AI6102: Machine Learning Methodologies & Applications

L8b: Ensemble Learning

Sinno Jialin Pan

Nanyang Technological University, Singapore

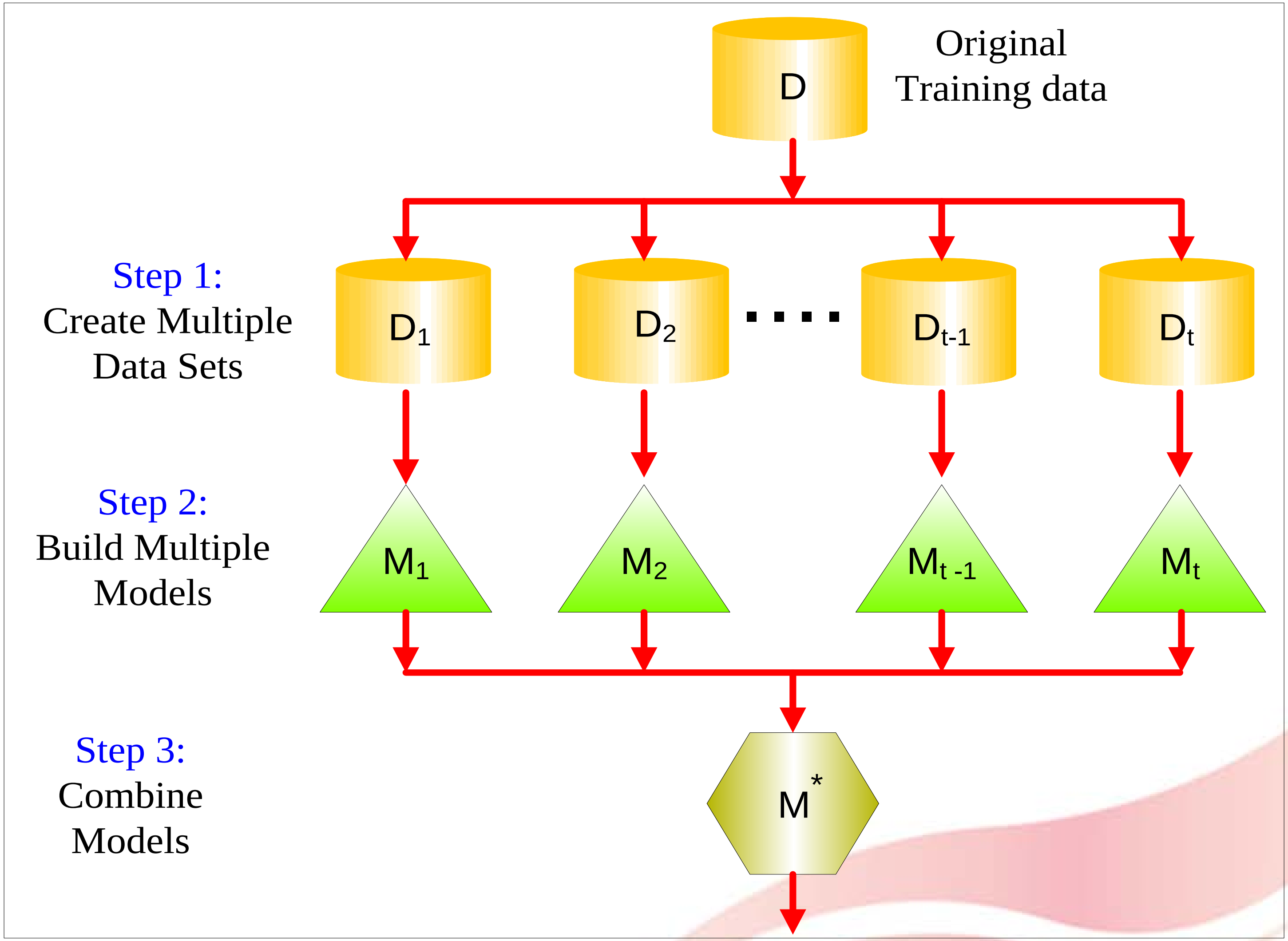
Homepage: http://www.ntu.edu.sg/home/sinnopan

Ensemble Learning

- Objective:
  - To improve model performance in terms of accuracy by aggregating the predictions of multiple models
- How to do it?
  - Construct a set of base models from the training data
  - Make predictions by combing the predicted results made by each base model

"Two heads are better than one"

High-level Idea



Stories of Success

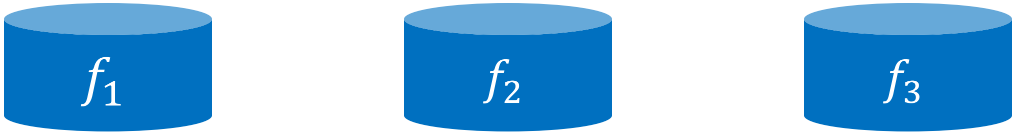
- Data mining competitions on Kaggle
  - Winning teams employ ensembles of classifiers

Feature Engineering → Ensemble Learning

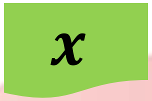
Why Ensemble Learning?

- Suppose there are 3 binary base classifiers
  - Each classifier has error rate, $\varepsilon = 0.35$ or accuracy $\text{acc} = 0.65$
  - Given a test data instance, if we choose any one of these classifiers to make prediction, the probability that the classifier makes a wrong prediction is 35%

Base classifiers:



A test data instance:



Why Ensemble Learning? (cont.)

- Consider to combine the 3 base classifiers to make a prediction on a test instance using a majority vote
- Assume classifiers are independent, then the ensemble makes a wrong prediction only if more than 1 out of the 3 base classifiers makes an incorrect prediction, i.e., either 2 or 3 base classifiers make incorrect predictions at the same time

Why Ensemble Learning? (cont.)

x	f_1	f_2	f_3	f_M	
Truth label: -1	+1	+1	+1	+1	✗
	+1	+1	-1	+1	✗
	+1	-1	+1	+1	✗
	-1	+1	+1	+1	✗
	+1	-1	-1	-1	✓
	-1	+1	-1	-1	✓
	-1	-1	+1	-1	✓
	-1	-1	-1	-1	✓

The combined model makes a wrong prediction if at least two out of the three base classifiers make a wrong prediction at the same time

Why Ensemble Learning? (cont.)

- Therefore, probability that the ensemble classifier makes a wrong prediction is:

$$\sum_{i=2}^{3}\binom{3}{i}\varepsilon^{i}(1-\varepsilon)^{3-i} = 3 \times 0.35^{2} \times 0.65 + 1 \times 0.35^{3} \times 1 = 0.2817$$

  - Case 1: when there are two exact classifiers make wrong predictions, the probability is

Two classifiers make wrong predictions

All possible combination → $\boxed{\binom{3}{2}}\boxed{\varepsilon^{2}}\boxed{(1-\varepsilon)^{3-2}}$ ← The rest one makes correct prediction

  - Case 2: when all the three classifiers make wrong predictions, the probability is

$$\binom{3}{3}\varepsilon^{3}(1-\varepsilon)^{3-3}$$

- That is the error rate of the ensemble classifier is 28.17%

$$\varepsilon_{f_i} = 35\% \longrightarrow \varepsilon_{M} = 28.17\%$$

Why Ensemble Learning? (cont.)

