

MACHINE LEARNING

Ensemble methods

Corso di Laurea Magistrale in Informatica

Università di Roma Tor Vergata

Prof. Giorgio Gambosi

a.a. 2021-2022



Improve performance by combining multiple models, in some way, instead of using a single model.

- ⊙ train a *committee* of L different models and make predictions by averaging the predictions made by each model on dataset samplings (**bagging**)
- ⊙ train different models in sequence: the error function used to train a model depend on the performance of previous models (**boosting**)

- ⊙ Classifiers (especially some of them, such as decision trees) may have low performances due to their high variance: their behavior may largely differ in presence of slightly different training sets (or even of the same training set).
- ⊙ For example, in trees, the separations made by splits are enforced at all lower levels: hence, if the data is perturbed slightly, the new tree can have a considerably different sequence of splits, leading to a different classification rule

- ⊙ The **bootstrap** is a fundamental resampling tool in statistics. The basic underlying idea is to estimate the true distribution of data \mathcal{F} by the so-called empirical distribution $\hat{\mathcal{F}}$
- ⊙ Given the training data $(\mathbf{x}_i, t_i), i = 1, \dots, n$, the empirical distribution function \hat{p} is defined as

$$\hat{p}(\mathbf{x}, t) = \begin{cases} \frac{1}{n} & \text{if } \exists i : (\mathbf{x}, t) = (\mathbf{x}_i, t_i) \\ 0 & \text{otherwise} \end{cases}$$

- ⊙ This is just a discrete probability distribution, putting equal weight $\frac{1}{n}$ on each of the observed training points

- ⊙ A **bootstrap sample** of size m from the training data is

$$(\mathbf{x}_i^*, t_i^*) \quad i = 1, \dots, m$$

where each (\mathbf{x}_i^*, t_i^*) is drawn uniformly at random from $(\mathbf{x}_1, t_1), \dots, (\mathbf{x}_n, t_n)$, **with replacement**

- ⊙ This corresponds exactly to m independent draws from $\hat{\mathcal{F}}$: it approximates what we would see if we could sample more data from the true \mathcal{F} . We often consider $m = n$, which is like sampling an entirely new training set

- ⊙ Given a training set $(\mathbf{x}_i, y_i), i = 1, \dots, n$, bagging averages the predictions done by classifiers of the same type (such as decision trees) over a collection of bootstrap samples. For $b = 1, \dots, B$ (e.g., $B = 100$), n bootstrap items $(\mathbf{x}_i^b, y_i^b), i = 1, \dots, n$ are sampled and a classifier is fit on this set.
- ⊙ At the end, to classify an input \mathbf{x} , we simply take the most commonly predicted class, among all B classifiers
- ⊙ This is just choosing the class with the most votes
- ⊙ In the case of regression, the predicted value is derived as the average among the predictions returned by the B regressors

If the used classifier returns class probabilities $\hat{p}_k^b(\mathbf{x})$, the final bagged probabilities can be computed by averaging

$$p_k^b(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^B \hat{p}_k^b(\mathbf{x})$$

the predicted class is, again, the one with highest probability

Bagging classification

- ⊙ Why is bagging working?
- ⊙ Let us consider, for simplicity, a binary classification problem. Suppose that for a given input \mathbf{x} , we have B independent classifiers, each with a given misclassification rate e (for example, $e = 0.4$). Assume w.l.o.g. that the true class at \mathbf{x} is 1: so the probability that the b -th classifier predicts class 0 is $e = 0.4$
- ⊙ Let $B_0 \leq B$ be the number of classifiers returning class 0 on input \mathbf{x} : the probability of B_0 is clearly distributed according to a binomial (if classifiers are independent)

$$B_0 \sim \text{Binomial}(B, e)$$

the misclassification rate of the bagged classifier is then

$$p\left(B_0 > \frac{B}{2}\right) = \sum_{k=\frac{B}{2}+1}^B \binom{B}{k} e^k (1-e)^{B-k}$$

which tends to 0 as B increases.

