AA 2022/2023

Machine Learning for Modelling: *Supervised Learning*

Simone Bianco

1

AA 2022/2023

Ensemble Methods

2

Ensemble methods

Ensemble methods train multiple learners to solve the same problem.

In contrast to ordinary learning approaches which try to construct one learner from training data, ensemble methods try to construct a set of learners and combine them.

Ensemble learning is also called **committee-based learning** or learning **multiple classifier systems**.

An ensemble contains a number of learners called base learners. Base learners are usually generated from training data by a **base learning algorithm** which can be decision tree, neural network or other kinds of learning algorithms.

3

Ensemble methods

Most ensemble methods use a single base learning algorithm to produce homogeneous base learners, i.e., learners of the same type, leading to **homogeneous ensembles**, but there are also some methods which use multiple learning algorithms to produce heterogeneous learners, i.e., learners of different types, leading to **heterogeneous ensembles**.

In the latter case there is no single base learning algorithm and thus, some people prefer calling the learners **individual learners** or **component learners**.

4

Ensemble methods

The generalization ability of an ensemble is often much stronger than that of base learners.

Ensemble methods are appealing mainly because they are able to boost **weak learners** which are even just slightly better than random guess to **strong learners** which can make very accurate predictions. So, *base learners* are also referred to as *weak learners*.

There are 3 threads of early contributions that led to the current area of ensemble methods:

- **Combining classifiers**: mostly studied in PR. Working on strong classifiers and try to design powerful combining rules to get stronger combined classifiers.
- **Ensembles of weak learners**: mostly studied in ML. Working on weak learners and try to design powerful algorithms to boost the performance from weak to strong.
- **Mixture of experts**: mostly studied in NN. Generally considering a divide-and-conquer strategy, try to learn a mixture of parametric models jointly and use combining rules to get an overall solution.

5

AA 2022/2023

Boosting

7

A general boosting procedure

Boosting refers to a family of algorithms that can convert weak learners to strong learners.

Intuitively, a weak learner is just slightly better than random guess, while a strong learner is very close to perfect performance.

The birth of boosting algorithms originated from the answer to an interesting theoretical question posed by Kearns and Valiant [1989]. That is, whether two complexity classes, weakly learnable and strongly learnable problems, are equal.

This question is of fundamental importance, since if the answer is positive, any weak learner is potentially able to be boosted to a strong learner, particularly if we note that in real practice it is generally very easy to obtain weak learners but difficult to get strong learners.

Schapire in 1990 proved that the answer is positive, and the proof is a construction, i.e., boosting.

8

A general boosting procedure

The general boosting procedure is quite simple.

Boosting works by training a set of learners sequentially and combining them for prediction, where the later learners focus more on the mistakes of the earlier learners.

Input: Sample distribution $\mathcal{D}$;
Base learning algorithm $\mathfrak{L}$;
Number of learning rounds T.

Process:

1. $\mathcal{D}_1 = \mathcal{D}$. % Initialize distribution
2. **for** $t = 1, \ldots, T$:
3. $h_t = \mathfrak{L}(\mathcal{D}_t)$; % Train a weak learner from distribution $\mathcal{D}_t$
4. $\epsilon_t = P_{\boldsymbol{x} \sim \mathcal{D}_t}(h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}))$; % Evaluate the error of h_t
5. $\mathcal{D}_{t+1} = Adjust\_Distribution(\mathcal{D}_t, \epsilon_t)$
6. **end**

Output: $H(\boldsymbol{x}) = Combine\_Outputs(\{h_1(\boldsymbol{x}), \ldots, h_t(\boldsymbol{x})\})$

9

AdaBoost algorithm

AdaBoost instantiates the *Adjust_Distribution()* and *Combine_Output()* parts of the previous general boosting procedure.

Consider binary classification on classes {+1, -1}. One version of derivation of AdaBoost is achieved by minimizing the exponential loss function

$$\ell_{exp}(h|\mathcal{D}) = \mathbb{E}_{x\sim\mathcal{D}}\left[e^{-f(x)h(x)}\right]$$

Using additive weighted combination of weak learners as

$$H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$$

The exponential loss is used since it gives an elegant and simple update formula, and it is consistent with the goal of minimizing classification error.

10

AdaBoost algorithm

When the exponential loss is minimized by H, the partial derivative of the exponential loss for every *x* has to be zero:

$$\frac{\partial e^{-f(x)H(x)}}{\partial H(x)} = -f(x)e^{-f(x)H(x)} = -e^{-H(x)}P(f(x)=1|x) + e^{H(x)}P(f(x)=-1|x) = 0$$

Solving the right part, we have

$$H(x) = \frac{1}{2}\ln\frac{P(f(x)=1|x)}{P(f(x)=-1|x)}$$

and hence

$$\begin{aligned}\operatorname{sign}(H(\boldsymbol{x})) &= \operatorname{sign}\left(\frac{1}{2}\ln\frac{P(f(x)=1\mid\boldsymbol{x})}{P(f(x)=-1\mid\boldsymbol{x})}\right)\\ &= \begin{cases}1, & P(f(x)=1\mid\boldsymbol{x}) > P(f(x)=-1\mid\boldsymbol{x})\\ -1, & P(f(x)=1\mid\boldsymbol{x}) < P(f(x)=-1\mid\boldsymbol{x})\end{cases}\\ &= \underset{y\in\{-1,1\}}{\arg\max} P(f(x)=y\mid\boldsymbol{x}),\end{aligned}$$

11

AdaBoost algorithm

The H is produced by iteratively generating h_t and α_t.

The first weak classifier h_1 is generated by training the weak learning algorithm on the original distribution.