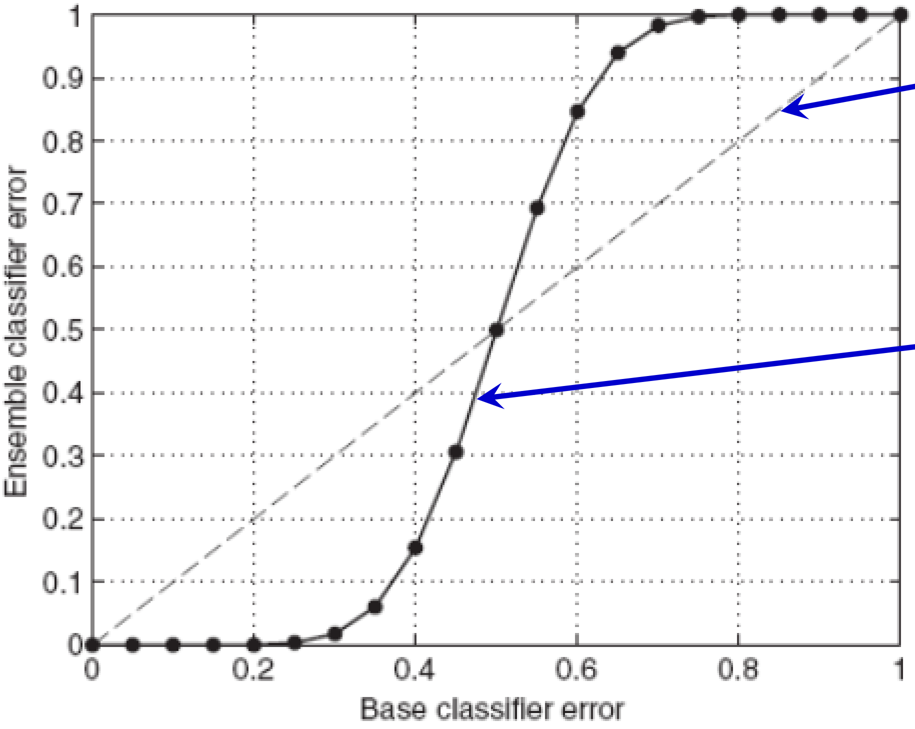
- Suppose there are 25 such independent base classifiers
  - Therefore, probability that the ensemble classifier makes a wrong prediction is:

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

  - That is the error rate of the ensemble classifier is 6%

$$\begin{array}{ll} \varepsilon_{f_i} = 35\% & \Bigg\downarrow \\ \varepsilon_M = 6\% & \end{array}$$

Necessary Conditions



The base classifiers are identical (perfectly correlated)

The base classifiers are independent

Error rate of an ensemble of 25 binary classifiers for different base classifier error rates

Observation: the ensemble classifier performs worse than the base classifiers when the base classifier error rate is larger than 0.5

Necessary Conditions (cont.)

- Two necessary conditions for an ensemble classifier to perform better than a single classifier:
  1. The base classifiers are independent of each other
  2. The base classifiers should do better than a classifier that performs random guessing (e.g., for binary classification, accuracy should be better than 0.5)

Independent Classifiers

- The base classifiers are independent of each other

f_1	f_2
$+1$	$+1$
-1	-1
-1	-1
$+1$	$+1$
$+1$	$+1$

Perfectly positively correlated

f_1	f_2
$+1$	-1
-1	$+1$
-1	$+1$
$+1$	-1
$+1$	-1

Perfectly negatively correlated

  - In practice, this condition can be relaxed that the base classifiers can be slightly correlated

Better than Random Guessing

- The base classifiers should do better than a classifier that performs random guessing (e.g., for binary classification, accuracy should be better than 0.5)

- Suppose there are N independent base classifiers, each of which has the same error rate ε
  - Therefore, probability that the ensemble classifier makes a wrong prediction is

There are i classifiers that make wrong predictions

$$P(N) = \sum_{\boxed{i=\lfloor N/2 \rfloor+1}}^{N} \boxed{\binom{N}{i}} \boxed{\varepsilon^{i}} \boxed{(1-\varepsilon)^{N-i}}$$

At least more than half of the base classifiers

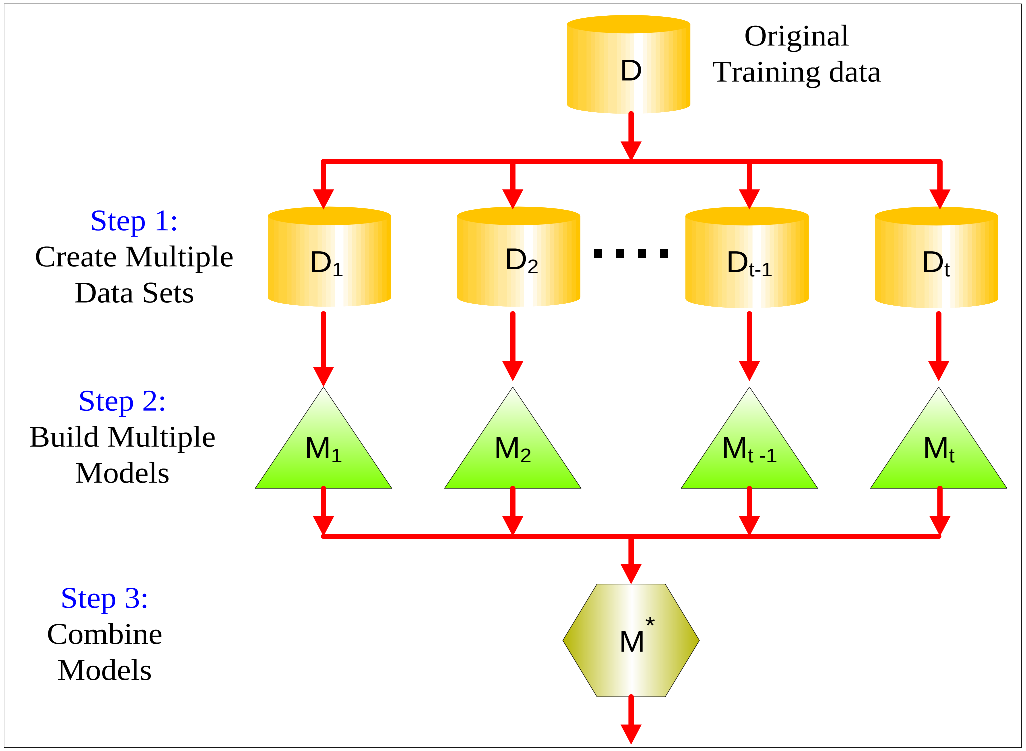
All possible combination

The rest $N-i$ classifiers making correct prediction

Better than Random Guess (cont.)

- It can be proved that
  - If $\varepsilon > 0.5$ then $P(N)$ is monotonically increasing in N, and $P(N) \rightarrow 1$ as $N \rightarrow \infty$
  - If $\varepsilon = 0.5$ then $P(N) = 0.5$ for all N
  - If $\varepsilon < 0.5$ then $P(N)$ is monotonically decreasing in N, and $P(N) \rightarrow 0$ as $N \rightarrow \infty$
- Detailed proof can be found in the paper "Application of Majority Voting to Pattern Recognition: An Analysis of Its Behavior and Performance, 1997"

Recall: High-level Idea



- How to generate a set of base classifiers?
- How to combine models?