Bagging regression

- Expected error of one model $y_i(\mathbf{x})$ wrt the true function $h(\mathbf{x})$:

$$E_{\mathbf{x}}[(y_i(\mathbf{x}) - h(\mathbf{x}))^2] = E_{\mathbf{x}}[\varepsilon_i(\mathbf{x})^2]$$

- Average expected error of the models

$$E_{av} = \frac{1}{m} \sum_{i=1}^m E_{\mathbf{x}}[\varepsilon_i(\mathbf{x})^2]$$

- Committee expected error

$$E_c = E_{\mathbf{x}} \left[\left(\frac{1}{m} \sum_{i=1}^m y_i(\mathbf{x}) - h(\mathbf{x}) \right)^2 \right] = E_{\mathbf{x}} \left[\left(\frac{1}{m} \sum_{i=1}^m \varepsilon_i(\mathbf{x}) \right)^2 \right]$$

If $E_{\mathbf{x}}[\varepsilon_i(\mathbf{x})\varepsilon_j(\mathbf{x})] = 0$ if $i \neq j$ (errors are uncorrelated) then $E_c = \frac{1}{m} E_{av}$.

- This is usually not verified: errors from different models are highly correlated.

- ⊙ Model evaluation can be performed by evaluating, for each item \mathbf{x}_i in the data set, the prediction done by the set of models trained on bootstrap samples not including \mathbf{x}_i .
- ⊙ If bootstrap samples have the same size of the dataset (i.e. $m = n$), there is a probability .63 that an item is included in a bootstrap sample: in fact, for each sample, the probability that item \mathbf{x}_i is not selected is $1 - \frac{1}{n}$. Hence there is a probability $\left(1 - \frac{1}{n}\right)^n$ that it is never sampled. For large enough values of n , the probability is about $\lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e} \approx .37$
- ⊙ In out-of-bag evaluation, the prediction of an item is done by using approximately a fraction .37 of all the trees. For those trees the item can be considered as a test set member.

Application of bagging to a set of (random) decision trees: classification performed by voting.

1. For $b = 1$ to B :
 - 1.1 Bootstrap sample from training set
 - 1.2 Grow a decision tree T_b on such data by performing the following operations for each node:
 - 1.2.1 select m variables at random
 - 1.2.2 pick the best variable among them
 - 1.2.3 split the node into two children
2. output the collection of trees T_1, \dots, T_B

Overall prediction is performed as majority (for classification) or average (for regression) among trees predictions.

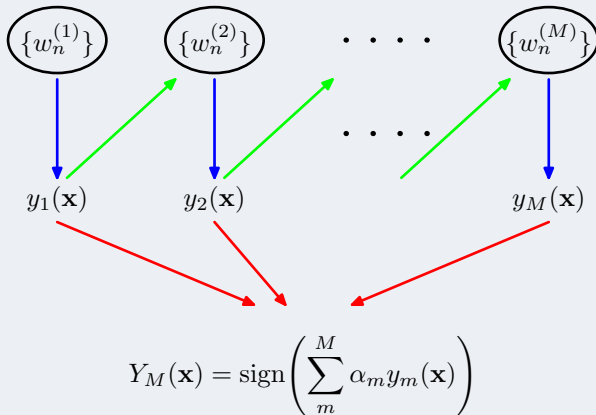
- ⊙ Boosting is a procedure to combine the output of many weak classifiers to produce a powerful committee.
- ⊙ A weak classifier is one whose error rate is only slightly better than random guessing.
- ⊙ Boosting produces a sequence of weak classifiers $y_m(\mathbf{x})$ for $m = 1, \dots, m$ whose predictions are then combined through a weighted majority to produce the final prediction

$$y(\mathbf{x}) = \text{sgn} \left(\sum_{j=1}^m \alpha_j y_j(\mathbf{x}) \right)$$

- ⊙ Each $\alpha_j > 0$ is computed by the boosting algorithm and reflects how accurately y_m classified the data.

