Ensemble methods

Course of Machine Learning
Master Degree in Computer Science
University of Rome "Tor Vergata"
a.a. 2024-2025

Giorgio Gambosi

Ensemble methods try to 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**)

Bootstrap

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 $(\mathbf{x}_i, t_i)$, $i = 1, \ldots, n$, the empirical distribution function $\hat{\mathcal{F}}$ 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

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

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

Bagging

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

- Given a training set $(\mathbf{x}_i, y_i)$, $i = 1, \ldots, n$, bagging averages the predictions done by classifiers of the same type (such as decision trees) over a collection of boostrap samples. For $b = 1, \ldots, B$ (e.g., B = 100), n bootstrap items $(\mathbf{x}_i^b, y_i^b)$, $i = 1, \ldots, n$ are sampled and a classifier is fit on this set.

- At the end, to classify an input 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

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.

In the case of 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.

Random forest

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

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

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

Boosting

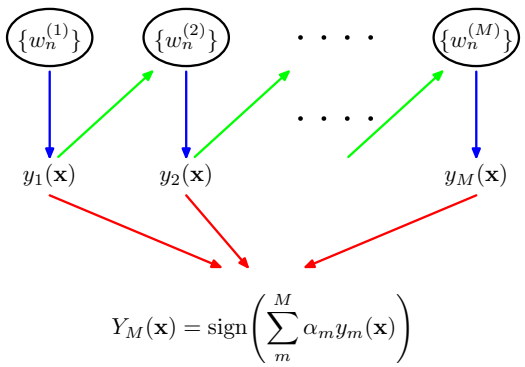
- Boosting is a procedure to combine the output of many weak classifiers to produce a powerful committee.
- A weak classifier is one whose error rate is only slightly better than random guessing.
- Boosting produces a sequence of weak classifiers $y_m(x)$ for $m = 1, \ldots, m$ whose predictions are 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 $w(\mathbf{x}) = (w_1, \ldots, w_n)$ 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
$$\pi^{(j)} = \frac{\sum_{\mathbf{x}_i \in \mathcal{E}^{(j)}} w_i^{(j)}}{\sum_i w_i^{(j)}}$$
    where $\mathcal{E}^{(j)}$ is the set of dataset elements misclassified by $y_j(\mathbf{x})$.
    * If $\pi^{(j)} > \frac{1}{2}$, consider the reverse learner, which returns opposite predictions for all elements.
    * $\pi^{(j)}$ 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 $(1 - \pi^{(j)})$ vs being misclassified $\pi^{(j)}$
$$\alpha_j = \frac{1}{2} \log \frac{1 - \pi^{(j)}}{\pi^{(j)}} > 0$$
- 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
$$w_i^{(j+1)} = \begin{cases} w_i^{(j)} e^{\alpha_j} > w_i^{(j)} & \text{if } \mathbf{x}_i \in \mathcal{E}^{(j)} \\ w_i^{(j)} e^{-\alpha_j} < w_i^{(j)} & \text{otherwise} \end{cases}$$
- 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

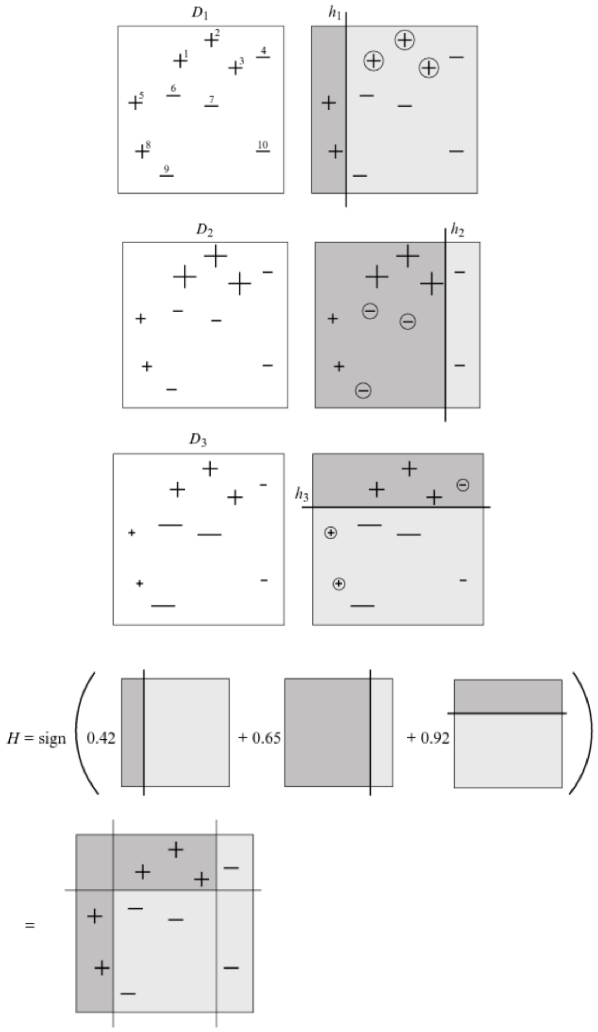
The overall prediction is
$$y(\mathbf{x}) = \text{sgn}\left(\sum_{j=1}^m \alpha_j y_j(\mathbf{x})\right)$$
since $y_j(\mathbf{x}) \in \{-1, 1\}$, this corresponds to a voting procedure, where each learner vote (class prediction) is weighted by the learner confidence.