General Methods

- To generate a set of base classifiers
  - By manipulating the training set: multiple training sets are created by resampling the training data according to some distribution. A classifier is then build from each training set
    - Bagging, Boosting
  - By manipulating the input features: a subset of input features is chosen to form a training set. A classifier is then build from each training set
    - Random forest
  - By manipulating the learning algorithm: applying the algorithm several times on the same training data using different hyper-parameters or applying different algorithms
- To combine classifiers?
  - Voting or weighted voting, etc.

A General Procedure

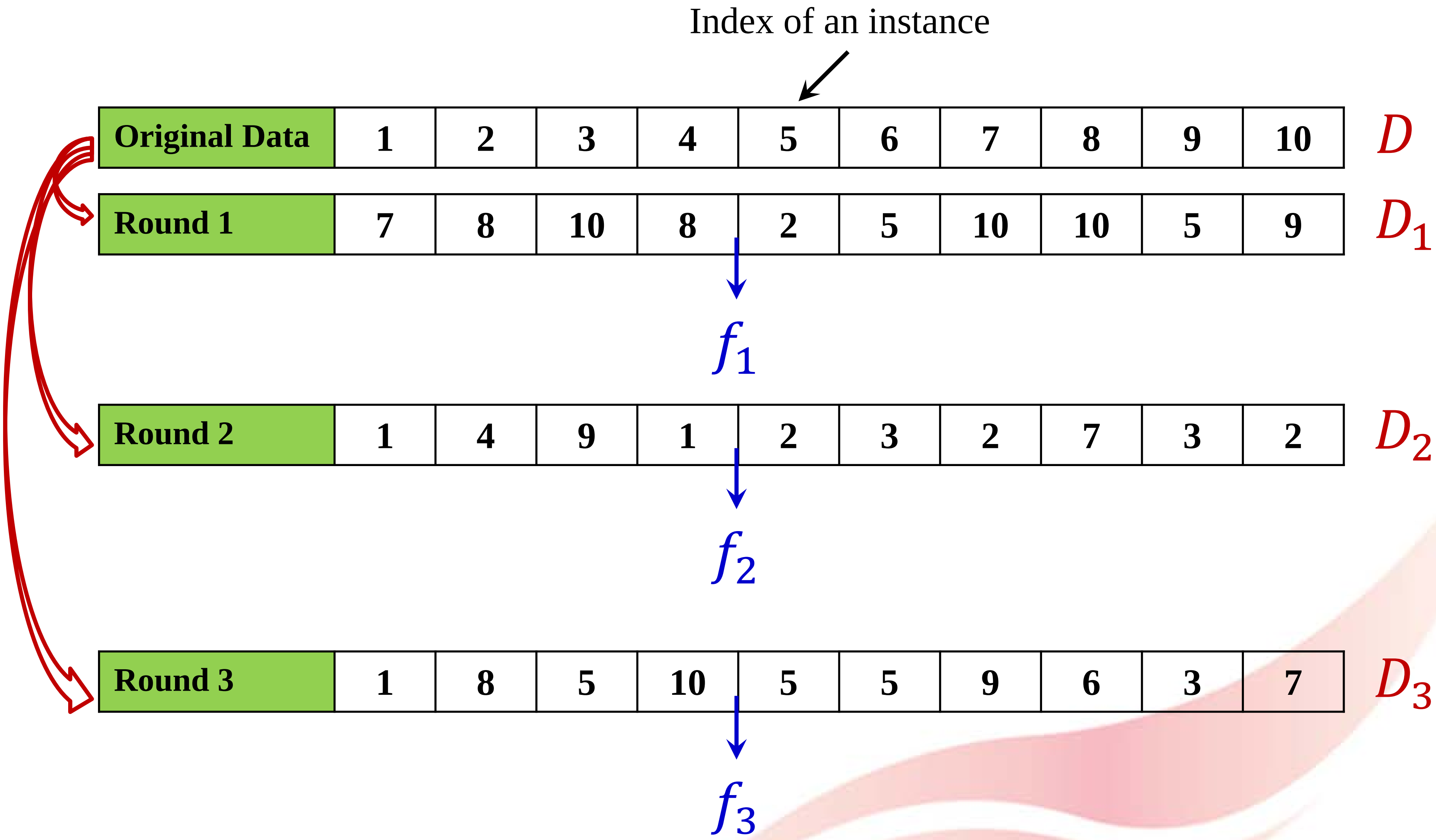
```
Let $D$ denote the original training data, $T$ denote the number of base classifiers, and $D^*$ be the test dataset.
for $i = 1$ to $T$ do
    Create training set $D_i$ from $D$
    Build a base classifier $f_i$ from $D_i$
end for
for each test record $\boldsymbol{x} \in D^*$ do
    Generate $f_1(\boldsymbol{x})$, $f_2(\boldsymbol{x})$, …, and $f_T(\boldsymbol{x})$
    Calculate $f_M(\boldsymbol{x}) = \boxed{\text{Combine}}(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_T(\boldsymbol{x}))$
end for
```

E.g., majority voting (can be other schemes)

Bagging

- Known as bootstrap aggregating, to repeatedly sample with replacement according to a uniform probability distribution
- Build classifier on each bootstrap sample, which is of the same size of the original data
- Use majority voting to determine the class label of ensemble classifier

Bagging (cont.)



Index of an instance

Original Data	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	D
Round 1	**7**	**8**	**10**	**8**	**2**	**5**	**10**	**10**	**5**	**9**	D_1
Round 2	**1**	**4**	**9**	**1**	**2**	**3**	**2**	**7**	**3**	**2**	D_2
Round 3	**1**	**8**	**5**	**10**	**5**	**5**	**9**	**6**	**3**	**7**	D_3

f_1

f_2

f_3

Bagging (cont.)

Index of an instance

Test Data	1	2	3	4	5	6	7	8	9	10
Round 1 (f_1)	+	+	+	+	-	-	-	+	-	-
Round 2 (f_2)	-	+	-	+	-	-	+	-	+	+
Round 3 (f_3)	+	-	-	+	+	-	-	+	+	-
Ensemble (f_M)	+	+	-	+	-	-	-	+	+	-