Adaboost (adaptive boosting)

- ⦿ Models are trained in sequence: each model is trained using a weighted form of the dataset
- ⦿ Element weights depend on the performances of the previous models (misclassified points receive larger weights)
- ⦿ Predictions are performed through a weighted majority voting scheme on all models



Binary classification, dataset (\mathbf{X}, \mathbf{t}) of size n , with $t_i \in \{-1, 1\}$. The algorithm maintains a set of weights $w(\mathbf{x}) = (w_1, \dots, w_n)$ associated to the dataset elements.

- ⊙ Initialize weights as $w_i^{(0)} = \frac{1}{n}$ for $i = 1, \dots, n$
- ⊙ For $j = 1, \dots, m$:
 - Train a **weak learner** $y_j(\mathbf{x})$ on \mathbf{X} in such a way to minimize the weighted misclassification wrt to $w^{(j)}(\mathbf{x})$.
 - Let

$$e^{(j)} = \frac{\sum_{\mathbf{x}_i \in \mathcal{E}^{(j)}} w_i^{(j)}}{\sum_i w_i^{(j)}}$$

where $\mathcal{E}^{(j)}$ is the set of dataset elements misclassified by $y_j(\mathbf{x})$.

- If $e^{(j)} > \frac{1}{2}$, consider the reverse learner, which returns opposite predictions for all elements.
- $e^{(j)}$ can be interpreted as the probability that a random item from the training set is misclassified, assuming that item \mathbf{x}_i can be sampled with probability $\frac{w_i^{(j)}}{\sum_i w_i^{(j)}}$

- ⊙ Compute the learner confidence as log odds of a random item being well classified $(1 - e^{(j)})$ vs being misclassified $e^{(j)}$

$$\alpha_j = \frac{1}{2} \log \frac{1 - e^{(j)}}{e^{(j)}} > 0$$

- ⊙ For each \mathbf{x}_i , update the corresponding weight as follows

$$w_i^{(j+1)} = w_i^{(j)} e^{-\alpha_j t_i y_j(\mathbf{x}_i)}$$

which results into

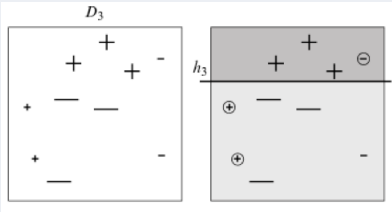
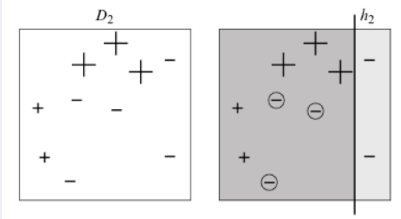
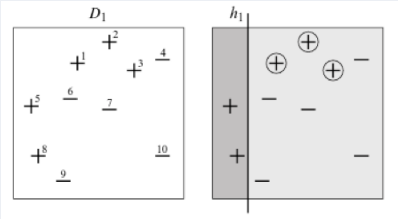
$$w_i^{(j+1)} = \begin{cases} w_i^{(j)} e^{\alpha_j} > w_i^{(j)} & \text{if } \mathbf{x}_i \in \mathcal{C}^{(j)} \\ w_i^{(j)} e^{-\alpha_j} < w_i^{(j)} & \text{otherwise} \end{cases}$$

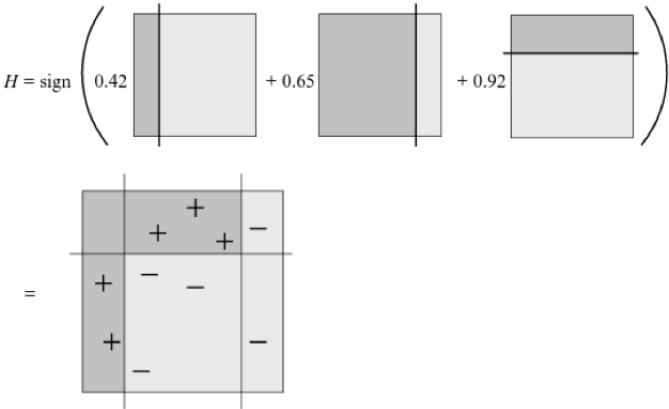
- ⊙ Normalize the set of $w_i^{(j+1)}$ by dividing each of them by $\sum_{i=1}^n w_i^{(j+1)}$, in order to get a distribution

The overall prediction is

$$y(\mathbf{x}) = \text{sgn} \left(\sum_{j=1}^m \alpha_j y_j(\mathbf{x}) \right)$$

since $y_j(\mathbf{x}) \in \{-1, 1\}$, this corresponds to a voting procedure, where each learner vote (class prediction) is weighted by the learner confidence.





