Ensemble learning

Course of Machine Learning
Master Degree in Computer Science
University of Rome “Tor Vergata”

Giorgio Gambosi

a.a. 2018-2019

Idea

Partly derived from R. Tibshirani slides, Data Mining course, Statistics degree, Carnegie-Mellon University

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)

Committees

- Average the predictions of a set (committee) of m individual models, each separately trained (Bagging, bootstrap aggregation).
- Generate m training sets by bootstrap on the original dataset $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$
  - A dataset $\mathbf{X}_i$ $(i = 1, \ldots, m)$ of size n is generated by drawing n samples with replacement from $\mathbf{X}$
- The committee prediction in the case of regression is

$$y_c(\mathbf{x}) = \frac{1}{m} \sum_{i=1}^{m} y_i(\mathbf{x})$$

Bagging

Classifiers (especially some of them, such as decision trees) may have low prediction accuracy competitive due to 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

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

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

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

Bootstrap

A bootstrap sample of size m from the training data is

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

where each $(\mathbf{x}_i^*, y_i^*)$ are are drawn from uniformly at random from $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, with replacement

This corresponds exactly to m independent draws from $\hat{\mathcal{F}}$. Hence 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

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

Bagging variant

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.

Out-of-bag error

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 $\lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e} \approx .37$

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.

Random forest

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 $T_1, \ldots, T_B$

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

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

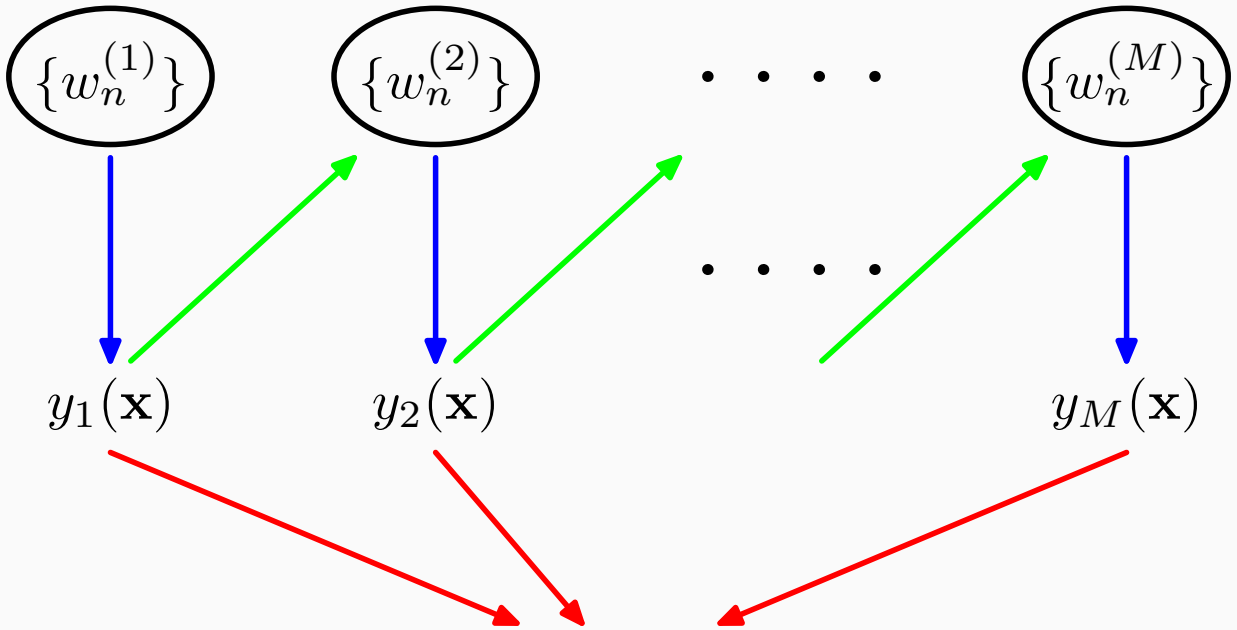
- Each $\alpha_j > 0$ is computed by the boosting algorithm and reflects how accurately y_m classified the data.

Boosting

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

Boosting



$$Y_M(\mathbf{x}) = \text{sign}\left(\sum_m^M \alpha_m y_m(\mathbf{x})\right)$$

Adaboost

Binary classification, dataset $(\mathbf{X}, \mathbf{t})$ of size n, with $t_i \in \{-1, 1\}$. The algorithm maintains a probability distribution $p(\mathbf{x}) = (p_1, \ldots, p_n)$ on the dataset elements.

- Set $p^{(0)} = (p_1^{(0)}, \ldots, p_n^{(0)})$, with $p_i^{(0)} = \frac{1}{n}$ for $i = 1, \ldots, n$
- For $j = 1, \ldots, m$:
  - Train a weak learner $y_j(\mathbf{x})$ on $\mathbf{X}$ in such a way to minimize the probability of misclassification wrt to $p^{(j)}(\mathbf{x})$.