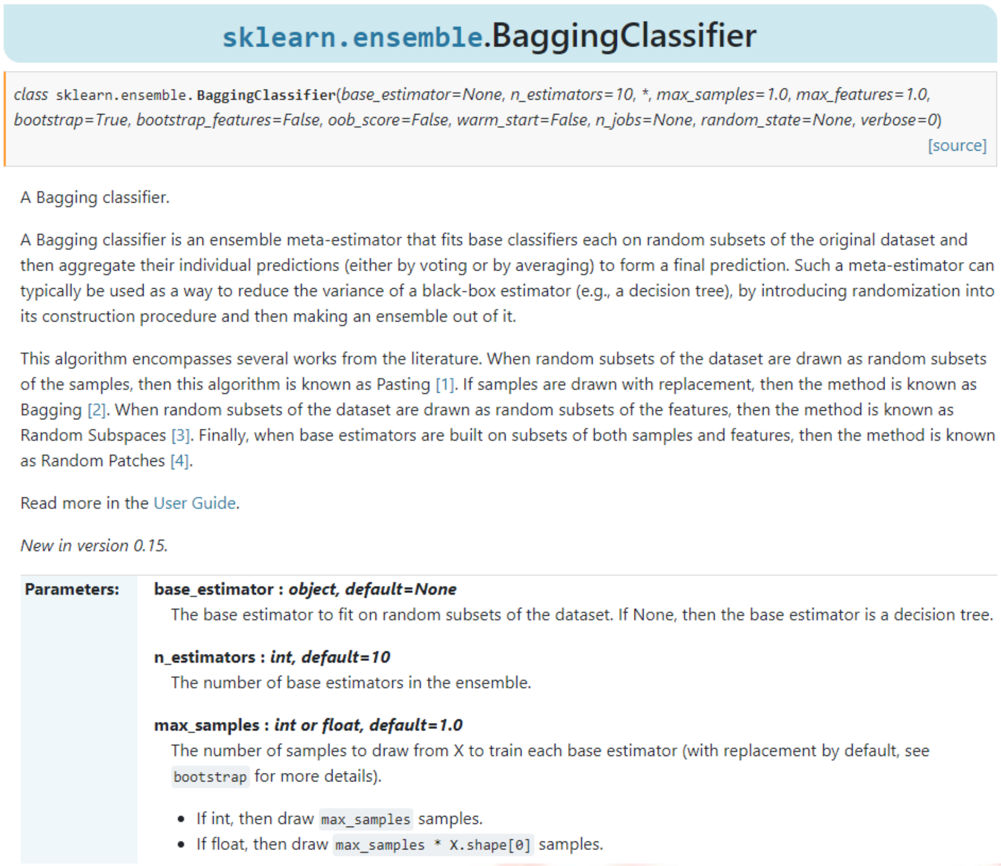
Majority Voting

Bagging (cont.)

- Suppose a training set D contains N examples
- A training example has a probability of $1-\frac{1}{N}$ of not being selected
- Its probability of ending up not in a training set D_i is $\left(1-\frac{1}{N}\right)^N \approx \frac{1}{e} = 0.368$
- A bootstrap sample D_i contains approximately 63.2% of the original training data

Implementation using scikit-learn

- API: sklearn.ensemble.BaggingClassifier
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html#sklearn.ensemble.BaggingClassifier



sklearn.ensemble.BaggingClassifier

class `sklearn.ensemble.`**`BaggingClassifier`**(*base_estimator=None, n_estimators=10, *, max_samples=1.0, max_features=1.0, bootstrap=True, bootstrap_features=False, oob_score=False, warm_start=False, n_jobs=None, random_state=None, verbose=0*)
[source]

A Bagging classifier.

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

This algorithm encompasses several works from the literature. When random subsets of the dataset are drawn as random subsets of the samples, then this algorithm is known as Pasting [1]. If samples are drawn with replacement, then the method is known as Bagging [2]. When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces [3]. Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches [4].

Read more in the User Guide.

New in version 0.15.

Parameters:

base_estimator : ***object, default=None***
The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.

n_estimators : ***int, default=10***
The number of base estimators in the ensemble.

max_samples : ***int or float, default=1.0***
The number of samples to draw from X to train each base estimator (with replacement by default, see `bootstrap` for more details).

- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples.

Example

classification algorithm used in bagging

```
>>> from sklearn.svm import SVC
>>> from sklearn.ensemble import BaggingClassifier
>>> import numpy as np

>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.integers(2, n_samples)
>>> X = rng.randn(n_samples, n_features)
```

specify how many base classifiers

```
>>> bagC = BaggingClassifier( base_estimator=SVC() n_estimators=10 )
>>> bagC.fit(X, y)
>>> pred= bagC.predict(X)
```

specify a base classification algorithm (usually decision tree is used)

Boosting

- Principles:
  - Boost a set of weak learners to a strong learner
  - Make data instances currently misclassified more important
  - To adaptively change the distribution of training data so that the base classifiers will focus more on previously misclassified data instances

Boosting: General Procedure

- Initially, all *N* data instances are assigned equal weights
  - Unlike bagging, weights may change at the end of each boosting round
- In each boosting round, after the weights are assigned to the training data instances, we can either
    - Draw a bootstrap sample from the original data by using the weights as a sampling distribution to build a model
    - Learn a model that is biased toward higher-weighted examples

Boosting: General Procedure

1. Initially, the data instances are assigned equal weights $\frac{1}{N}$, so that they are equally likely to be chosen for training. A sample is drawn uniformly to obtain a new training set.
2. A classifier is induced from the training set, and used to classify all the data instances in the original training set
3. The weights of the training instances are updated at the end of each boosting round
   - Instances that are wrongly classified will have their weights increased
   - Instances that are classified correctly will have their weights decreased
4. Repeat Step 2 and 3 until the stopping condition is met
5. Finally, a ensemble learner is obtained by aggregating the base classifiers obtained from each boosting round

Illustrative Example

Initially, all the examples are assigned the same weights.

	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	
Original Data	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	D
	+1	+1	+1	+1	-1	-1	-1	+1	+1	-1	