Why does it work?

- ⊙ It minimizes a loss function related to classification error
- ⊙ Suppose we have a classifier $y(\mathbf{x}) = \text{sgn} f(\mathbf{x})$
- ⊙ We know that 0/1 loss

$$l(y(\mathbf{x}), t) = \begin{cases} 0 & \text{if } t f(\mathbf{x}) > 0 \\ 1 & \text{otherwise} \end{cases}$$

has drawbacks (non convex, gradient 0 almost everywhere). We need a surrogate loss.

- ⊙ Exponential loss

$$l(y(\mathbf{x}), t) = e^{-t f(\mathbf{x})}$$

- ⊙ Additive models are defined as the additive composition of simpler “base” predictors

$$y(\mathbf{x}) = \sum_{j=1}^m c_j \bar{y}(\mathbf{x})$$

where c_j 's are weights and $\bar{y}_j(\mathbf{x}) = \bar{y}(\mathbf{x}; \mathbf{w}) \in \mathbb{R}$ is a simple functions of the input \mathbf{x} parameterized by \mathbf{w}

- ⊙ in this case, the predictors are binary classifiers; that is, $\bar{y}_j(\mathbf{x}) = \bar{y}(\mathbf{x}; \mathbf{w}) \in \{-1, 1\}$

- ⊙ As usual, an additive model is fit by minimizing a loss function averaged over the training data:

$$\min_{c_j, \mathbf{w}_j} \sum_{i=1}^n L(t_i, \sum_{k=1}^m c_k \bar{y}(\mathbf{x}_i; \mathbf{w}_k))$$

- ⊙ For many loss functions L and/or predictors \bar{y} this is too hard

More simply, one can greedily add one predictor at a time in the following fashion.

- ⊙ Set $y_0(\mathbf{x}) = 0$
- ⊙ For $k = 1, \dots, m$:
 - Compute

$$(\hat{c}_k, \hat{\mathbf{w}}_k) = \underset{c_k, \mathbf{w}_k}{\operatorname{argmin}} \sum_{i=1}^n L(t_i, y_{k-1}(\mathbf{x}_i) + c_k \bar{y}(\mathbf{x}_i; \mathbf{w}_k))$$

- Set $y_k(\mathbf{x}) = y_{k-1}(\mathbf{x}) + \hat{c}_k \bar{y}(\mathbf{x}; \hat{\mathbf{w}}_k)$

That is, fitting is performed not modifying previously added terms (**greedy** paradigm)

Adaboost can be interpreted as fitting an additive model with **exponential loss**

$$L(y, f(\mathbf{x})) = e^{-tf(\mathbf{x})}$$

that is, minimizing

$$\sum_{i=1}^n e^{-t_i \sum_{k=1}^m \alpha_k \bar{y}(\mathbf{x}_i; \mathbf{w}_k)}$$

with respect to $\mathbf{w}_1, \dots, \mathbf{w}_m$ and $\alpha_1, \dots, \alpha_m$