When a classifier h_t is generated under the distribution $\mathcal{D}_t$, its weight α_t is to be determined such that $\alpha_t h_t$ minimizes the exponential loss

$$\begin{aligned}\ell_{\exp}(\alpha_t h_t \mid \mathcal{D}_t) &= \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}_t}[e^{-f(\boldsymbol{x})\alpha_t h_t(\boldsymbol{x})}]\\ &= \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}_t}\left[e^{-\alpha_t}\mathbb{I}(f(\boldsymbol{x})=h_t(\boldsymbol{x})) + e^{\alpha_t}\mathbb{I}(f(\boldsymbol{x})\neq h_t(\boldsymbol{x}))\right]\\ &= e^{-\alpha_t}P_{\boldsymbol{x}\sim\mathcal{D}_t}(f(\boldsymbol{x})=h_t(\boldsymbol{x})) + e^{\alpha_t}P_{\boldsymbol{x}\sim\mathcal{D}_t}(f(\boldsymbol{x})\neq h_t(\boldsymbol{x}))\\ &= e^{-\alpha_t}(1-\epsilon_t) + e^{\alpha_t}\epsilon_t\,,\end{aligned}$$

To get the optimal α_t let the derivative of the exponential loss equal zero:

$$\frac{\partial\ell_{\exp}(\alpha_t h_t \mid \mathcal{D}_t)}{\partial\alpha_t} = -e^{-\alpha_t}(1-\epsilon_t) + e^{\alpha_t}\epsilon_t = 0$$

Then the solution is:

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

12

AdaBoost algorithm

Once a sequence of weak classifiers and their corresponding weights have been generated, these classifiers are combined as H_{t-1}.

AdaBoost then adjusts the sample distribution such that in the next round, the base learning algorithm will output a weak classifier h_t that corrects some mistakes of H_{t-1}.

The ideal classifier h_t that corrects all mistakes of H_{t-1} should minimize the exponential loss

$$\begin{aligned}\ell_{\exp}(H_{t-1}+h_t \mid \mathcal{D}) &= \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[e^{-f(\boldsymbol{x})(H_{t-1}(\boldsymbol{x})+h_t(\boldsymbol{x}))}]\\ &= \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}e^{-f(\boldsymbol{x})h_t(\boldsymbol{x})}]\,.\end{aligned}$$

Using Taylor expansion of $e^{-f(x)h_t(x)}$, we can approximate the exponential loss as

$$\begin{aligned}\ell_{\exp}(H_{t-1}+h_t \mid \mathcal{D}) &\approx \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\left[e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}\left(1-f(\boldsymbol{x})h_t(\boldsymbol{x}) + \frac{f(\boldsymbol{x})^2 h_t(\boldsymbol{x})^2}{2}\right)\right]\\ &= \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\left[e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}\left(1-f(\boldsymbol{x})h_t(\boldsymbol{x}) + \frac{1}{2}\right)\right]\end{aligned}$$

13

AdaBoost algorithm

Thus, the ideal classifier h_t is

$$\begin{aligned} h_t(\boldsymbol{x}) &= \arg\min_h \ell_{\exp}(H_{t-1} + h \mid \mathcal{D}) \\ &= \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\left[e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}\left(1 - f(\boldsymbol{x})h(\boldsymbol{x}) + \frac{1}{2}\right)\right] \\ &= \arg\max_h \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\left[e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})} f(\boldsymbol{x})h(\boldsymbol{x})\right] \\ &= \arg\max_h \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\left[\frac{e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}}{\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}]} f(\boldsymbol{x})h(\boldsymbol{x})\right], \end{aligned}$$

Let's denote a distribution $\mathcal{D}_t$ as

$$\mathcal{D}_t(x) = \frac{\mathcal{D}(x)e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}}{\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}]} .$$

14

AdaBoost algorithm

Then we can rewrite the ideal classifier h_t as

$$\begin{aligned} h_t(\boldsymbol{x}) &= \arg\max_h \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\left[\frac{e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}}{\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}]} f(\boldsymbol{x})h(\boldsymbol{x})\right] \\ &= \arg\max_h \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}_t}\left[f(\boldsymbol{x})h(\boldsymbol{x})\right] . \end{aligned}$$

Further noticing that $f(\boldsymbol{x})h_t(\boldsymbol{x}) = 1 - 2\mathbb{I}(f(\boldsymbol{x}) \neq h_t(\boldsymbol{x}))$, the ideal classifier is

$$h_t(\boldsymbol{x}) = \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}_t}\left[\mathbb{I}(f(\boldsymbol{x}) \neq h(\boldsymbol{x}))\right] .$$

i.e., the ideal classifier minimizes the error under the distribution $\mathcal{D}_t$.

15

AdaBoost algorithm

Considering the relation between $\mathcal{D}_t$ and $\mathcal{D}_{t+1}$ we have

$$\begin{aligned} \mathcal{D}_{t+1}(\boldsymbol{x}) &= \frac{\mathcal{D}(x)e^{-f(\boldsymbol{x})H_t(\boldsymbol{x})}}{\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[e^{-f(\boldsymbol{x})H_t(\boldsymbol{x})}]} \\ &= \frac{\mathcal{D}(\boldsymbol{x})e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}e^{-f(\boldsymbol{x})\alpha_t h_t(\boldsymbol{x})}}{\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[e^{-f(\boldsymbol{x})H_t(\boldsymbol{x})}]} \\ &= \mathcal{D}_t(\boldsymbol{x}) \cdot e^{-f(\boldsymbol{x})\alpha_t h_t(\boldsymbol{x})} \frac{\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[e^{-f(\boldsymbol{x})H_{t-1}(\boldsymbol{x})}]}{\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[e^{-f(\boldsymbol{x})H_t(\boldsymbol{x})}]} , \end{aligned}$$

which is the way AdaBoost updates the sample distribution.

