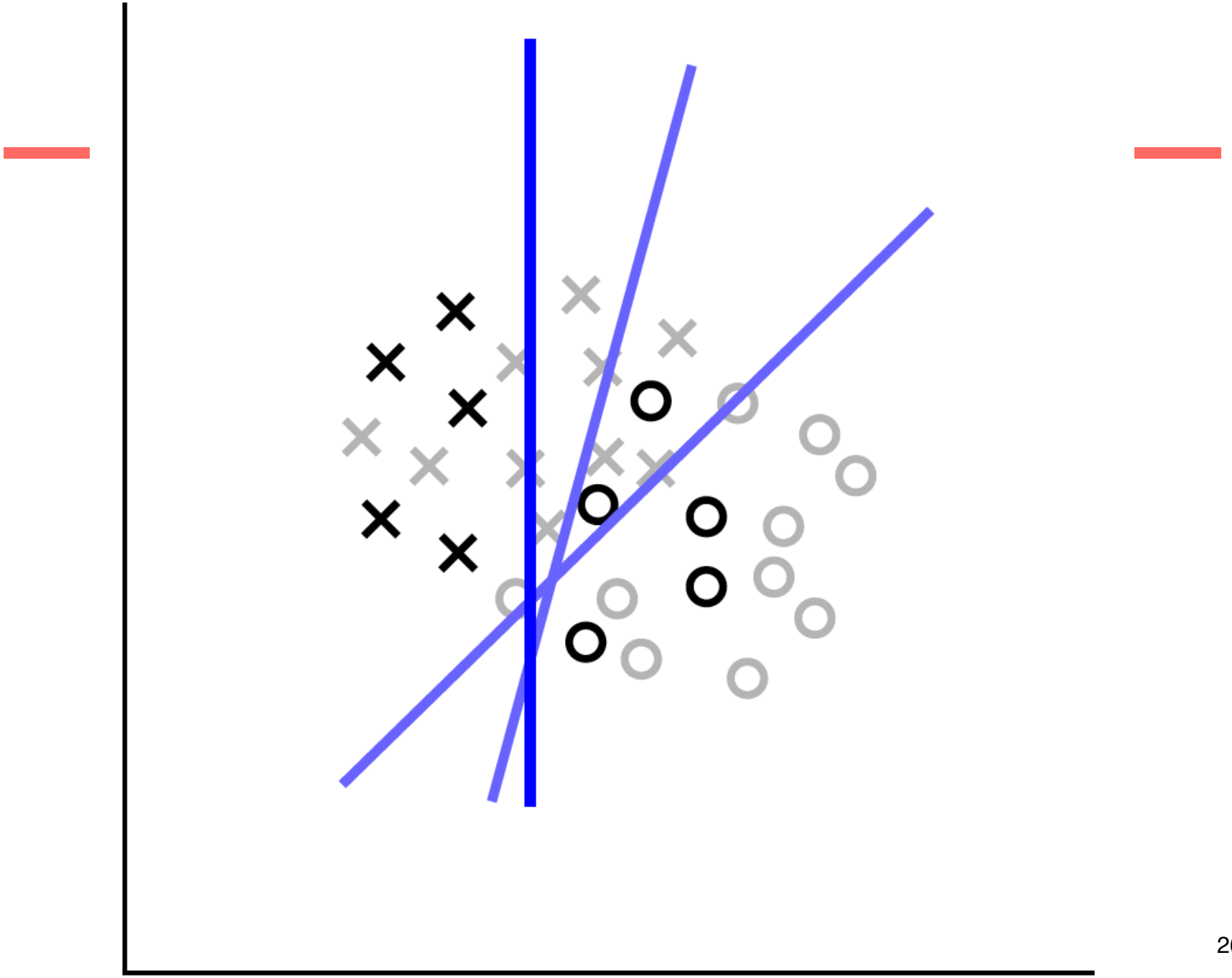
Accuracy of ensemble classifier: 100% ☺

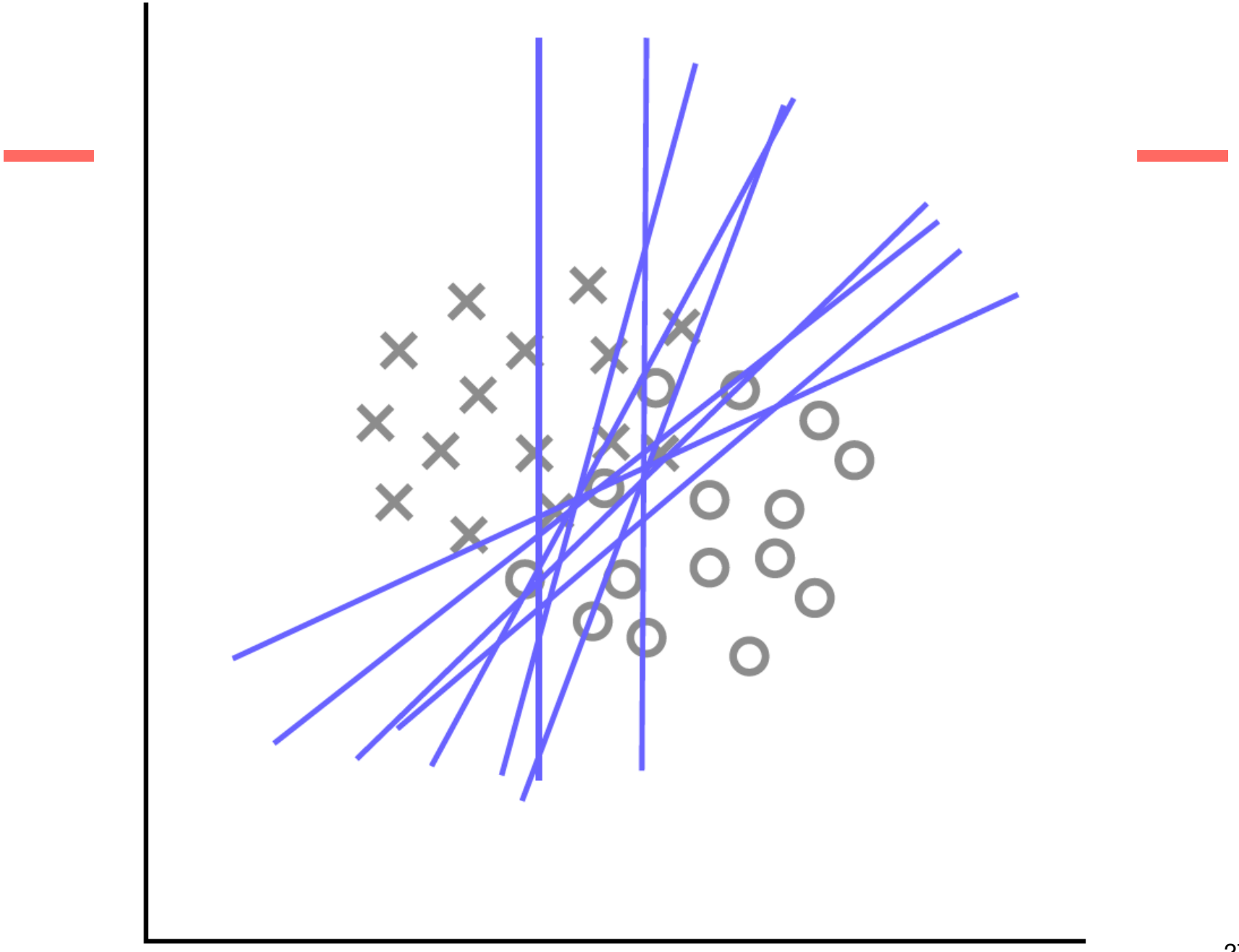
Example 2 of ensembles: non-linear classifier out of many linear classifiers (e.g perceptrons)