Applying forward stagewise additive modeling, at each step k we compute

$$\begin{aligned}(\hat{\alpha}_k, \hat{\mathbf{w}}_k) &= \operatorname{argmin}_{\alpha_k, \mathbf{w}_k} \sum_{i=1}^n e^{-t_i y(\mathbf{x}_i)} \\&= \operatorname{argmin}_{\alpha_k, \mathbf{w}_k} \sum_{i=1}^n e^{-t_i (y_{k-1}(\mathbf{x}_i) + \alpha_k \bar{y}(\mathbf{x}_i; \mathbf{w}_k))} \\&= \operatorname{argmin}_{\alpha_k, \mathbf{w}_k} \sum_{i=1}^n w_i^{(k)} e^{-\alpha_k t_i \bar{y}(\mathbf{x}_i; \mathbf{w}_k)}\end{aligned}$$

where $w_i^{(k)} = e^{-t_i y_{k-1}(\mathbf{x}_i)} = e^{-\frac{1}{2} t_i \sum_{r=1}^{k-1} \alpha_r \bar{y}_r(\mathbf{x}_i)}$ is a constant wrt α_k and \mathbf{w}_k

The approach can be extended to the case of different loss functions

Find the next learner and related weight

We may decompose the weighted loss function as follows

$$\sum_{i=1}^n w_i^{(k)} e^{-\alpha_k t_i \bar{y}_k(\mathbf{x}_i)} = \sum_{i=1}^n w_i^{(k)} e^{-\alpha_k t_i \bar{y}(\mathbf{x}_i; \mathbf{w}_k)} = \sum_{\mathbf{x}_i \in \mathcal{G}^{(k)}} w_i^{(k)} e^{\alpha_k} + \sum_{\mathbf{x}_i \notin \mathcal{G}^{(k)}} w_i^{(k)} e^{-\alpha_k}$$

where $\mathcal{G}^{(k)}$ is the set of elements misclassified by \bar{y}_k

By adding and subtracting $\sum_{\mathbf{x}_i \in \mathcal{G}^{(k)}} w_i^{(k)} e^{-\alpha_k}$ we obtain the weighted loss function

$$\sum_{\mathbf{x}_i \in \mathcal{G}^{(k)}} w_i^{(k)} (e^{\alpha_k} - e^{-\alpha_k}) + e^{-\alpha_k} \sum_{\mathbf{x}_i} w_i^{(k)}$$

to be minimized wrt $\mathbf{w}^{(k)}$ and α_k

Find the next learner

To derive the best learner coefficients $\mathbf{w}^{(k)}$, we observe that they affect, through $\mathcal{G}^{(k)}$, only the first term

$$\sum_{\mathbf{x}_i \in \mathcal{G}^{(k)}} w_i^{(k)} (e^{\alpha_k} - e^{-\alpha_k})$$

has to be considered, since the other one is constant.

Since α_k is considered as a constant here, we have to minimize the sum of the current weights of misclassified items

$$\sum_{\mathbf{x}_i \in \mathcal{G}^{(k)}} w_i^{(k)}$$

wrt $\mathbf{w}^{(k)}$, which is what is done in Adaboost

Find the next learner weight

To derive the best learner weight α_k , we need to take into account the whole loss function.

By setting to 0 the derivative of the loss function wrt α_k , we get

$$\alpha_k = \frac{1}{2} \log \frac{1 - e^{(k)}}{e^{(k)}}$$

which again corresponds to what is done in Adaboost

Updating the element weights

By introducing the new learner \bar{y}_k with weight α_k , the overall predictor turn out to be

$$y_k(\mathbf{x}) = y_{k-1}(\mathbf{x}) + \alpha_k \bar{y}_k(\mathbf{x}) = y_{k-1}(\mathbf{x}) + \alpha_k \bar{y}(\mathbf{x}; \mathbf{w}_k)$$

Since by definition $w_i^{(k)} = e^{-t_i y_{k-1}(\mathbf{x}_i)}$ we have for the new weights $w_i^{(k+1)}$

$$w_i^{(k+1)} = e^{-t_i y_k(\mathbf{x}_i)} = e^{-t_i (y_{k-1}(\mathbf{x}_i) + \alpha_k \bar{y}(\mathbf{x}_i; \mathbf{w}_k))} = w_i^{(k)} e^{-t_i \alpha_k \bar{y}(\mathbf{x}_i; \mathbf{w}_k)}$$

again, as in Adaboost

General idea:

- ⊙ Fit an additive model $\sum_{j=1}^m \alpha_j y_j(\mathbf{x})$ in a forward stage-wise manner.
- ⊙ At each stage, introduce a weak learner to compensate the shortcomings of existing ones.
- ⊙ Shortcomings are identified by high-weight data points.