16

AdaBoost algorithm

Pseudo-code of the complete AdaBoost algorithm

Input: Data set $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
Base learning algorithm $\mathfrak{L}$;
Number of learning rounds T.

Process:
1. $\mathcal{D}_1(\boldsymbol{x}) = 1/m$. % Initialize the weight distribution
2. **for** $t = 1, \ldots, T$:
3. $h_t = \mathfrak{L}(D, \mathcal{D}_t)$; % Train a classifier h_t from D under distribution $\mathcal{D}_t$
4. $\epsilon_t = P_{\boldsymbol{x}\sim\mathcal{D}_t}(h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}))$; % Evaluate the error of h_t
5. **if** $\epsilon_t > 0.5$ **then break**
6. $\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$; % Determine the weight of h_t
7. $\mathcal{D}_{t+1}(\boldsymbol{x}) = \frac{\mathcal{D}_t(\boldsymbol{x})}{Z_t} \times \begin{cases} \exp(-\alpha_t) & \text{if } h_t(\boldsymbol{x}) = f(\boldsymbol{x}) \\ \exp(\alpha_t) & \text{if } h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}) \end{cases}$
$= \frac{\mathcal{D}_t(\boldsymbol{x})\exp(-\alpha_t f(\boldsymbol{x})h_t(\boldsymbol{x}))}{Z_t}$ % Update the distribution, where
% Z_t is a normalization factor which
% enables $\mathcal{D}_{t+1}$ to be a distribution
8. **end**

Output: $H(\boldsymbol{x}) = \mathtt{sign}\left(\sum_{t=1}^{T}\alpha_t h_t(\boldsymbol{x})\right)$

17

AdaBoost algorithm

Note that AdaBoost algorithm requires the base learning algorithm to be able to learn with specified distribution. This is often accomplished by **re-weighting**: weighting training examples in each round according to the sample distribution.

For base learning algorithms that cannot handle weighted training examples, it is accomplished by **re-sampling**: sampling training examples in each round according to the desired distribution.

For base learning algorithms which can be used with both re-weighting and re-sampling, generally there is no clear performance difference between these two implementations.

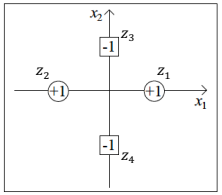
However, re-sampling provides an option for **Boosting with restart**. In each round of AdaBoost, there is a sanity check to ensure that the current base learner is better than random guess.

18

AdaBoost algorithm: an example

Consider an artificial data set in a 2-d space with only 4 instances:

$$\begin{cases} (\boldsymbol{z}_1 = (+1, 0), y_1 = +1) \\ (\boldsymbol{z}_2 = (-1, 0), y_2 = +1) \\ (\boldsymbol{z}_3 = (0, +1), y_3 = -1) \\ (\boldsymbol{z}_4 = (0, -1), y_4 = -1) \end{cases}$$



Where $y_i = f(z_i)$ is the label of each instance. It is the XOR problem, and there is no straight line able to separate the two classes (i.e., the classes cannot be separated by a linear classifier).

Suppose that we have the following base learning algorithm:

- It evaluates 8 basis functions $h_1, \cdots, h_8$ on the training data under the given distribution, and returns the one with the smallest error
- If more than one basis function have the same smallest error, it selects one of them randomly

19

AdaBoost algorithm: an example

The 8 basis functions:

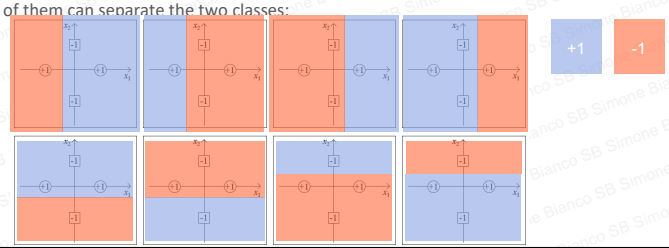
$$h_1(\boldsymbol{x}) = \begin{cases} +1, \text{ if } (x_1 > -0.5) \\ -1, \quad \text{otherwise} \end{cases} \quad h_2(\boldsymbol{x}) = \begin{cases} -1, \text{ if } (x_1 > -0.5) \\ +1, \quad \text{otherwise} \end{cases}$$

$$h_3(\boldsymbol{x}) = \begin{cases} +1, \text{ if } (x_1 > +0.5) \\ -1, \quad \text{otherwise} \end{cases} \quad h_4(\boldsymbol{x}) = \begin{cases} -1, \text{ if } (x_1 > +0.5) \\ +1, \quad \text{otherwise} \end{cases}$$

$$h_5(\boldsymbol{x}) = \begin{cases} +1, \text{ if } (x_2 > -0.5) \\ -1, \quad \text{otherwise} \end{cases} \quad h_6(\boldsymbol{x}) = \begin{cases} -1, \text{ if } (x_2 > -0.5) \\ +1, \quad \text{otherwise} \end{cases}$$

$$h_7(\boldsymbol{x}) = \begin{cases} +1, \text{ if } (x_2 > +0.5) \\ -1, \quad \text{otherwise} \end{cases} \quad h_8(\boldsymbol{x}) = \begin{cases} -1, \text{ if } (x_2 > +0.5) \\ +1, \quad \text{otherwise} \end{cases}$$

Notice that none of them can separate the two classes:

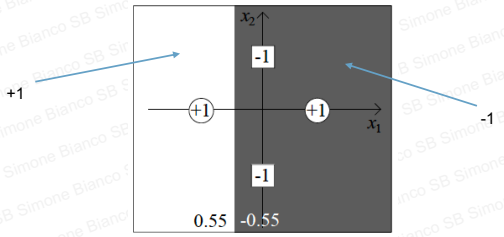


20

AdaBoost algorithm: an example

We run the AdaBoost algorithm:

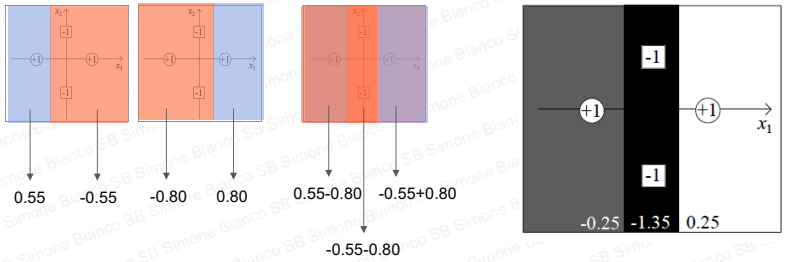
1. Invoke the base learning algorithm on the original data. The smallest classification (0.25) is reached by h_2, h_3, h_5, h_8. Suppose h_2 is returned.
   One instance is wrongly classified (z_1), so the error is ¼=0.25.
   The weight of h_2 is therefore $0.5 \ln {}^{1-0.25}/_{0.25} \approx 0.55$



21

AdaBoost algorithm: an example

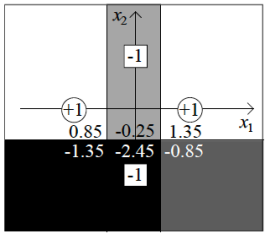
2. The weight of the wrongly classified sample z_1 is increased.
   The base learning algorithm is invoked again.
   This time h_3, h_5, h_8 have the smallest errors. Suppose h_3 is returned.
   One instance is wrongly classified (z_2), and its re-weighted error is $^{0.577}/_{0.577*3+2.887} \approx 0.167$
   Its weight is therefore $0.5 \ln {}^{1-0.167}/_{0.167} \approx 0.80$



22

AdaBoost algorithm: an example

3. The weight of the wrongly classified sample z_2 is increased.
   The base learning algorithm is invoked again.
   This time h_5, h_8 have the smallest errors. Suppose h_5 is returned.
   One instance is wrongly classified (z_3), and its re-weighted error is $^{0.259}/_{0.778+1.287+2*0.259} \approx 0.100$
   Its weight is therefore $0.5 \ln {}^{1-0.100}/_{0.100} \approx 1.10$



23

AdaBoost algorithm: an example

After 3 steps let us consider the sign of classification weights in each area.

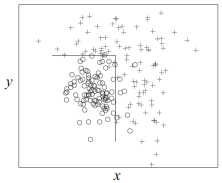
It can be observed that the signs correctly predicts the labels of the 4 instances.

Thus, by combining the imperfect linear classifiers, AdaBoost has produced a non-linear classifier with zero error.

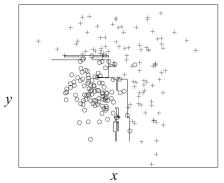
24

AdaBoost algorithm: an example

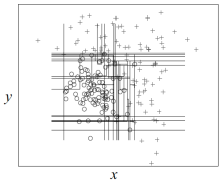
As a further example we run AdaBoost on the three-Gaussians dataset.



Decision boundary of a single decision tree



Decision boundary of AdaBoost



Decision boundary of the 10 trees used by AdaBoost

25

Multiclass AdaBoost algorithm

So far, we focused on AdaBoost for binary classification, i.e., $\mathcal{Y} = \{+1, -1\}$. In many classification tasks an instance may belong to one of **many** classes.

There are many alternative ways to extend AdaBoost for multiclass classification.

AdaBoost.M1 is a very straightforward extension, where the base learners are now multiclass learners instead for binary classifiers.

SAMME is an improvement over AdaBoost.M1 that changes the way in which the weight α_t is computed:

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) + \ln(|\mathcal{Y}| - 1).$$

26

Multiclass AdaBoost algorithm

A commonly used solution to the multiclass classification problem is to decompose the task into multiple binary classification problems. Popular decomposition schemes include:

- **one-versus-rest** (a.k.a. one-versus-all): decomposes a multiclass task of $|\mathcal{Y}|$ classes into $|\mathcal{Y}|$ binary classification tasks, where the i-th task is to classify if an instance belongs to the i-th class or not.
- **one-versus-one**: decomposes into $\frac{|\mathcal{Y}|(|\mathcal{Y}|-1)}{2}$ binary classification tasks, where each task is to classify if an instance belongs to the i-th class or the j-th class.

27

Multiclass AdaBoost algorithm

AdaBoost.MH follows the one-versus-rest strategy. After training $|\mathcal{Y}|$ number of AdaBoost classifiers, the real value output $H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$ rather than the crisp classification of each AdaBoost classifier is used to identify the most probable class, i.e.:

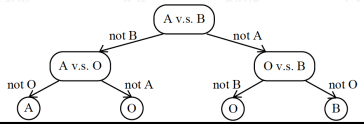
$$H(x) = \arg\max_{y \in \mathcal{Y}} H_y(x)$$

with H_y being the binary AdaBoost classifier that classifies the y-th class from the rest.

AdaBoost.M2 follows the one-versus-one strategy, which minimizes a pseudo-loss.

It has been later generalized as AdaBoost.MR which minimizes a ranking loss motivated by the fact that the highest ranked class is more likely to be the correct class.