Why does it work?

Observe that a weak learner confidence is inversely related to the probability of misclassification. Moreover,
$$w_i^{(t)} = \frac{1}{n} \prod_{j \in \mathcal{B}_i} \frac{1 - e^{(j)}}{e^{(j)}}$$
where $\mathcal{B}_i$ is the set of indices of "bad" weak learners wrt $\mathbf{x}_i$ (that is ones that misclassify $\mathbf{x}_i$)

Since $1 - e^{(j)} > e^{(j)}$ it derives that bad learners increase the probability of an element, while good learners decrease it.

- As iterations proceed, observations difficult to classify correctly receive more influence.
- Each successive classifier is forced to concentrate on training observations missed by previous ones in the sequence.



Additive models

Additive models are defined as the additive composition of simple "base" predictors h_j

$$y(\mathbf{x}) = \sum_{j=1}^{m} \alpha_j h_j(\mathbf{x})$$

where, for each j, α_j is a weight and $h_j(\mathbf{x}) = h(\mathbf{x}; \mathbf{w}_j) \in \mathbb{R}$ is a simple function of the input $\mathbf{x}$ parameterized by $\mathbf{w}_j \in \mathbb{R}^p$ for a given p

In this case, the predictors are binary classifiers; that is, $h_j(\mathbf{x}) \in \{-1, 1\}$.

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

$$\min_{\boldsymbol{\alpha}, \mathbf{W}} L\left(t_i, y(\mathbf{x})\right) = \min_{\boldsymbol{\alpha}, \mathbf{W}} \sum_{i=1}^{n} L\left(t_i, \sum_{j=1}^{m} \alpha_j h(\mathbf{x}_i; \mathbf{w}_j)\right)$$

with $\boldsymbol{\alpha} = \{\alpha_1, \ldots, \alpha_m\}$ and $\mathbf{W} = \cup_{j=1}^{m} \mathbf{w}_j$. For many loss functions L and/or additive predictors h this is too hard.

We may make things simpler by greedily adding one predictor at a time as follows: this is called **Forward stage-wise additive modeling**. According to this approach, the minimum of the loss function of the additive model is

approximated by sequentially adding new base predictors to the sum without adjusting the parameters and coefficients of those that have already been added. This is outlined below: at each iteration, one minimizes the loss function wrt the new predictor parameters $\mathbf{w}_k$and corresponding coefficient α_k. Previously added terms are not modified.

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

$$(\hat{\alpha}_k, \hat{\mathbf{w}}_k) = \underset{\alpha_k, \mathbf{w}_k}{\operatorname{argmin}} \sum_{i=1}^{n} L\left(t_i, y_{k-1}(\mathbf{x}_i) + \alpha_k h_k(\mathbf{x}_i)\right) = \underset{\alpha_k, \mathbf{w}_k}{\operatorname{argmin}} \sum_{i=1}^{n} L\left(t_i, y_{k-1}(\mathbf{x}_i) + \alpha_k h(\mathbf{x}_i; \mathbf{w}_k)\right)$$

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

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

The general idea, hence, is the following:

- 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.