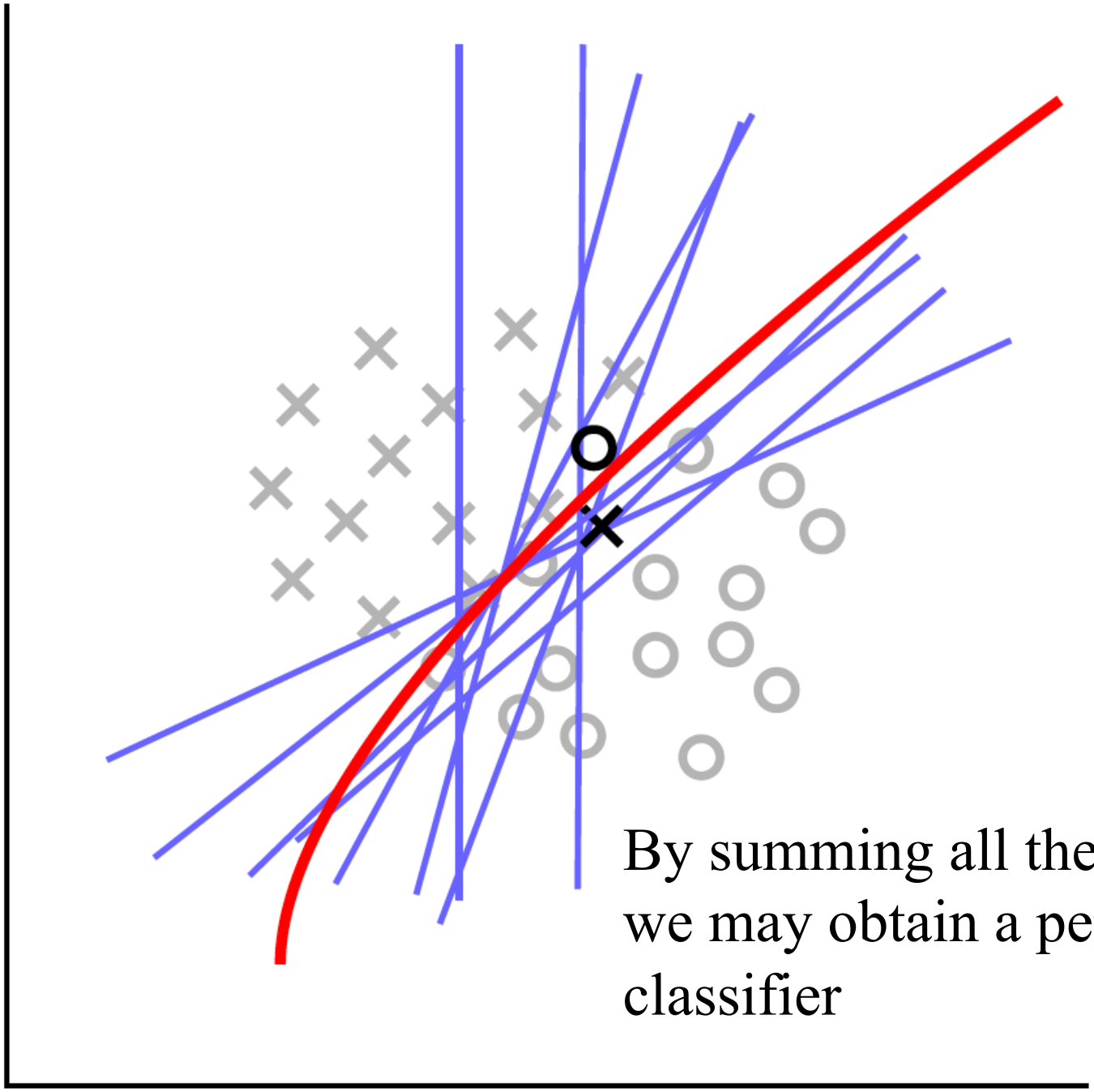












By summing all the lines we may obtain a perfect classifier

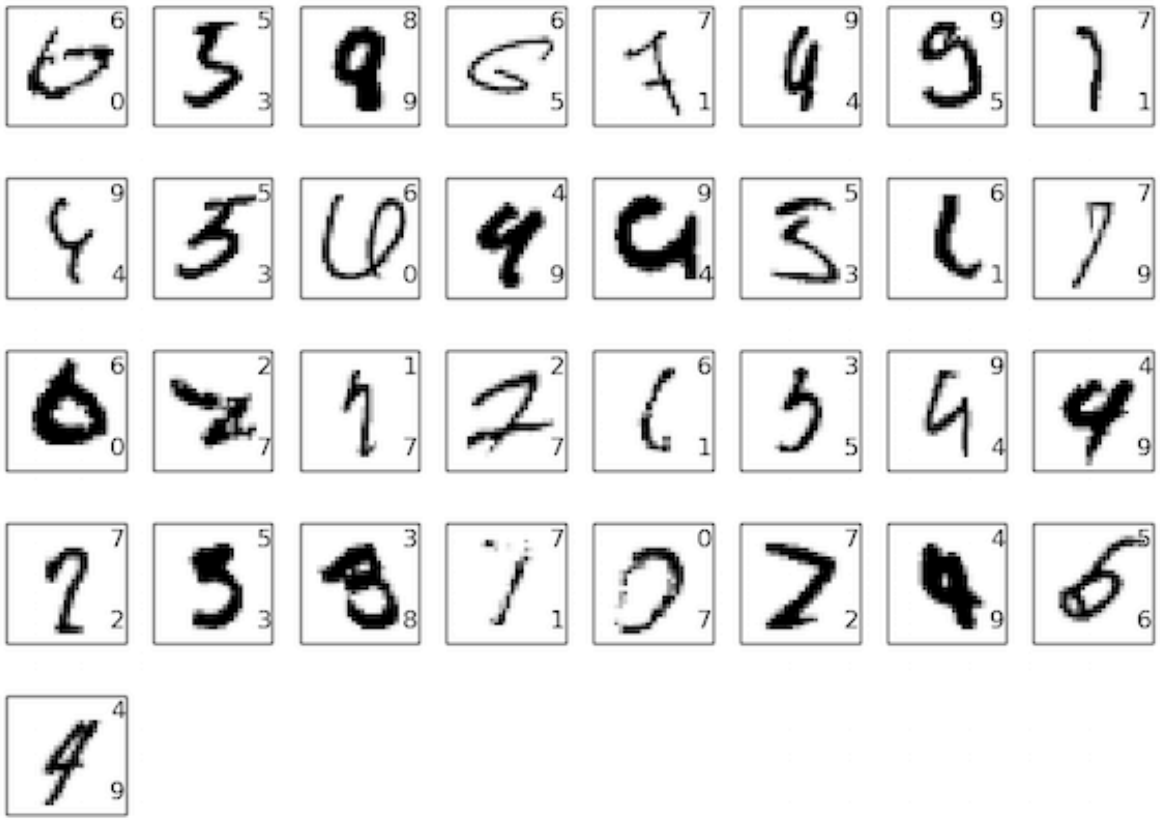
N simple classifiers work like a complex classifier

- Note: initial data could not be correctly separated by a simple threshold/linear classifier
- **With bagging , we obtain a perfect classifier!**

Bagging- Summary

- Works well if all instances have equal probability of being classified correctly or wrongly (means: $Pr(c(x) \neq h(x)) = p$ for all x in X)
- Increased accuracy because it ***reduces the variance*** of the individual classifier, by averaging over many
- Does not focus on any particular instance of the training data- assumption is that all instances have same probability of misclassification
- What if we want to focus on a particular instances of training data?
- E.g. some instance can be more difficult to classify than others (and on these instances most "simple" classifiers may err, so majority voting won't work)

Example 1: handwriting recognition



Example 2: face recognition

II. Boosting

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

1. Get a dataset.
2. Take a bootstrap, and train a model on it.
3. See which examples the model got **wrong**.
4. Upweight those 'hard' examples, downweight the 'easy' ones.
5. Go back to step 2, but with a *weighted* bootstrap.