Binary classifiers obtained by one-vs-one decomposition can be aggregated by voting, pairwise coupling, directed acyclic graph.



28

AdaBoost algorithm for regression

AdaBoost can be modified to solve regression tasks.

An AdaBoost regressor is a meta-estimator that begins by fitting a regressor on the original dataset and then fits additional copies of the regressor on the same dataset but where the weights of the instances are adjusted according to the error of the current prediction. As such, subsequent regressors focus more on difficult cases.

Popular AdaBoost algorithms for regression are AdaBoost.R2, AdaBoost.RT, etc.

29

Bagging

31

Ensemble paradigms

According to how the base learners are generated, there are two paradigms of ensemble methods:

- **Sequential ensemble methods**, where the base learners are generated sequentially (e.g., AdaBoost)
- **Parallel ensemble methods**, where the base learners are generated in parallel (e.g., Bagging).

Sequential methods exploit the dependence between the base learners, since the overall performance can be boosted in a residual-decreasing way.

Parallel ensemble methods exploit the independence between the base learners, since the error can be reduced dramatically by combining independent base learners.

32

Ensemble paradigms

Let's take the binary classification as an example, on classes $\{+1, -1\}$. Suppose the ground truth function is f and each base classifier has an independent generalization error ϵ, i.e., for the base classifier h_i:

$$P(h_i(x) \neq f(x)) = \epsilon$$

After combining T of such base classifiers according to

$$H(x) = \text{sign}\left(\sum_{i=1}^{T} h_i(x)\right)$$

The ensemble H makes an error only when at least half of its base classifier makes errors.

Therefore, by Hoeffding inequality, the generalization error of the ensemble is

$$P(H(\boldsymbol{x}) \neq f(\boldsymbol{x})) = \sum_{k=0}^{\lfloor T/2 \rfloor} \binom{T}{k} (1-\epsilon)^k \epsilon^{T-k} \leq \exp\left(-\frac{1}{2} T (2\epsilon - 1)^2\right)$$

that clearly decreases exponentially wrt the ensemble size T, and approaches to zero as $T \to \infty$

33

Ensemble paradigms

Though it is practically impossible to get really independent base learners since they are generated from the same training data set, base learners with less dependence can be obtained by introducing randomness in the learning process, and a good generalization ability can be expected by the ensemble.

Another benefit of the parallel ensemble methods is that they are inherently favorable to parallel computing, and the training speed can be easily accelerated using multi-core computing processors or parallel computers. This is attractive as multi-core processors are commonly available nowadays.

34

The Bagging Algorithm

The name **Bagging** came from the abbreviation of **Bootstrap AGGregatING**: as the name implies, the two ingredients of Bagging are **bootstrap** and **aggregation**.

We know that the combination of independent base learners will lead to a dramatic decrease of errors and therefore, we want to get base learners as independent as possible.

Given a training data set, one possibility seems to be sampling a number of non-overlapped data subsets and then training a base learner from each of the subsets. However, since we do not have infinite training data, such a process will produce very small and unrepresentative samples, leading to poor performance of base learners.

35

The Bagging Algorithm

Bagging adopts the **bootstrap distribution** for generating different base learners. I.e., it applies bootstrap sampling to obtain the data subsets for training the base learners.

In detail, given a training data set containing a number m of training examples, a sample of m training examples will be generated by sampling with replacement.

Some original examples will appear more than once, while some original examples will not be present in the sample.

By applying the process T times, T samples of m training examples are obtained.

Then, from each sample a base learner can be trained by applying the base learning algorithm.

36

The Bagging Algorithm

Bagging adopts the most popular strategies for aggregating the outputs of the base learners:

- voting for classification and
- averaging for regression.

To predict a test instance, taking classification for example, Bagging feeds the instance to its base classifiers and collects all their outputs.

Then votes the labels and takes the winner label as the prediction, where ties are broken arbitrarily.

37

The Bagging Algorithm

The Bagging algorithm is summarized as follows

Input: Data set $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
Base learning algorithm $\mathfrak{L}$;
Number of base learners T.

Process:
1. **for** $t = 1, \ldots, T$:
2. $h_t = \mathfrak{L}(D, \mathcal{D}_{bs})$ % $\mathcal{D}_{bs}$ is the bootstrap distribution
3. **end**

Output: $H(\boldsymbol{x}) = \arg\max_{y \in \mathcal{Y}} \sum_{t=1}^{T} \mathbb{I}(h_t(\boldsymbol{x}) = y)$

38

The Bagging Algorithm

Notice that Bagging can deal with binary classification as well as multi-class classification.

Bootstrap sampling also offers Bagging another advantage: given m training examples, the probability that the i-th training example is selected 0, 1, 2, … times is approximately Poisson distributed with $\lambda = 1$.

Thus, the probability that the i-th sample will occur at least once is $1 - (1/e) \approx 0.632$.

This means that for each base learner in Bagging there are about 36.8% original training examples that are not used to train it.

The goodness of the base learner can be estimated by using these **out-of-bag** examples, and thereafter the generalization error of the bagged ensemble can be estimated.