Adaboost as additive model

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

$$L(t, y(\mathbf{x})) = e^{-ty(\mathbf{x})}$$

that is, minimizing

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

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

In Adaboost, we have that $p = n$. That is, the number of parameters in $h(\mathbf{x}, \mathbf{w})$ is equal to the number of items: hence, $\mathbf{w}_k = (w_{k1}, \dots, w_{kn})$ for all k.

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

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

where

$$w_i^{(k)} = e^{-t_i y_{k-1}(\mathbf{x}_i)} = e^{-\frac{1}{2} t_i \sum_{r=1}^{k-1} \alpha_r h(\mathbf{x}_i; \mathbf{w}_r)}$$

is a weight assigned to item $\mathbf{x}_i$ as an input to step k and is a constant wrt α_k and $\mathbf{w}_k$. Observe that the weight assigned to $\mathbf{x}_i$ varies at different steps, since it assumes values $w_i^{(0)}, w_i^{(1)}, w_i^{(2)}, \dots$.

Find the next learner and related weight

We may decompose the weighted loss function as follows

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

where $\mathcal{E}^{(k)}$ is the set of elements misclassified by h_k, that is the ones such that $t_i h(\mathbf{x}_i; \mathbf{w}_k) = -1$.

By adding and subtracting $\sum_{\mathbf{x}_i \in \mathcal{E}^{(k)}} w_i^{(k)} e^{-\alpha_k}$ the weighted loss function, to be minimized wrt $\mathbf{w}^{(k)}$ and α_k, can be written as

$$\left(\sum_{\mathbf{x}_i \in \mathcal{E}^{(k)}} w_i^{(k)} e^{\alpha_k} - \sum_{\mathbf{x}_i \in \mathcal{E}^{(k)}} w_i^{(k)} e^{-\alpha_k} \right) + \left(\sum_{\mathbf{x}_i \notin \mathcal{E}^{(k)}} w_i^{(k)} e^{-\alpha_k} + \sum_{\mathbf{x}_i \in \mathcal{E}^{(k)}} w_i^{(k)} e^{-\alpha_k} \right) =$$

$$\sum_{\mathbf{x}_i \in \mathcal{E}^{(k)}} w_i^{(k)} (e^{\alpha_k} - e^{-\alpha_k}) + e^{-\alpha_k} \sum_{i=1}^{n} w_i^{(k)}$$

To derive the best values of the learner weights $\hat{\mathbf{w}}_k$, we observe that their values affect, through $\mathcal{E}^{(k)}$, only the first term

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

The other one is indeed constant, since it only depends on $\mathbf{w}_1, \ldots, \mathbf{w}_{k-1}$ and $\alpha_1, \ldots, \alpha_{k-1}$.

Since α_k is considered as a constant here, also $e^{\alpha_k} - e^{-\alpha_k}$ is a constant, and we have to derive the value $\hat{\mathbf{w}}_k$ which makes the sum of the current weights of misclassified items

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

as small as possible. This is precisely what is done what is done in Adaboost.

To derive the best learner weight α_k, we need to take into account the whole loss function. This can be done by setting

$$\frac{\partial}{\partial \alpha_k} \sum_{i=1}^{n} w_i^{(k)} e^{-\alpha_k t_i h(\mathbf{x}_i;\mathbf{w}_k)} = \frac{\partial}{\partial \alpha_k} \sum_{\mathbf{x}_i \in \mathcal{E}^{(k)}} w_i^{(k)} e^{\alpha_k} + \frac{\partial}{\partial \alpha_k} \sum_{\mathbf{x}_i \notin \mathcal{E}^{(k)}} w_i^{(k)} e^{-\alpha_k} = 0$$

which results into

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

with

$$\pi^{(k)} = \frac{\sum_{\mathbf{x}_i \in \mathcal{E}^{(k)}} w_i^{(k)}}{\sum_{i=1}^{n} w_i^{(k)}}$$

This again corresponds to what is done in Adaboost.

Updating the element weights

By introducing the new learner y_k with weight α_k, the overall predictor turns out to be

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

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

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

again, as in Adaboost.