Each new member of the ensamble focuses on the instances that the previous ones got wrong!

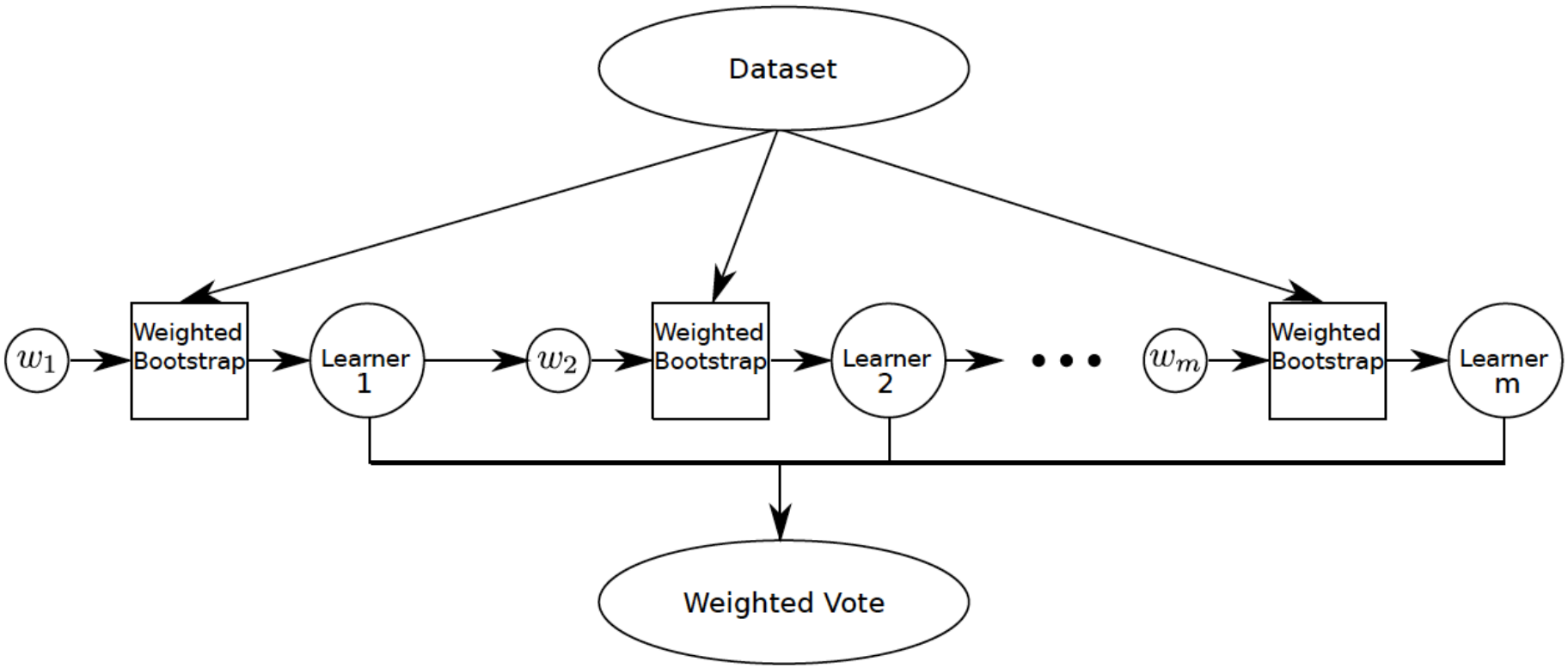
Boosting (2)

- Instances xi are extracted with a probability that depends on their weight w_i ($P(X=x_i)=w_i$)
- In iteration j, instances that are wrongly classified when testing the classifier cj will have their weights increased
- Those that are classified correctly will have their weights decreased

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

- Suppose example 4 is hard to classify (round 1 is wrong on example 4)
- Its weight is increased, therefore it is more likely to be extracted again in subsequent sampling rounds (2)
- If round 2 is again wrong on example 4, it probability of being extracted increases again

Boosting flow diagram



The key idea is that, since the learning set (in step i) includes more examples of the "complex" cases, the current classifier Ci is trained on those cases.
E.g., recognizing faces of people with a hat, or with dark glasses.

Boosting (3)

- At first, equal weights are assigned to each training instance (1/n for round 1), so all instances have the same probability of being sampled
- After a classifier C_i is learned, the instance weights are adjusted to allow the subsequent classifier C_{i+1} to "pay more attention" to data that were misclassified by C_i. Higher weights ➔ higher probability for an instance of being extracted
- Final boosted classifier C* combines the votes of each individual classifier
  - Weight of each classifier's vote is a function of its accuracy
- **Adaboost** – most popular boosting algorithm

Adaboost (Adaptive Boost)

- Input:
  - Training set D containing n instances
  - T iterations, or “rounds” (i=1…T)
  - A classification learning scheme
- Output:
  - A composite model

Adaboost: Training Phase

- Training data D contain n labeled data (X_1,y_1), (X_2,y_2), (X_3,y_3),….(X_n,y_n) y_i are the correct classifications
- Initially assign equal weight **1/n** to each example
- To generate *T* "base" classifiers, we need *T* rounds or iterations
- Round i, examples (instances) from D are sampled **with replacement** , to generate dataset Di (of size *n*)
- Each instance chance of being selected in the next rounds **depends on its weight**
  - Each time the new sample is generated directly from the training data D with **different sampling probability according to the weights**;

Testing phases in AdaBoost

- Testing occurs on individual classifiers C_i at the end of each round.
- The performance of each classifier is used to assess the "importance" or authority of C_i
- Final testing is performed on unseen data. To combine individual classifications by each Ci, the decision of each classifier is taken into consideration **proportionally to its importance**