39

The Bagging Algorithm

To get the out-of-bag estimate, we need to record the training examples used for each base learner. Denote $H^{oob}(x)$ as the out-of-bag prediction on x, where only the learners that have not been trained on x are considered, i.e.

$$H^{oob}(\boldsymbol{x}) = \arg\max_{y \in \mathcal{Y}} \sum_{t=1}^{T} \mathbb{I}(h_t(\boldsymbol{x}) = y) \cdot \mathbb{I}(\boldsymbol{x} \notin D_t)$$

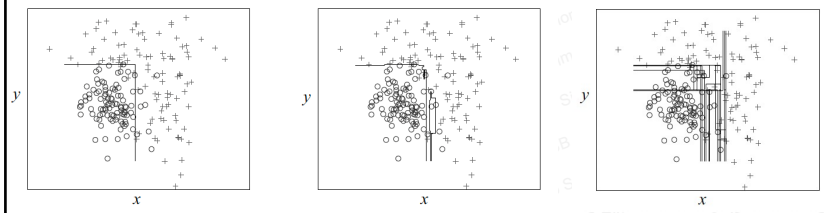
Then, the out-of-bag estimate of the generalization error of Bagging is:

$$err^{oob} = \frac{1}{|D|} \sum_{(\boldsymbol{x},y) \in D} \mathbb{I}(H^{oob}(\boldsymbol{x}) \neq y)$$

40

The Bagging algorithm: an example

As an example, we run Bagging on the three-Gaussians dataset.



Decision boundary of a single decision tree

Decision boundary of Bagging

Decision boundary of the 10 trees used by Bagging

41

AA 2022/2023

Random Tree Ensembles

42

Random Forest

Random Forest (RF) is a representative of the state-of-the-art ensemble methods.

It is an extension of bagging, where the major difference with Bagging is the incorporation of randomized feature selection.

During the construction of a component decision tree, at each step of split selection, RF first randomly selects a subset of features, and then carries out the conventional split selection procedure within the selected feature subset.

43

Random Forest

The random tree algorithm in RF:

Input: Data set $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
Feature subset size K.

Process:
1. $N \leftarrow$ create a tree node based on D;
2. **if** *all instances in the same class* **then return** N
3. $\mathcal{F} \leftarrow$ the set of features that can be split further;
4. **if** $\mathcal{F}$ *is empty* **then return** N
5. $\tilde{\mathcal{F}} \leftarrow$ select K features from $\mathcal{F}$ randomly;
6. $N.f \leftarrow$ the feature which has the best split point in $\tilde{\mathcal{F}}$;
7. $N.p \leftarrow$ the best split point on $N.f$;
8. $D_l \leftarrow$ subset of D with values on $N.f$ smaller than $N.p$;
9. $D_r \leftarrow$ subset of D with values on $N.f$ no smaller than $N.p$;
10. $N_l \leftarrow$ call the process with parameters (D_l, K);
11. $N_r \leftarrow$ call the process with parameters (D_r, K);
12. **return** N

Output: A random decision tree

44

Random Forest

The parameter K controls the incorporation of randomness.

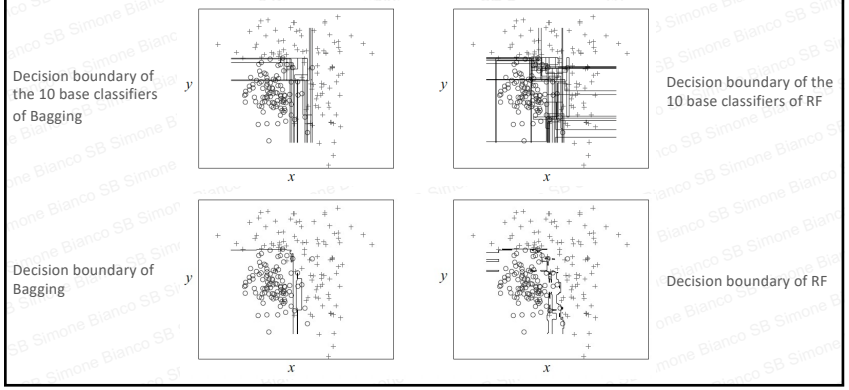
When K equals the total number of features, the constructed decision tree is identical to the traditional deterministic decision tree; when K = 1, a feature will be selected randomly.

The suggested value of K is the logarithm of the number of features. Notice that randomness is only introduced into the feature selection process, not into the choice of split points on the selected feature.

45

Random Forest: an example

As an example, we run RF on the three-Gaussians dataset.

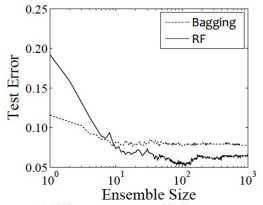


46

Random Forest

The convergence property of RF is similar to that of Bagging.

RF usually has worse starting point, particularly when the ensemble size is one, owing to the performance degeneration of single base learners by the incorporation of randomized feature selection; however, it usually converges to lower test errors.



The training stage of RF is generally more efficient than Bagging: this is because in the tree construction process, Bagging uses deterministic decision trees which need to evaluate all features for split selection, while RF uses random decision trees which only need to evaluate a subset of features.

47

Random Tree Ensembles for Density Estimation

Random tree ensembles can be used for **density estimation**.

Since density estimation is an unsupervised task, there is no label information for the training instances, and thus, completely random trees are used.

A completely random tree does not test whether the instances belong to the same class; instead, it grows until every leaf node contains only one instance or indistinguishable instances.