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

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

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

  - For each $\mathbf{x}_i \in \mathcal{E}^{(j)}$ update the corresponding weight as follows

    $$p_i^{(j+1)} = p_i^{(j)} e^{\alpha_j}$$

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

Adaboost

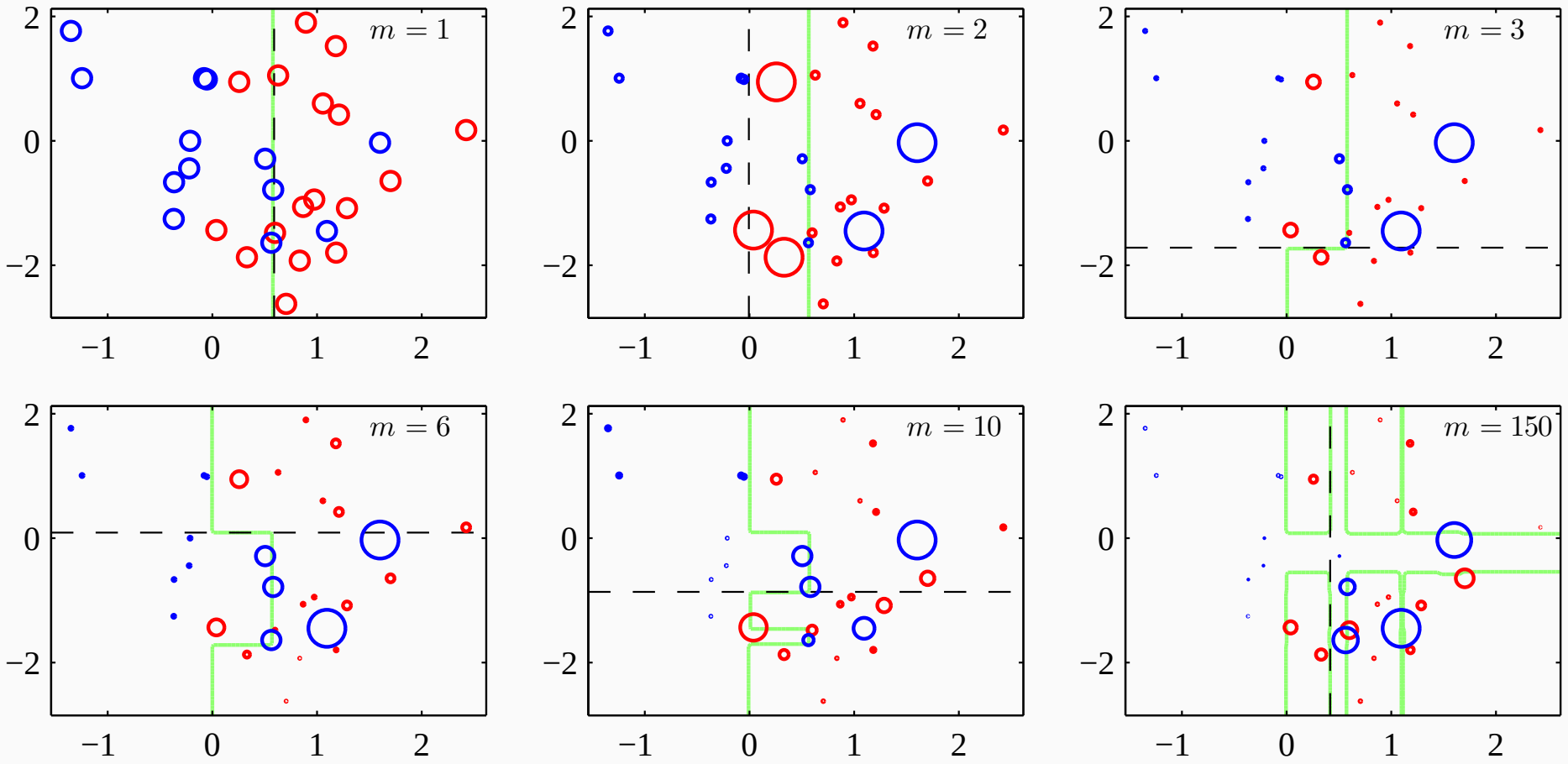
The overall prediction is

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

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.

- 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.