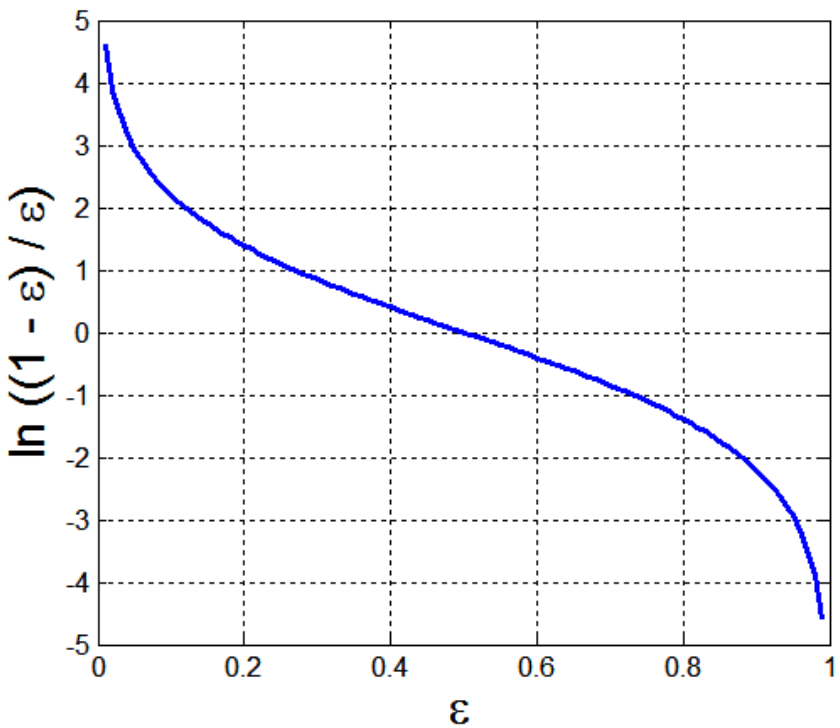
Testing phase for individual classifiers in AdaBoost

- "Base" learned classifiers: C_1, C_2, …, C_T
- Error rate of C_i: (i = index of classifier, j=index of instance, $C(x_j)=y_j$ correct class for x_j)

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

- Importance of a classifier:

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



Final Testing Phase

- The lower a classifier's error rate ε_i , the more accurate it is, and therefore, **the higher its weight when voting should be**
- **Weight** of a classifier C_i' s vote is $\alpha_i = \frac{1}{2}\ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$
- Final Testing (on unseen data):
  - For each class y_j, sum the weights of each classifier that assigned class y_j to instance X_{test}. The class with the highest sum is the WINNER!

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

$$\delta(x) = 1 \;\; if \;\; x = true$$

It is again a majority voting but votes of each classifier are weighted by its importance

Training Phase of C_i depends on previous testing phase on C_{i-1}

- Base classifier C_i, is derived from training data of set D_i
- Error of C_i is tested **using Di** (same data)
- Weights of training data are **adjusted** depending on how they were classified
  - Correctly classified: Decrease weight
  - Incorrectly classified: Increase weight
- Weight of a data indicates how hard it is to classify it

Weighting rule for data in AdaBoost

- Weight update rule **on all training data in *D***:

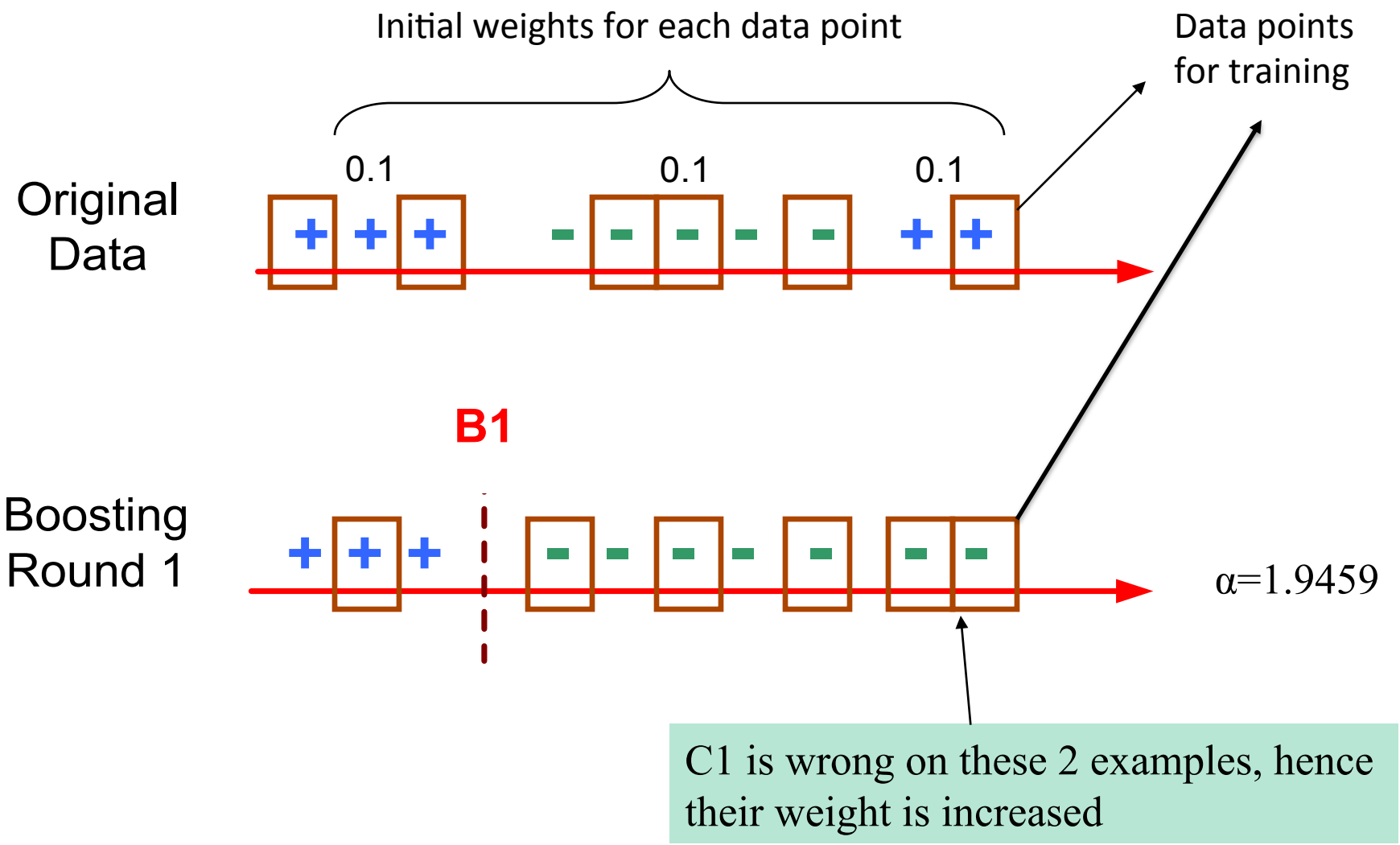
$$w_j^{(i+1)} = \frac{w_j^{(i)}}{Z_i} \times \begin{cases} \exp^{-\alpha_i} & \textbf{if } C_i(x_j) = y_j \\ \exp^{\alpha_i} & \textbf{if } C_i(x_j) \neq y_j \end{cases}$$