The completely random decision tree construction algorithm can be obtained as follows:

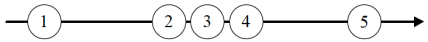
Input: Data set $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$;
Feature subset size K.
Process:
1. $N \leftarrow$ create a tree node based on D;
2. **if** ~~*all instances in the same class*~~ **then return** N → *only one instance*
3. $\mathcal{F} \leftarrow$ the set of features that can be split further;
4. **if** $\mathcal{F}$ *is empty* **then return** N
5. ~~$\tilde{\mathcal{F}} \leftarrow$ select K features from $\mathcal{F}$ randomly;~~
6. ~~$N.f \leftarrow$ the feature which has the best split point in $\tilde{\mathcal{F}}$;~~
7. ~~$N.p \leftarrow$ the best split point on $N.f$;~~
8. ~~$D_l \leftarrow$ subset of D with values on $N.f$ smaller than $N.p$;~~
9. $D_r \leftarrow$ subset of D with values on $N.f$ no smaller than $N.p$;
10. $N_l \leftarrow$ call the process with parameters (D_l, K);
11. $N_r \leftarrow$ call the process with parameters (D_r, K);
12. **return** N

Output: A random decision tree

48

Random Tree Ensembles for Density Estimation

Let's plot five 1-D data points



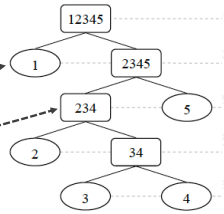
The completely random tree grows until every instance falls into a sole leaf node.

With a dominating probability, the split point falls either in between the points 1 and 2, or in between the points 4 and 5, since the gaps between these pairs of points are larger.

Suppose the split point adopted is in between the points 1 and 2, and thus, the point 1 is in a sole leaf node.

Then, the next split point will be picked in between the points 4 and 5 with a large probability, and this will make the point 5 be put into a sole leaf node.

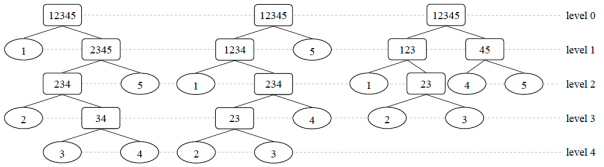
The points 1 and 5 are more likely to be put into "shallow" leaf nodes, while the points 2 to 4 are more likely to be put into "deep" leaf nodes.



49

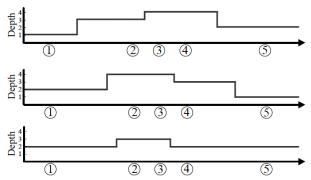
Random Tree Ensembles for Density Estimation

These are 3 completely random trees generated from the data:



We can count the average depth of each data point:

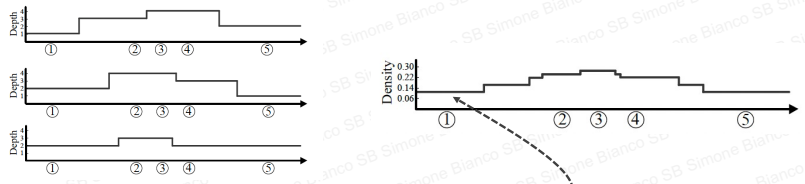
- (1): (1+2+2)/3 = 5/3 = 1.67
- (2): (3+4+3)/3 = 10/3 = 3.33
- (3): (4+4+3)/3 = 11/3 = 3.67
- (4): (4+3+2)/3 = 9/3 = 3
- (5): (2+1+2)/3 = 5/3 = 1.67



50

Random Tree Ensembles for Density Estimation

These are 3 completely random trees generated from the data:



By summation and normalization, we can obtain the density estimation:

- Density @(1): 1.67/(1.67*2+3.33+3.67+3) = 1.67/13.34 = 0.13

…

From the density plot generated we can conclude that even though just 3 trees are used, the points (1) and (5) are located in a relatively sparse area, while the points (2), (3), and (4) are located in relatively dense areas.

51

Random Tree Ensembles for Density Estimation

The principle illustrated on one-dimensional data above also holds for higher-dimensional data and for more complex data distributions.

It is also easy to extend to tasks where the ensembles are constructed incrementally, such as in online learning or on streaming data.

Notice that the construction of a completely random tree is quite efficient, since all it has to do is to pick random numbers.

Overall, the data density can be estimated through generating an ensemble of completely random trees and then calculating the average depth of each data point; this provides a practical and efficient tool for density estimation.

52

Random Tree Ensembles for Anomaly Detection

Anomalies are data points which do not conform to the general or expected behavior of the majority of data.

The task of anomaly detection is to separate anomaly points from normal ones in a given data set.

In general, the terms anomalies and **outliers** are used interchangeably, and anomalies are also referred to as discordant observations, exceptions, peculiarities, etc.

There are many established approaches to anomaly detection: a typical one is to estimate the data density, and then treat the data points with very low densities as anomalies.

However, density is not a good indicator to anomaly, because a clustered small group of anomaly points may have a high density, while the bordering normal points may be with low density.

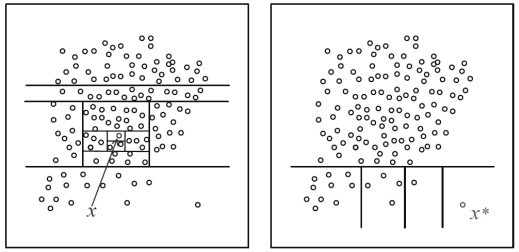
Since the basic property of anomalies is *few and different*, **isolation** is more indicative than density.

53

Random Tree Ensembles for Anomaly Detection

RT ensembles can serve well for anomaly detection, since random trees are simple yet effective for measuring the difficulty of isolating data points.

It can be observed that a normal point x generally requires more partitions to be isolated, while an anomaly point x^* is much easier to be isolated with many fewer partitions.



54

Random Tree Ensembles for Anomaly Detection

Liu et al. [2008b] described the iForest (Isolation Forest) method for anomaly detection.

For each random tree, the number of partitions required to isolate a data point can be measured by the path length from the root node to the leaf node containing the data point.

The fewer the required partitions, the easier the data point to be isolated.

It is obvious that only the data points with short path lengths are of interest.