Uniformly randomly sample using boostrapping

Round 1	**7**	**3**	**2**	**8**	**7**	**9**	**4**	**10**	**6**	**3**	D_1

A classifier built from the data

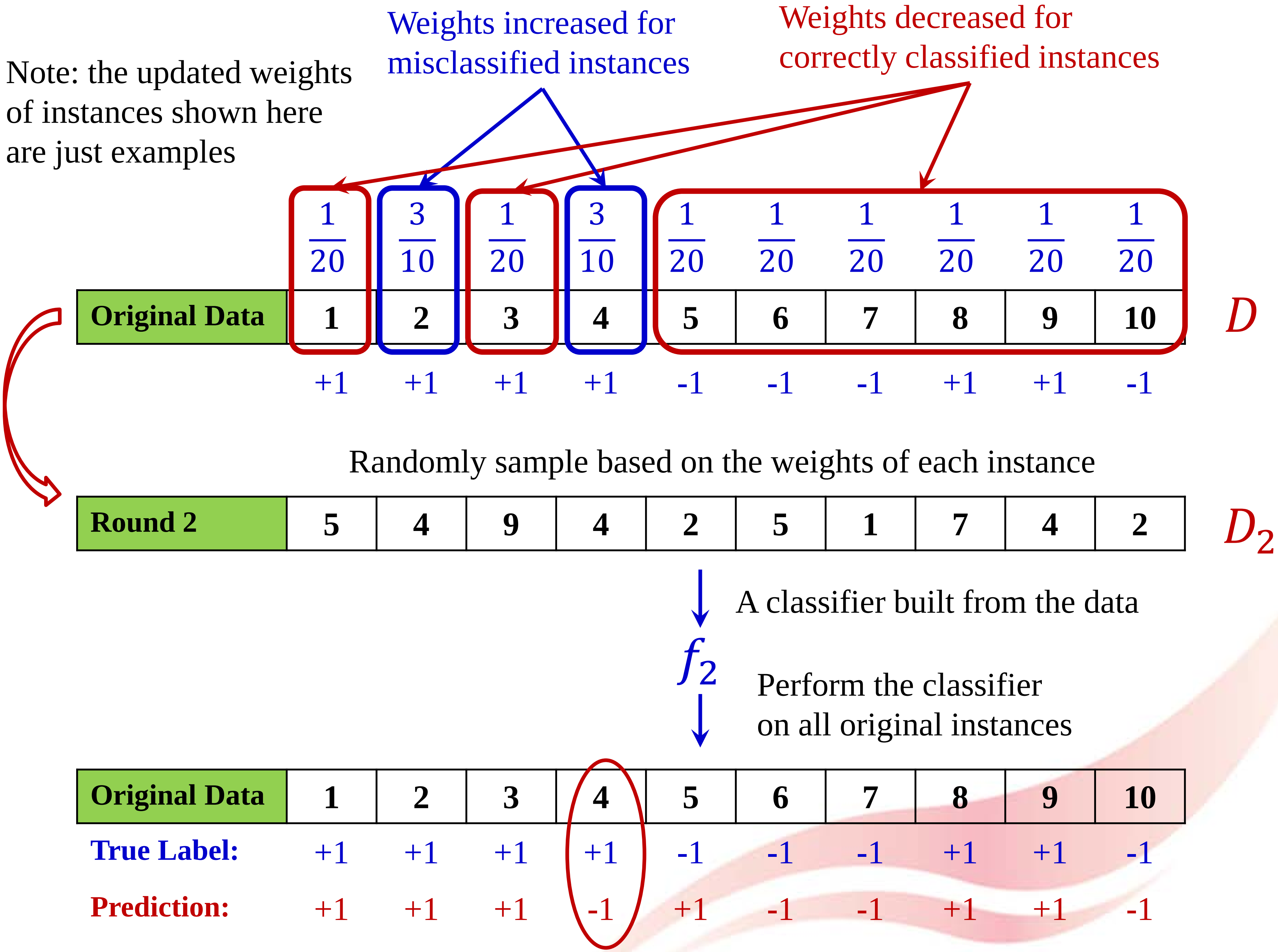
f_1

Perform the classifier on all the original instances

Misclassified

Original Data	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
True Label:	+1	+1	+1	+1	-1	-1	-1	+1	+1	-1
Prediction:	+1	-1	+1	-1	-1	-1	-1	+1	+1	-1

Note: the updated weights of instances shown here are just examples



Weights increased for misclassified instances

Weights decreased for correctly classified instances

	$\frac{1}{20}$	$\frac{3}{10}$	$\frac{1}{20}$	$\frac{3}{10}$	$\frac{1}{20}$	$\frac{1}{20}$	$\frac{1}{20}$	$\frac{1}{20}$	$\frac{1}{20}$	$\frac{1}{20}$	
Original Data	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	D
	+1	+1	+1	+1	-1	-1	-1	+1	+1	-1	

Randomly sample based on the weights of each instance

Round 2	**5**	**4**	**9**	**4**	**2**	**5**	**1**	**7**	**4**	**2**	D_2

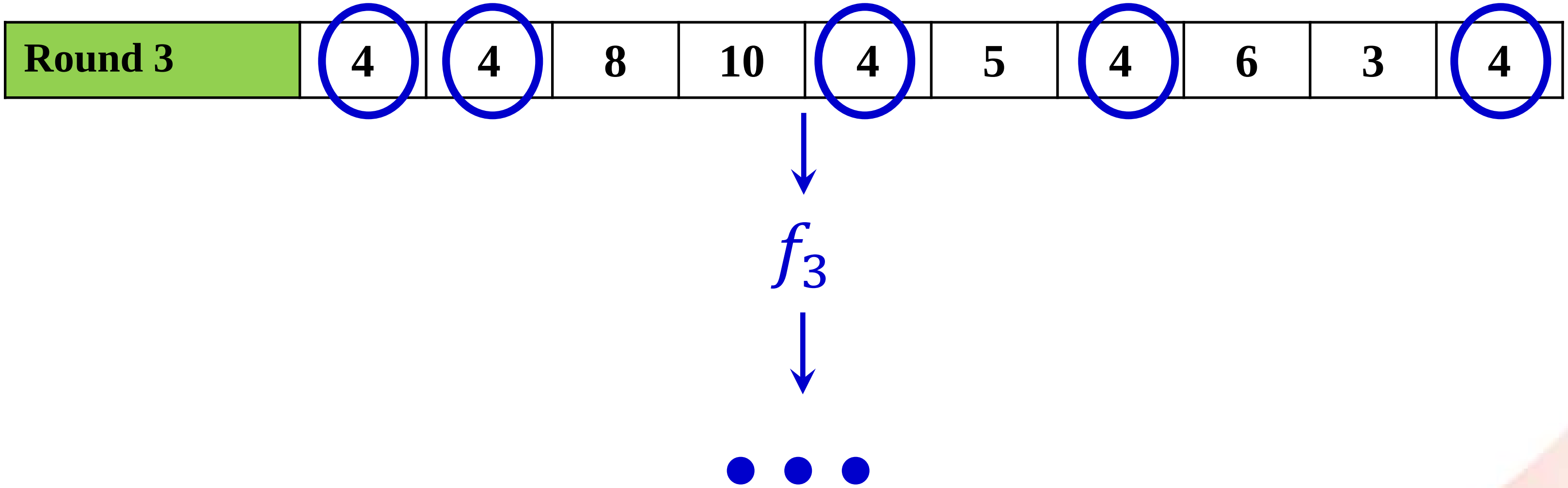
A classifier built from the data

f_2

Perform the classifier on all original instances

Original Data	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
True Label:	+1	+1	+1	+1	-1	-1	-1	+1	+1	-1
Prediction:	+1	+1	+1	-1	+1	-1	-1	+1	+1	-1

Randomly sample based on the updated weights of each instance



Round 3	**4**	**4**	**8**	**10**	**4**	**5**	**4**	**6**	**3**	**4**

f_3

● ● ●

As the boosting rounds proceed, examples that are the hardest to classify tend to become even more prevalent, e.g., instance 4

Testing

Index of an instance

Weights for each classifier

Test Data	1	2	3	4	5	6	7	8	9	10
Round 1 (f_1)	+	+	+	+	-	-	-	+	-	-
Round 2 (f_2)	-	+	-	+	-	-	+	-	+	+
Round 3 (f_3)	+	-	-	+	+	-	-	+	+	-
Ensemble (f_{M})	+2.2	-0.2	-3.8	+5.8	+0.2	-5.8	-2.2	+2.2	+3.8	-2.2
Sign	+	-	-	+	+	-	-	+	+	-

1

1.8

3

Weighted Voting. In binary classification $\{-1, +1\}$, it is equivalent to $\mathrm{sign}\big(\alpha_1 f_1(\boldsymbol{x}) + \alpha_2 f_2(\boldsymbol{x}) + \alpha_3 f_3(\boldsymbol{x})\big)$

Note: the weights of different classifiers shown here are just examples

AdaBoost (Adaptive Boosting)

- Given $\{\boldsymbol{x}_i, y_i\}, i = 1, \dots, N$, where $y_i \in \{-1, +1\}$, with initialized weights: $w_i^{(1)} = \frac{1}{N}$, and classification algorithm
- For $t = 1, \dots, T$
  - Train a classifier f_t, and compute its weighted error rate as
$$\epsilon_t = \sum_{\{f_t(\boldsymbol{x}_i) \neq y_i\}} w_i^{(t)}$$
  - Set $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$ if $\epsilon_t < \frac{1}{2}$ (better than random guessing)
  - Update the weights via
$$w_i^{(t+1)} = \frac{w_i^{(t)}}{\boxed{Z_t}} \times \begin{cases} e^{-\alpha_t} \text{ if } f_t(\boldsymbol{x}_i) = y_i \\ e^{\alpha_t} \text{ if } f_t(\boldsymbol{x}_i) \neq y_i \end{cases}$$

Normalization term to ensure $\sum_{i=1}^{N} w_i^{(t+1)} = 1$

- Final output of the boosted classifier:
$$F(\boldsymbol{x}^*) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t f_t(\boldsymbol{x})\right)$$

Discussion

- Regarding the weighted error rate $\epsilon_t = \sum_{\{f_t(\boldsymbol{x}_i)\neq y_i\}} w_i^{(t)}$

$$\text{error rate} = \frac{1}{N}\sum_{i=1}^{N} \boxed{I(f_t(\boldsymbol{x}_i) \neq y_i)} \leftarrow$$

Indicator function that returns 1 if its input is true, otherwise 0

$$0 \leq \epsilon_t = \sum_{i=1}^{N} w_i^{(t)} I(f_t(\boldsymbol{x}_i) \neq y_i) \;=\; \sum_{\{f_t(\boldsymbol{x}_i)\neq y_i\}} w_i^{(t)} \;\leq 1$$

- Regarding the weight of base classifier $\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
  - As $\epsilon_t < 1/2$, thus $\alpha_t > 0$, a better-performance base classifier (ϵ_t is smaller) has larger α_t

Discussion (cont.)