where Z_i is *a* normalization factor

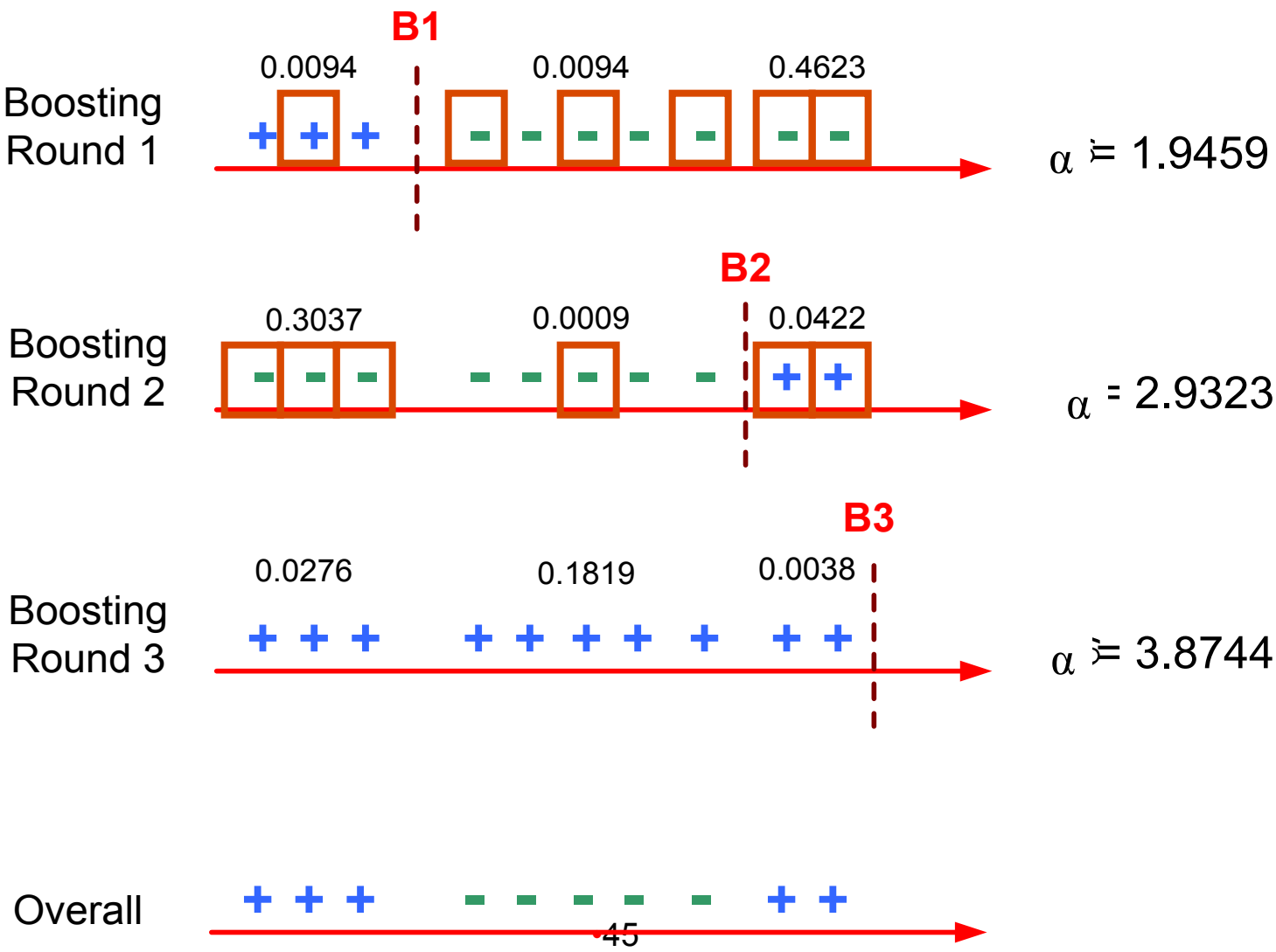
α_i is the "importance" of classifier C_i, as previously computed

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

Illustrating AdaBoost



Illustrating AdaBoost



Adaboost pseudo-code (summary)

Given D:<xi,yi> |D|=n

1. Initialize 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. Iff $\varepsilon_i>1/2$ then T=t-1 abort loop
   a. Compute αi
   b. Update wj
4. Output: for any unseen x_{test}

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

III. Random Forests

- Ensemble method specifically designed for **decision tree classifiers**
- Random Forests grows many trees
  - Ensemble of **unpruned** decision trees
  - Each base classifier classifies a "new" vector of attributes from the original data
  - Final result on classifying a new instance: 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" and "Random input vectors"
  - 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, best attribute to test is chosen from a **random sample of *m*** attributes, rather than from all attributes

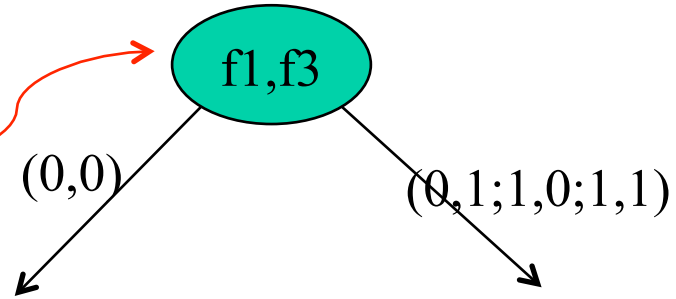
Random forest algorithm

- Let the number of training instances be *N*, and the number of features (attributes) describing instances be *M*.
- We are told the number *m* of input features to be used to determine the decision at a node of the tree; *m* should be much less than *M*
- Choose a training set D_i for tree DT_i by choosing *n* times **with replacement** from all *N* available training cases (i.e. take a bootstrap sample). Use the rest of examples to estimate the error of DT_i.
- For each node of the tree, **randomly choose** *m* features on which to base the decision at that node. Calculate the best split based on these *m* variables in the training set (either test on best attribute in m based on Infogain, - or select the best binary split based on IG).
- Each tree is fully grown and **not pruned** (as may be done in constructing a normal tree classifier).

At each node:
choose some ballsubset of variables at random
find a variable (and a value for that variable) which optimizes the split

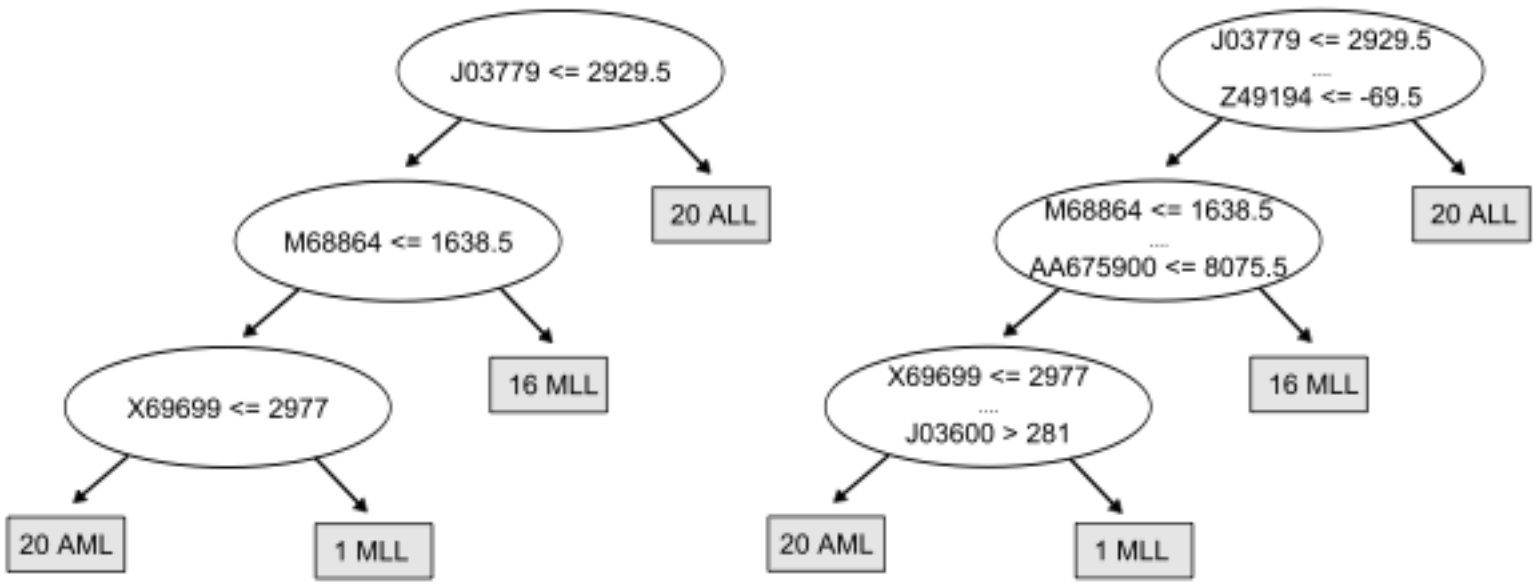
Best binary split, example

- Say we select m=2 and sample two binary features at random, e.g., f1 and f3
- Possible splits:
  - (0,0) (0,1; 1,0; 1,1)
  - (0,1) (0,0;1,0; 1,1)
  - (1,0) (0,0;0,1; 1,1)
  - (1,1) (0,0;1,0; 1,0)