Adaboost



Adaboost as additive model

- Boosting fits an additive expansion in a set of elementary basis functions.
- In general, basis function expansions take the form

$$f(\mathbf{x}) = \sum_{j=1}^{m} c_j \phi(\mathbf{x}; \mathbf{w}_j)$$

  where c_j's are the expansion coefficients and $\phi(\mathbf{x}; \mathbf{w}) \in \mathbb{R}$ are simple functions of the input $\mathbf{x}$ parameterized by $\mathbf{w}$
  Example: single-hidden-layer neural networks

$$\phi(\mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x} + w_0)$$

- In our case, the basis functions are the weak classifiers $y_j(\mathbf{x}) \in \{-1, 1\}$

Fitting additive models

- Typically fit model by minimizing a loss function averaged over the training data:

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

- For many loss functions L and/or basis functions ϕ this is too hard

Forward stagewise additive modelling

- More simply, one can greedily add one basis function at a time in the following fashion.
  - Set $f_0(\mathbf{x}) = 0$
  - For $k = 1, \ldots, m$:
    - Compute

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

    - Set $f_k(\mathbf{x}) = f_{k-1}(\mathbf{x}) + \hat{c}_k \phi(\mathbf{x}; \hat{\mathbf{w}}_k)$

    That is, fitting is performed not modifying previously added terms

Adaboost as additive model

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

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

that is, minimizing

$$\underset{c_j, \mathbf{w}_j; j=1,\ldots,m}{\operatorname{argmin}} \sum_{i=1}^{n} e^{-t_i \sum_{k=1}^{m} c_k \phi(\mathbf{x}_i; \mathbf{w}_k)}$$

Adaboost as additive model

Applying forward stagewise additive modelling, at each step it computes

$$
\begin{aligned}
(\hat{c}_k, \hat{\mathbf{w}}_k) &= \underset{c_k, \mathbf{w}_k}{\operatorname{argmin}} \sum_{i=1}^{n} e^{-t_i (f_{k-1}(\mathbf{x}_i) + c_k \phi(\mathbf{x}_i; \mathbf{w}_k))} \\
&= \underset{c_k, \mathbf{w}_k}{\operatorname{argmin}} \sum_{i=1}^{n} p_i^{(k)} e^{c_k \phi(\mathbf{x}_i; \mathbf{w}_k)}
\end{aligned}
$$

where $p_i^{(k)} = e^{-t_i f_{k-1}(\mathbf{x}_i)}$ is a constant wrt c_k and $\mathbf{w}_k$

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

Gradient boosting

Adaboost 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.

Gradient boosting 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.

Gradient boosting

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

Gradient boosting

How is this related to gradient descent?

- Loss function $L(t, y(\mathbf{x})) = \frac{1}{2}(t - y(\mathbf{x}))^2$
- We want to minimize the risk $R = \sum_{i=1}^{n} L(t_i, y(\mathbf{x}_i))$ by adjusting $y(\mathbf{x}_1), \ldots, y(\mathbf{x}_n)$
- Consider $y(\mathbf{x}_i)$ as parameters and take derivatives

$$\frac{\partial R}{\partial y(\mathbf{x}_i)} = y(\mathbf{x}_i) - t_i$$

  So, we can consider residuals as negative gradients

$$t_i - y(\mathbf{x}_i) = -\frac{\partial R}{\partial y(\mathbf{x}_i)}$$

- $\overline{y}(\mathbf{x})$ can then be derived by considering the dataset $(\mathbf{x}_i, -\frac{\partial R}{\partial y(\mathbf{x}_i)})$, $i = 1, \ldots, n$

Gradient boosting for regression

The following algorithm derives

- Set $y_1(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} t_i$
- For $k = 1, \ldots, m$:
  - Compute negative gradients

$$-g(\mathbf{x}_i) = -\frac{\partial R}{\partial y(\mathbf{x}_i)} = t_i - y(\mathbf{x}_i)$$

  - Fit a weak learner $\overline{y}_k(\mathbf{x})$ to negative gradients, considering dataset $(\mathbf{x}_i, -g(\mathbf{x}_i))$, $i = 1, \ldots, n$
  - Derive the new classifier $y_{k+1}(\mathbf{x}) = y_k(\mathbf{x}) + \overline{y}_k(\mathbf{x})$

Gradient boosting for regression

- 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.
- Why do we need to consider other loss functions? Isn't square loss good enough?
- 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 $L(t, y) = |t - y|$: $-g(\mathbf{x}_i) = \text{sgn}(t_i - y(\mathbf{x}_i)$
  - Huber loss

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

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

Gradient boosting for classification

Consider a K-multiclass framework

- Set $y_1^{(j)}(\mathbf{x}) = \frac{1}{K}$, for $j = 1, \ldots, K$
- For $k = 1, \ldots, m$:
  - Compute negative gradients

$$-g^{(j)}(\mathbf{x}_i) = -\frac{\partial R}{\partial y^{(j)}(\mathbf{x}_i)}$$

    for $j = 1, \ldots, K$
  - Fit K weak learners $\overline{y}_k^{(j)}(\mathbf{x})$ $(j = 1, \ldots, K)$ to negative gradients, considering dataset $(\mathbf{x}_i, -g(\mathbf{x}_i))$, $i = 1, \ldots, n$
  - Derive the new classifiers $y_{k+1}^{(j)}(\mathbf{x}) = y_k^{(j)}(\mathbf{x}) + \overline{y}_k^{(j)}(\mathbf{x})$