- Regarding the weight of data instance As $y_i \in \{-1, +1\}$

$$w_i^{(t+1)} = \frac{w_i^{(t)}}{Z_t} \times \begin{cases} e^{-\alpha_t} \text{ if } f_t(\boldsymbol{x}_i) = y_i \\ e^{\alpha_t} \text{ if } f_t(\boldsymbol{x}_i) \neq y_i \end{cases} = \frac{w_i^{(t)}}{Z_t} \times e^{-\alpha_t \times f_t(\boldsymbol{x}_i) y_i}$$

  - As $\alpha_t > 0$, thus $e^{-\alpha_t} < 1$, and $e^{\alpha_t} > 1$
  - The normalization term $Z_t = \sum_{j=1}^{N} w_j^{(t)} e^{-\alpha_t \times f_t(\boldsymbol{x}_j) y_j}$

$$\sum_{i=1}^{N} w_i^{(t+1)} = \sum_{i=1}^{N} \frac{w_i^{(t)} e^{-\alpha_t \times f_t(\boldsymbol{x}_i) y_i}}{\sum_{j=1}^{N} w_j^{(t)} e^{-\alpha_t \times f_t(\boldsymbol{x}_j) y_j}} = \frac{\sum_{j=1}^{N} w_i^{(t)} e^{-\alpha_t \times f_t(\boldsymbol{x}_i) y_i}}{\sum_{j=1}^{N} w_j^{(t)} e^{-\alpha_t \times f_t(\boldsymbol{x}_j) y_j}} = 1$$

- Further readings:
  - A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting, by Freund and Schapire
  - Multi-class AdaBoost, by Zhu, Zou, Rosset and Hastie

Boosting as Gradient Descent

- Given $\{\boldsymbol{x}_i, y_i\}, i = 1, \ldots, N$, where $y_i \in \{-1, +1\}$, the goal of boosting is to adaptively generate a set of T (weak) classifiers $f_t, t = 1, \ldots, T$, and combine them to construct a boosted (strong) classifier

$$F(\boldsymbol{x}) = \sum_{t=1}^{T} \alpha_t f_t(\boldsymbol{x}), \text{and } y = \text{sign}(F(\boldsymbol{x}))$$

- Suppose after $t - 1$ iterations, a boosted classifier has been constructed

$$F_{t-1}(\boldsymbol{x}) = \alpha_1 f_1(\boldsymbol{x}) + \cdots + \alpha_{t-1} f_{t-1}(\boldsymbol{x})$$

- At iteration t, the goal is to boost $F_{t-1}(\boldsymbol{x})$ to a better classifier via

$$F_t(\boldsymbol{x}) = F_{t-1}(\boldsymbol{x}) + \alpha_t f_t(\boldsymbol{x})$$

- How to learn $f_t(\boldsymbol{x})$ and determine its weight α_t?
  - The performance of $F_{t-1}(\boldsymbol{x})$ is measure by the difference or loss between $F_{t-1}(\boldsymbol{x}_i)$ and y_i: $\ell\big(y_i, F_{t-1}(\boldsymbol{x}_i)\big)$

Boosting as Gradient Descent (cont.)

- Boosting can be considered to find $\{F(\boldsymbol{x}_i)\}$'s iteratively to minimize $\ell\big(y_i, F(\boldsymbol{x}_i)\big)$

$$\begin{matrix} F_1(\boldsymbol{x}_1) \\ \cdots \\ F_1(\boldsymbol{x}_N) \end{matrix} \longrightarrow \cdots \longrightarrow \boxed{\begin{matrix} F_{t-1}(\boldsymbol{x}_1) \\ \cdots \\ F_{t-1}(\boldsymbol{x}_N) \end{matrix}} \longrightarrow \begin{matrix} F_t(\boldsymbol{x}_1) \\ \cdots \\ F_t(\boldsymbol{x}_N) \end{matrix} \longrightarrow \cdots \longrightarrow \begin{matrix} y_1 \\ \cdots \\ y_N \end{matrix}$$

consider as a N dimensional vector

Gradient descent: $$F_t(\boldsymbol{x}_i) = F_{t-1}(\boldsymbol{x}_i) - \rho \frac{\partial \ell\big(y_i, F_{t-1}(\boldsymbol{x}_i)\big)}{\partial F_{t-1}(\boldsymbol{x}_i)}$$

Boosting: $$F_t(\boldsymbol{x}_i) = F_{t-1}(\boldsymbol{x}_i) + \alpha_t f_t(\boldsymbol{x}_i)$$

Consider the negative gradient as "ground-truth" for each $f(\boldsymbol{x}_i)$

$$f_t^*(\boldsymbol{x}) = \arg\min_f \mathcal{L}\left(-\frac{\partial \ell\big(y_i, F_{t-1}(\boldsymbol{x}_i)\big)}{\partial F_{t-1}(\boldsymbol{x}_i)}, f(\boldsymbol{x}_i)\right)$$

Boosting as Gradient Descent (cont.)

Boosting: $F_t(\boldsymbol{x}_i) = F_{t-1}(\boldsymbol{x}_i) + \alpha_t f_t(\boldsymbol{x}_i)$

After $f_t^*(\boldsymbol{x}) = \arg\min_f \mathcal{L}\left(-\frac{\partial \ell(y_i, F_{t-1}(\boldsymbol{x}_i))}{\partial F_{t-1}(\boldsymbol{x}_i)}, f(\boldsymbol{x}_i)\right)$ is learned

$$\alpha_t^* = \arg\min_\alpha \ell(y_i, F_{t-1}(\boldsymbol{x}_i) + \alpha f_t(\boldsymbol{x}_i))$$

This is a one-dimensional optimization problem w.r.t. α

$$F_t(\boldsymbol{x}) = F_{t-1}(\boldsymbol{x}) + \alpha_t^* f_t^*(\boldsymbol{x})$$