Thus, to reduce unnecessary computation, the random trees used in iForest are set with a depth limit:

Input: Data set $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$;
Feature subset size K.

Process:
1. $N \leftarrow$ create a tree node based on D;
2. **if** ~~*all instances in the same class*~~ **then return** N → *only one instance or height limit is reached*
3. $\mathcal{F} \leftarrow$ the set of features that can be split further;
4. **if** $\mathcal{F}$ *is empty* **then return** N
5. ~~$\tilde{\mathcal{F}} \leftarrow$ select K features from $\mathcal{F}$ randomly;~~
6. ~~$N.f \leftarrow$ the feature which has the best split point in $\tilde{\mathcal{F}}$;~~
7. ~~$N.p \leftarrow$ the best split point on $N.f$;~~
8. ~~$D_l \leftarrow$ subset of D with values on $N.f$ smaller than $N.p$;~~
9. $D_r \leftarrow$ subset of D with values on $N.f$ no smaller than $N.p$;
10. $N_l \leftarrow$ call the process with parameters (D_l, K);
11. $N_r \leftarrow$ call the process with parameters (D_r, K);
12. **return** N

Output: A random decision tree

55

Random Tree Ensembles for Anomaly Detection

To improve the efficiency and scalability on large data sets, the random trees are constructed from small sized samples instead of the original data set.

Given the data sample size ψ, the height limit of iForest is set to $\log_2 \psi$, which is approximately the average tree height with ψ leaf nodes.

To calculate the anomaly score $s(x_i)$ for x_i the expected path length $\mathbb{E}[h(x_i)]$ is derived by passing $x_i$through every random tree in the iForest ensemble.

The path length obtained at each random tree is added with an adjustment term $c(n)$ to account for the ungrown branch beyond the three height (depth) limit. For a tree with n nodes, $c(n)$ is set as the average path length

$$c(n) = \begin{cases} 2H(n-1) - (2(n-1)/n), & n > 1 \\ 0, & n = 1 \end{cases}$$

where $H(a)$ is the harmonic number that can be estimated by

$$H(a) \approx \ln(a) + 0.5772156649 \text{ (Euler's constant)}$$

56

Random Tree Ensembles for Anomaly Detection

Then, the anomaly score $s(x_i)$ is calculated according to

$$s(\boldsymbol{x_i}) = 2^{-\frac{\mathbb{E}[h(\boldsymbol{x_i})]}{c(\psi)}}$$

where c(ψ) serves as a normalization factor corresponding to the average path length of traversing a random tree constructed from a sub-sample with size ψ.

Finally:

- If $s(x_i) \approx 1$, then x_i is definitely an anomaly.
- If $s(x_i) \ll 0.5$, then x_i is quite safe to be regarded as a normal data point.
- If $s(x_i) \approx 0.5$ for all x_i, then there is no distinct anomaly.

57

AA 2022/2023

Stacking classifiers

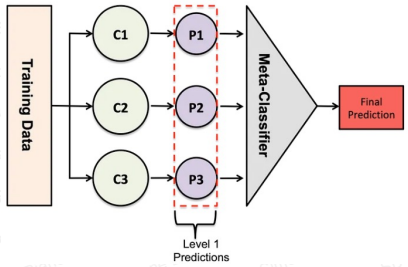
58

Stacking classifiers

The simplest form of stacking can be described as an ensemble learning technique where the predictions of multiple classifiers (referred as level-one classifiers) are used as new features to train a meta-classifier.

The meta-classifier can be any classifier of our choice.

E.g., we train three classifiers C1, C2, and C3 and their predictions get stacked and are used to train the meta-classifier.



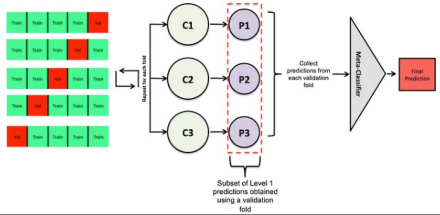
59

Stacking classifiers

To prevent information from leaking into the training from the target, the following rule should be followed when stacking classifiers:

The level one predictions should come from a subset of the training data that was not used to train the level one classifiers.

A robust way to do this is to use k-fold cross validation to generate the level one predictions: the training data is split into k-folds. Then the first k-1 folds are used to train the level one classifiers. The validation fold is then used to generate a subset of the level one predictions. The process is repeated for each unique group.



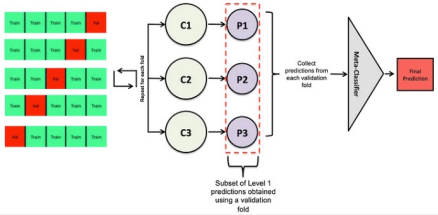
60

Stacking classifiers

To prevent information from leaking into the training from the target, the following rule should be followed when stacking classifiers:

The level one predictions should come from a subset of the training data that was not used to train the level one classifiers.

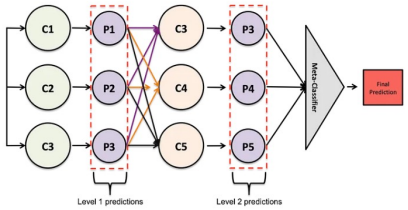
A robust way to do thisis to use k-fold cross validation to generate the level one predictions: the training data is split into k-folds. Then the first k-1 folds are used to train the level one classifiers. The validation fold is then used to generate a subset of the level one predictions. The process is repeated for each unique group.



61

Stacking classifiers

Why not stacking another layer of classifiers? Well… you can but this would add more complexity to the stack



Does it remind you anything?

This is starting to look like a neural network where each neuron is a classifier.

62