- Select the one with highest IG (or other selection methods)

Decision forest with multiple test



m=1 m>1

Leukemia MLL vs ALL vs AML based on marker genes

Random Forests – combining results

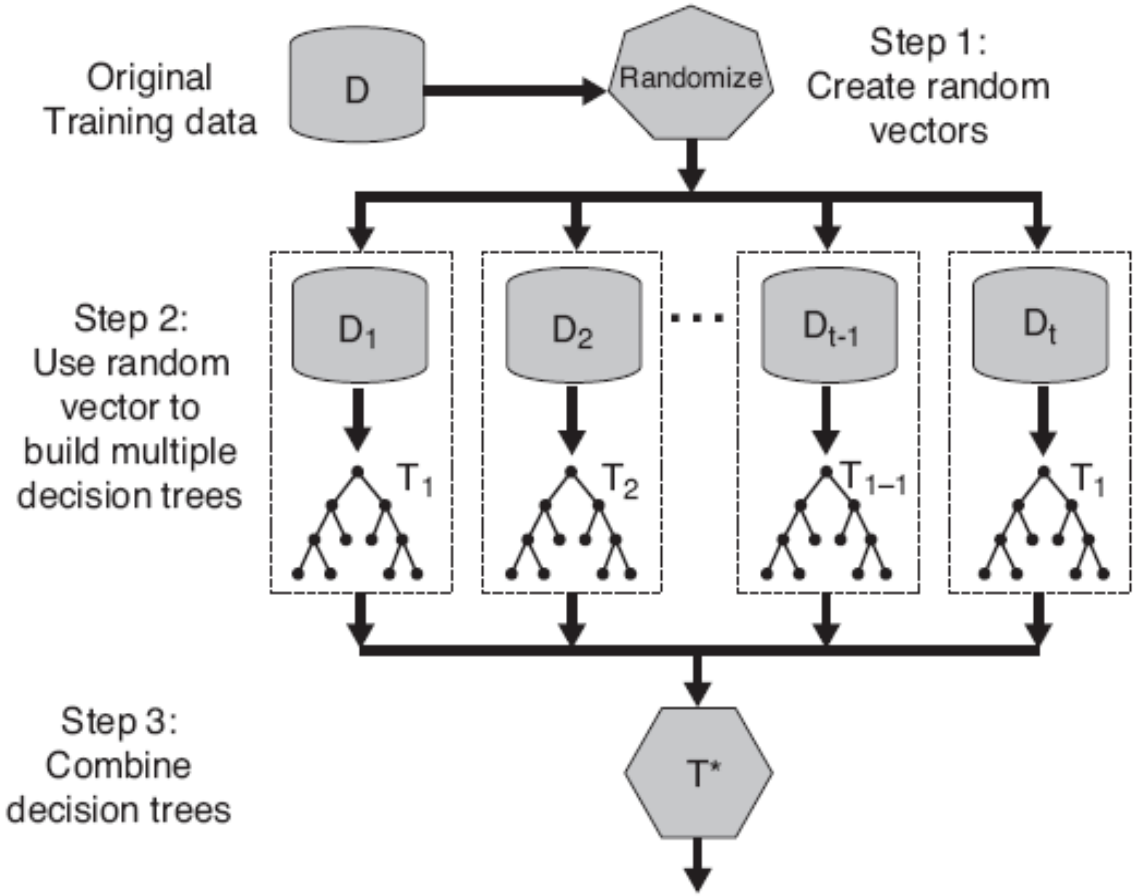
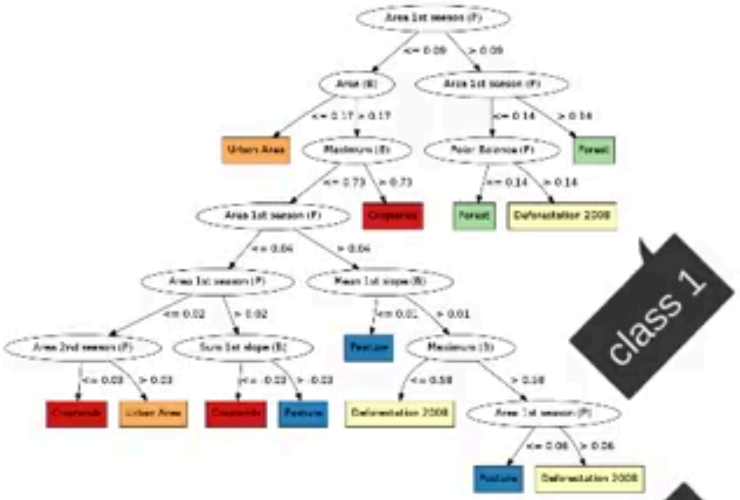


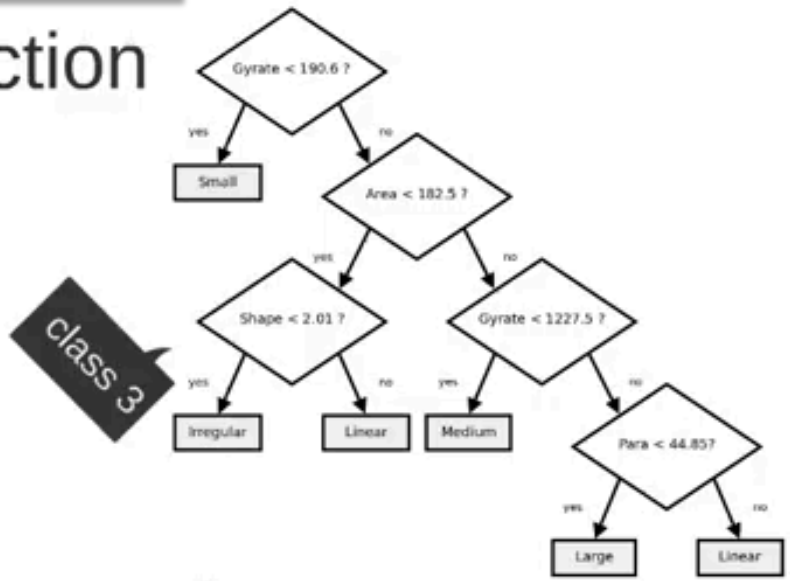
Figure 5.40. Random forests.