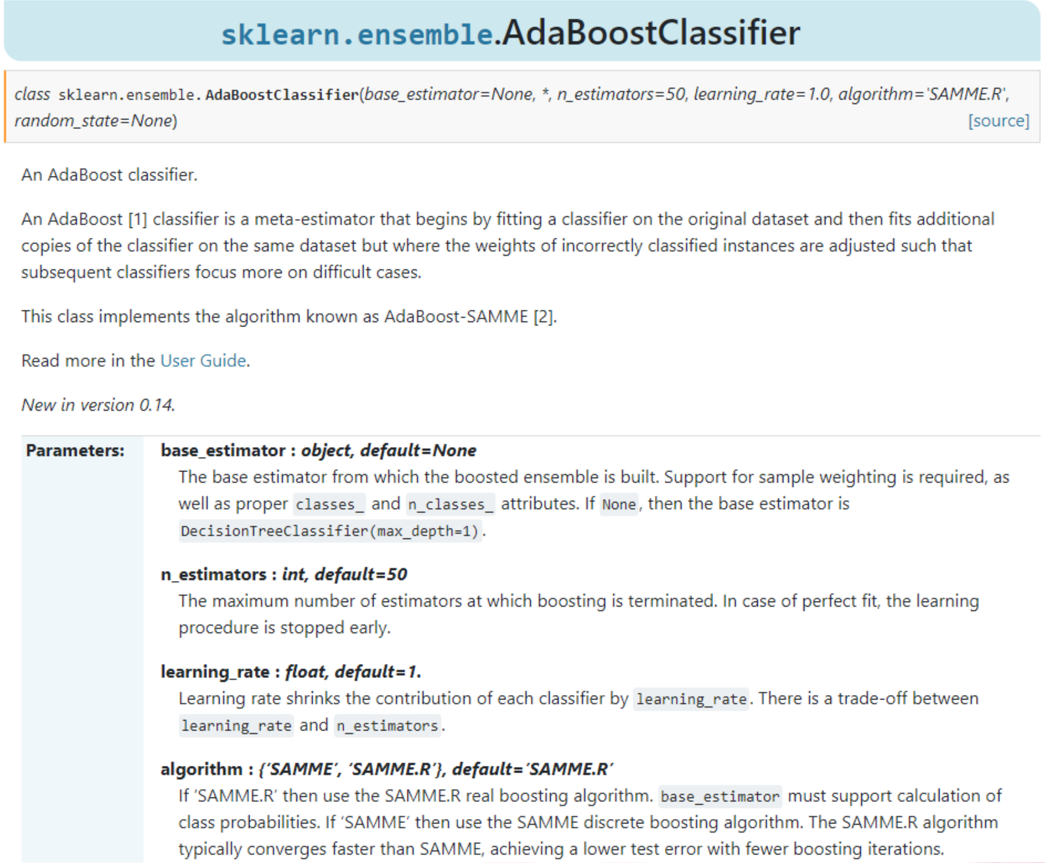
This is also known as Gradient Boosting Machines

Further readings:

Greedy Function Approximation: A Gradient Booting Machine, by Friedman

Implementation using scikit-learn

- API: sklearn.ensemble.AdaBoostClassifier
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier



sklearn.ensemble.AdaBoostClassifier

class `sklearn.ensemble.AdaBoostClassifier`(*base_estimator=None, *, n_estimators=50, learning_rate=1.0, algorithm='SAMME.R', random_state=None*) [source]

An AdaBoost classifier.

An AdaBoost [1] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

This class implements the algorithm known as AdaBoost-SAMME [2].

Read more in the User Guide.

New in version 0.14.

Parameters:

base_estimator : *object, default=None*
The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper `classes_` and `n_classes_` attributes. If `None`, then the base estimator is `DecisionTreeClassifier(max_depth=1)`.

n_estimators : *int, default=50*
The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

learning_rate : *float, default=1.*
Learning rate shrinks the contribution of each classifier by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

algorithm : *{'SAMME', 'SAMME.R'}, default='SAMME.R'*
If 'SAMME.R' then use the SAMME.R real boosting algorithm. `base_estimator` must support calculation of class probabilities. If 'SAMME' then use the SAMME discrete boosting algorithm. The SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations.

Example

classification algorithm used in AdaBoost

```
>>> from sklearn.svm import SVC
>>> from sklearn.ensemble import AdaBoostClassifier
>>> import numpy as np

>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.integers(2, n_samples)
>>> X = rng.randn(n_samples, n_features)
```

specify how many base classifiers

```
>>> adaC = AdaBoostClassifier (base_estimator=SVC(), n_estimators=10 )
>>> adaC.fit(X, y)
>>> pred= adaC.predict(X)
```

specify a base classification algorithm
(usually decision tree is used)

Random Forests

- A class of ensemble methods specifically designed for decision tree classifiers
- Random Forests grows many trees
- Each tree is generated based on a random subset of features
- Final result on classifying a new instance: voting. Forest chooses the classification result having the most votes (over all the trees in the forest)

Random Forests (cont.)