- ⊙ You are given $(\mathbf{x}_i, t_i), i = 1, \dots, n$, and the task is to fit a model $y(\mathbf{x})$ to minimize square loss.
- ⊙ Assume a model $y^{(1)}(\mathbf{x})$ is available, with residuals $t_i - y^{(1)}(\mathbf{x}_i) = t_i - y_i^{(1)}$
- ⊙ A new dataset $(\mathbf{x}_i, t_i - y_i^{(1)}), i = 1, \dots, n$ can be defined, and a model $\bar{y}^{(1)}(\mathbf{x})$ can be fit to minimize square loss wrt such dataset
- ⊙ Clearly, $y_2(\mathbf{x}) = y_1(\mathbf{x}) + \bar{y}_1(\mathbf{x})$ is a model which improves $y_1(\mathbf{x})$
- ⊙ The role of $\bar{y}_1(\mathbf{x})$ is to compensate the shortcoming of $y(\mathbf{x})$
- ⊙ If $y_2(\mathbf{x})$ is unsatisfactory, we may define new models $\bar{y}_2(\mathbf{x})$ and $y_3(\mathbf{x}) = y_2(\mathbf{x}) + \bar{y}_2(\mathbf{x})$

How is this related to gradient descent?

- ⊙ Let us consider the squared loss function $L(t, y) = \frac{1}{2}(t - y)^2$
- ⊙ We want to minimize the risk $R = \sum_{i=1}^n L(t_i, y_i)$ by adjusting y_1, \dots, y_n
- ⊙ Consider y_i as parameters and take derivatives

$$\frac{\partial R}{\partial y_i} = y_i - t_i$$

So, we can consider residuals as negative gradients

$$t_i - y_i = -\frac{\partial R}{\partial y_i}$$

- ⊙ Model $\bar{y}(\mathbf{x})$ can then be derived by considering the dataset

$$(\mathbf{x}_i, t_i - y_i) = \left(\mathbf{x}_i, -\frac{\partial R}{\partial y_i} \right) \quad i = 1, \dots, n$$

The following algorithm results

- ⊙ Set $y^{(1)}(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n t_i$
- ⊙ For $k = 1, \dots, m$:
 - Compute negative gradients

$$-g_i^{(k)} = -\frac{\partial R}{\partial y_i} \Big|_{y_i=y^{(k)}(\mathbf{x}_i)} = -\frac{\partial}{\partial y_i} L(t_i, y_i) \Big|_{y_i=y^{(k)}(\mathbf{x}_i)} = t_i - y^{(k)}(\mathbf{x}_i)$$

- Fit a weak learner $\bar{y}^{(k)}(\mathbf{x})$ to negative gradients, considering dataset $(\mathbf{x}_i, -g_i^{(k)}), i = 1, \dots, n$
- Derive the new classifier $y^{(k+1)}(\mathbf{x}) = y^{(k)}(\mathbf{x}) + \bar{y}^{(k)}(\mathbf{x})$

- ⊙ The benefit of formulating this algorithm using gradients is that it allows us to consider other loss functions and derive the corresponding algorithms in the same way.
- ⊙ For example, square loss is easy to deal with mathematically, but not robust to outliers, i.e. pays too much attention to outliers.
- ⊙ Different loss functions
 - Absolute loss
 - $L(t, y) = |t - y|$
 - $-g = \text{sgn}(t - y)$
 - Huber loss
 -
 -

$$L(t, y) = \begin{cases} \frac{1}{2}(t - y)^2 & |t - y| \leq \delta \\ \delta(|t - y|) - \frac{\delta}{2} & |t - y| > \delta \end{cases}$$

$$-g = \begin{cases} y - t & |t - y| \leq \delta \\ \delta \cdot \text{sgn}(t - y) & |t - y| > \delta \end{cases}$$

Which weak learners?

- ⊙ Regression trees (special case of decision trees)
- ⊙ Decision stumps (trees with only one node)