Gradient boosting

- You are given $(\mathbf{x}_i, t_i)$, $i = 1, \ldots, n$, 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 $t_i - y_i^{(1)} = t_i - y^{(1)}(\mathbf{x}_i)$
- A new dataset $(\mathbf{x}_i, t_i - y_i^{(1)})$, $i = 1, \ldots, n$ can be defined, and a model $h^{(1)}(\mathbf{x})$ can be fit to minimize square loss wrt such dataset
- Clearly, $y_2(\mathbf{x}) = y_1(\mathbf{x}) + h_1(\mathbf{x})$ is a model which improves $y_1(\mathbf{x})$
- The role of $h_1(\mathbf{x})$ is to compensate the shortcoming of $y(\mathbf{x})$
- If $y_2(\mathbf{x})$ is unsatisfactory, we may define new models $h_2(\mathbf{x})$ and $y_3(\mathbf{x}) = y_2(\mathbf{x}) + h_2(\mathbf{x})$

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 empirical risk $R = \sum_{i=1}^{n} L(t_i, y_i)$ by adjusting $y_1, \ldots, y_n$, considered as parameters
- For each y_i we consider the derivative

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

  The residuals correspond then to negative gradients

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

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

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

Looking at the new dataset

$$\left\{\left(\mathbf{x}_i, -\frac{\partial R}{\partial y_i}\right), \ldots, \left(\mathbf{x}_n, -\frac{\partial R}{\partial y_n}\right)\right\}$$

we wonder what is the meaning of looking for a predictor h which fits such points.

- The idea is that $h(\mathbf{x}_i)$ should be small if the cost derived from the current prediction y_i of $\mathbf{x}_i$ is almost constant: modifying the prediction results into a limited gain wrt the cost
- similarly, if the cost would increase considerably by increasing the prediction value, then $h(\mathbf{x}_i)$ should modify such cost by decreasing it; that is it should be more negative
- finally, by symmetry, if the cost would decrease considerably by increasing the prediction value, then $h(\mathbf{x}_i)$ should modify such cost by increasing it; that is it should be more positive

Gradient boosting for regression

The following algorithm results for a regression task

- Set $y^{(1)}(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} t_i$
- For $k = 1, \ldots, m$:
  - 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 $h^{(k)}(\mathbf{x})$ to negative gradients, considering dataset $(\mathbf{x}_i, -g_i^{(k)})$, $i = 1, \ldots, n$
  - Derive the new classifier $y^{(k+1)}(\mathbf{x}) = y^{(k)}(\mathbf{x}) + h^{(k)}(\mathbf{x})$

A 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

$$\begin{aligned} L(t, y) &= |t - y| \\ -g &= \operatorname{sgn}(t - y) \end{aligned}$$

- Huber loss

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

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

A similar approach can be applied on K-class classification, with

$$R = \sum_{i=1}^{n} L(t_i, y_1(\mathbf{x}_i), \ldots, y_K(\mathbf{x}_i)) = \sum_{i=1}^{n} L((t_{i1}, \ldots, t_{iK}), (y_{i1}, \ldots, y_{iK}))$$

for a given loss function, where $(t_{i1}, \ldots, t_{iK})$ is the 1-to-K encoding of t_i.

- Set $y_{ij}^{(1)} = y_j^{(1)}(\mathbf{x}_i) = \frac{1}{K}$, for $j = 1, \ldots, K$ and $i = 1, \ldots, n$
- For $k = 1, \ldots, m$:
  - Compute negative gradients

$$-g_{ij}^{(k)} = -\frac{\partial R}{\partial y_{ij}}\Big|_{y_{ij} = y_j^{(k)}(\mathbf{x}_i)} = -\frac{\partial}{\partial y_{ij}} L((t_{i1}, \ldots, t_{iK}), (y_{i1}, \ldots, y_{iK}))\Big|_{y_{ij} = y_j^{(k)}(\mathbf{x}_i)}$$

  - for $j = 1, \ldots, K$
    1. Fit K weak learners $h_j^{(k)}(\mathbf{x})$ $(j = 1, \ldots, K)$ to negative gradients, considering dataset

$$(\mathbf{x}_i, (-g_{i1}^{(k)}, -g_{i2}^{(k)}, \ldots, -g_{iK}^{(k)})) \qquad i = 1, \ldots, n$$

    2. Derive the new classifiers $y_j^{(k+1)}(\mathbf{x}) = y_j^{(k)}(\mathbf{x}) + h_j^{(k)}(\mathbf{x})$

Which weak learners?

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