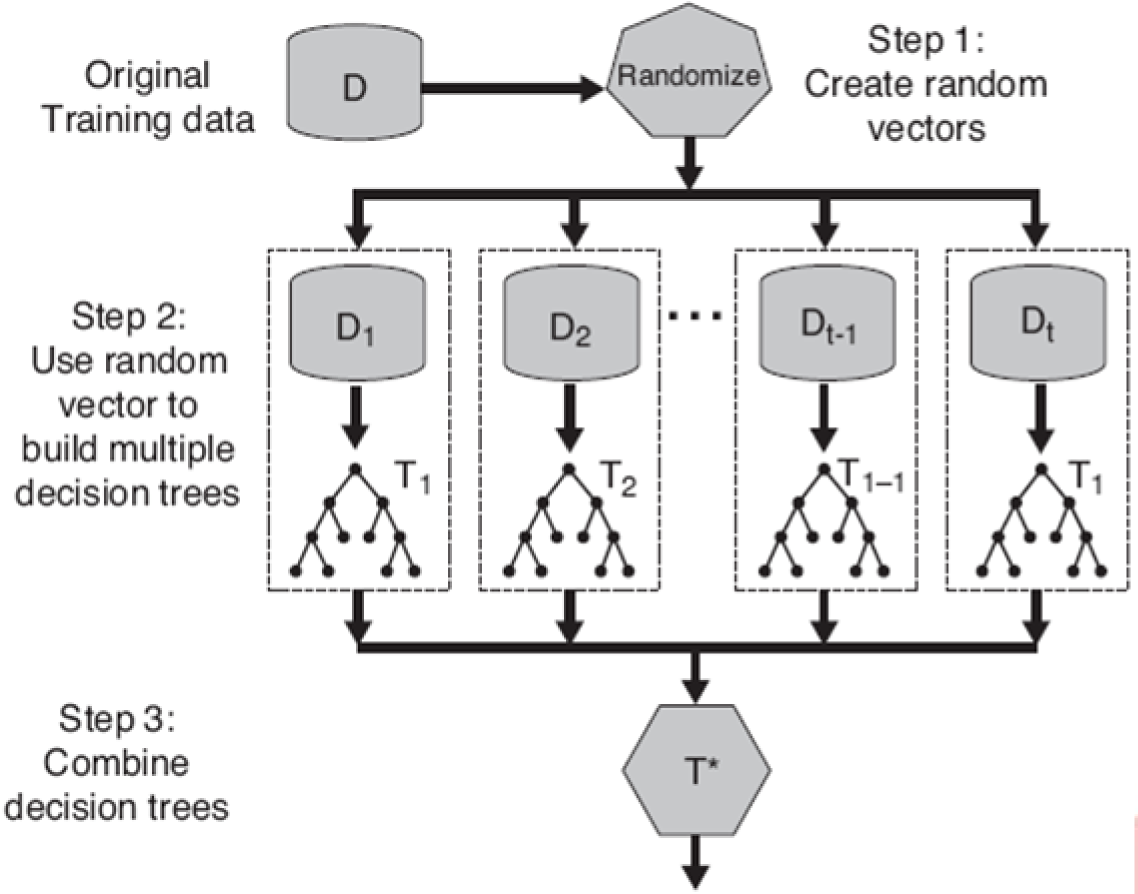


Illustration of random forests

Random Forests: General Algorithm

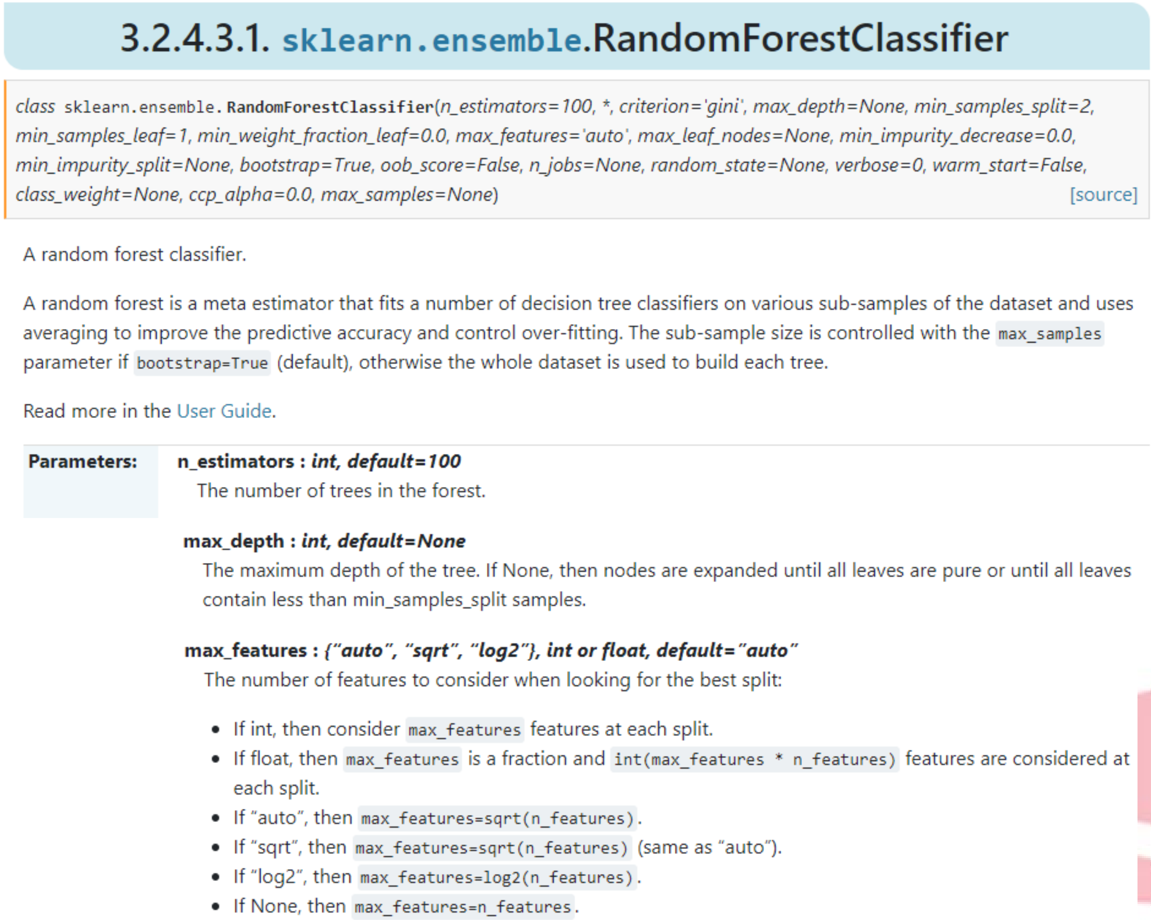
- Choose T: number of trees to grow
- Choose $d < m$ (m is the number of total features): number of features used to calculate the best split at each node (typically 20%)
- For each tree
  - Choose a training set boostrapping
  - For each node, randomly choose d features and calculate the best split
  - Fully grown and not pruned
- Use majority vote among all the trees

Random Forests: Discussion

- Bagging + random features
- Improve Accuracy
  - Incorporate more diversity
- Improve Efficiency
  - Searching among subsets of features is much faster than searching among the complete set

Implementation using scikit-learn

- API: sklearn.ensemble.RandomForestClassifier
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier



3.2.4.3.1. sklearn.ensemble.RandomForestClassifier

class sklearn.ensemble.**RandomForestClassifier**(*n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None*) [source]

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the `max_samples` parameter if `bootstrap=True` (default), otherwise the whole dataset is used to build each tree.

Read more in the User Guide.

Parameters:

n_estimators : *int, default=100*

The number of trees in the forest.

max_depth : *int, default=None*

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

max_features : *{"auto", "sqrt", "log2"}, int or float, default="auto"*

The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)` (same as "auto").
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Example

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> import numpy as np

>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.integers(2, n_samples)
>>> X = rng.randn(n_samples, n_features)

>>> rfC = RandomForestClassifier(n_estimators = 20, max_depth = 2)
>>> rfC.fit(X, y)
>>> pred= rfC.predict(X)
```

Combining Classifiers

- Average
  - Simple average
  - Weighted average
- Voting
  - Majority voting
  - More restrictive mojority voting
  - Weighted voting
- Combining by learning

Average

- Simple average:

$$f_M(\boldsymbol{x}) = \frac{1}{T}\sum_{i=1}^{T} f_i(\boldsymbol{x})$$

- Weighted average:

$$f_M(\boldsymbol{x}) = \sum_{i=1}^{T} w_i f_i(\boldsymbol{x})$$

$$\text{where } w_i \geq 0, \text{and } \sum_{i=1}^{T} w_i = 1$$

Voting

- Majority voting:
  - Every classifier votes for one class and takes the class label that receives the largest number of votes as the final winner
- More restrictive majority voting:
  - Final output class label is the one that receives more than half of votes
  - If none of the class labels receives more than half of votes, a rejection option will be given (fails to make a prediction)
- Weighted voting:
  - By introducing weights for each classifier

Learning of Combination

- Stacking:
  - A general procedure where a learner is trained to combine the individual learners
  - Individual learners: first-level learners
  - Combiner: second-level learner, or meta-learner

Stacking: An Example

- Consider a binary classification problem

m input features

ground-truth y_i

N training instances

$\boldsymbol{x}_1$	3	1	…	0.3	+1
$\boldsymbol{x}_2$	4	1.1	…	0.6	−1
…	…	…	…	…	…
$\boldsymbol{x}_N$	−10	0	…	0.2	+1

k base classifiers are trained

$f_1 \quad f_2 \quad \cdots \quad f_k$

Stacking: An Example (cont.)

predicted value of classifier f_1 on the instance $\boldsymbol{x}_1$

k base classifiers

ground-truth y_i

N data instances

	f_1	f_2	…	f_k	
$\boldsymbol{x}_1$	0.6	1	…	0.3	$+1$
$\boldsymbol{x}_2$	0.3	1	…	0.6	-1
…	…	…	…	…	…
$\boldsymbol{x}_N$	0.1	-1	…	0.2	$+1$

A meta classifier is learned, $f_M\colon \mathbb{R}^k \longrightarrow \mathbb{R}$

	f_1	f_2	…	f_k
$\boldsymbol{x}^*$	0.8	-1	…	0.7

prediction

$f_M(\boldsymbol{x}^*)$

Thank you!