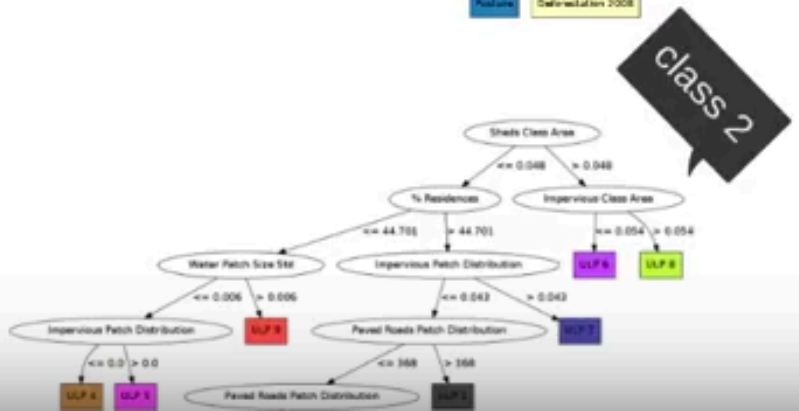
Example of random forest prediction

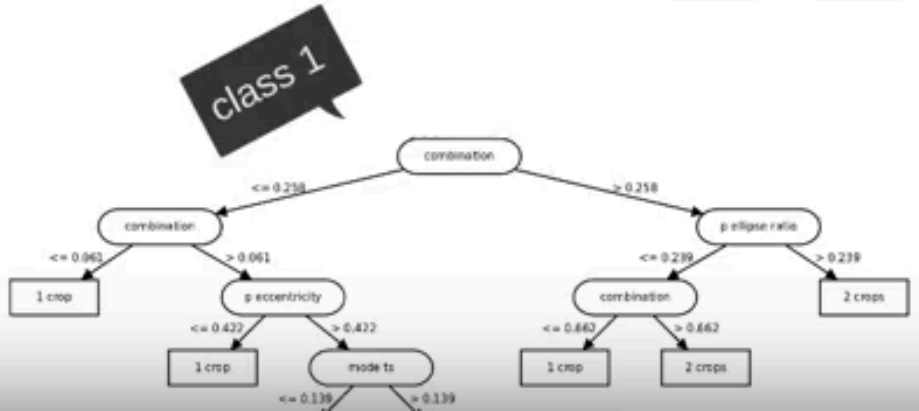
Class 2 is selected

Class prediction









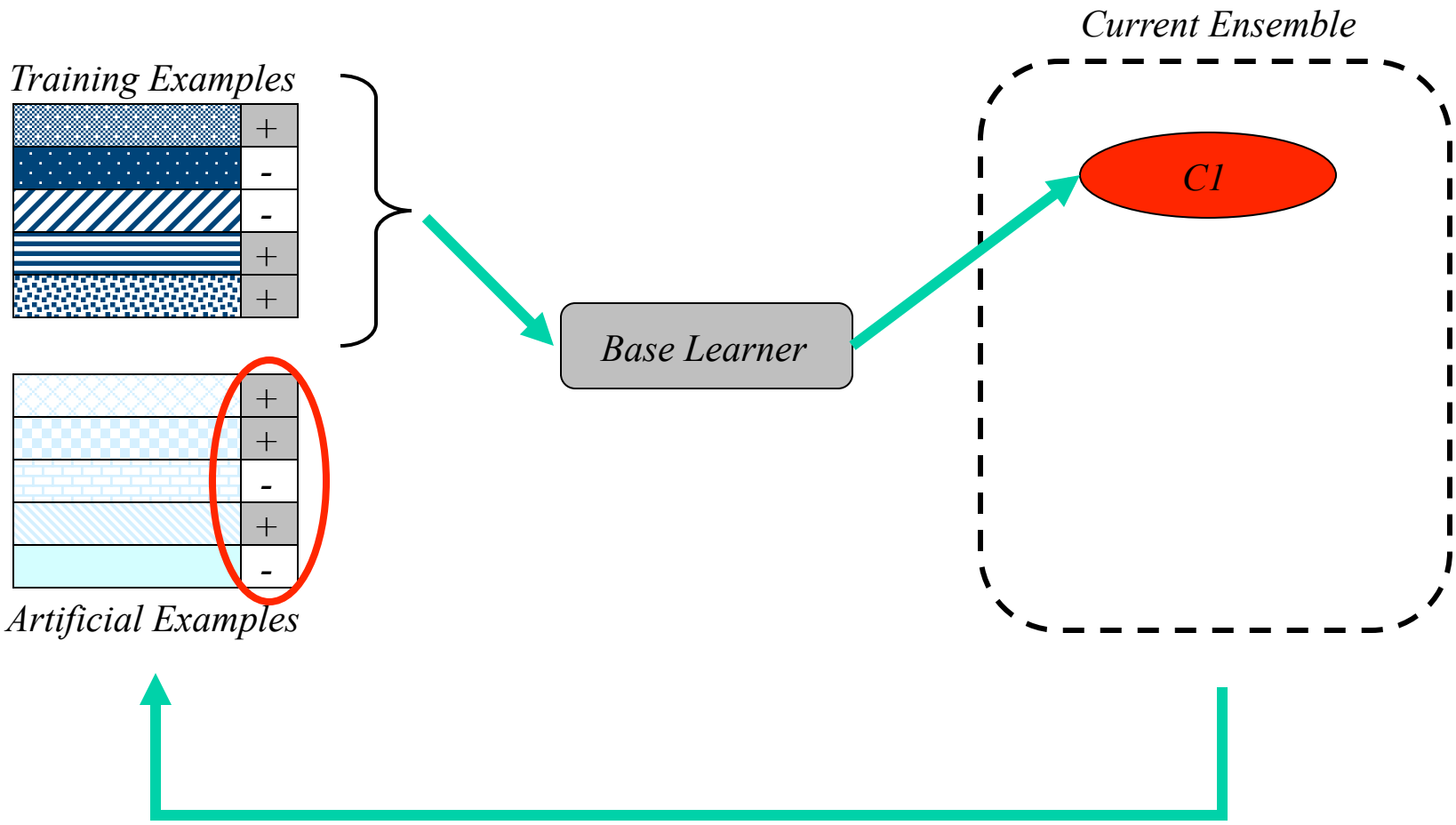
Advantage of Random Forest

- 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 considered a good “dimensionality reduction” method
- I does not do so well when features are continuous (regression trees)

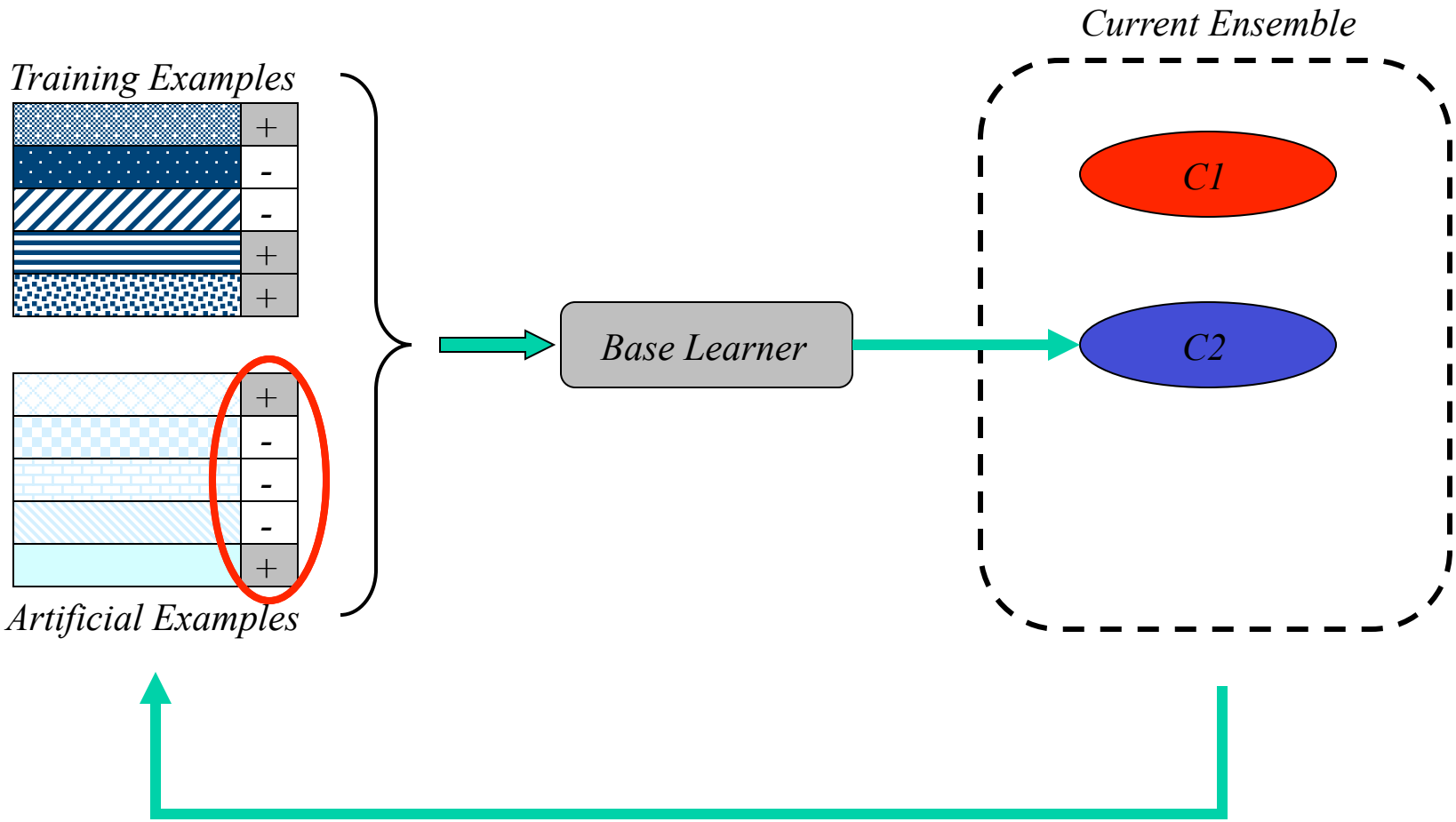
IV. DECORATE
(Melville & Mooney, 2003)

- Change training data by adding **new artificial training examples** that encourage diversity in the resulting ensemble.
- Improves accuracy when the training set is **small**, and therefore resampling and reweighting the training set would have limited ability to generate diverse alternative hypotheses.

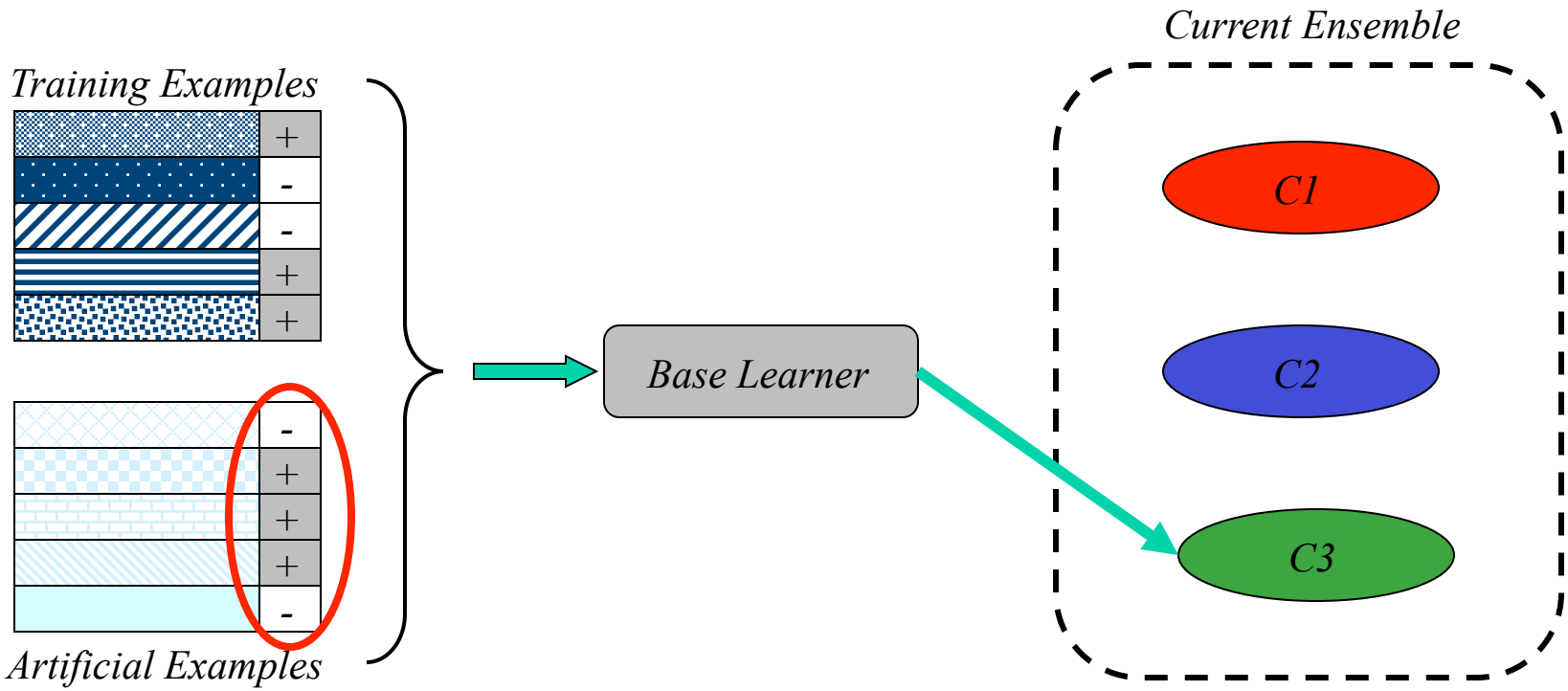
Overview of DECORATE



Overview of DECORATE



Overview of DECORATE



Creating artificial examples

- Create a set of new examples which are **maximally diverse** from training set. Let ***x*** be an (unclassified) example in this new set.
- To classify ***x***, proceed as follows:
  - Each base classifer, C_i, in the ensemble **C***, provides probabilities for the class membership of a sample **x**, i.e. $\hat{P}_{C_i,y}(x) = \hat{P}_{C_i}(C(x) = y)$
  - E.g., Naïve Bayes
  - The category label for x is selected according to:

$$C^*(x) = \arg\max_{y \in Y} \frac{\sum_{C_i \in C^*} \hat{P}_{C_i,y}(x)}{|C^*|}$$

Issues in Ensembles

- Parallelism in Ensembles: Bagging is easily parallelized, Boosting is not.
- Variants of Boosting to handle noisy data.
- How "weak" should a base-learner for Boosting be? (beyond the simple rule that error<50%)
- Exactly how does the diversity of ensembles affect their generalization performance.
- Combining Boosting and Bagging.