Ensemble methods

Course of Machine Learning
Master Degree in Computer Science

University of Rome “Tor Vergata”

a.a. 2019-2020

Giorgio Gambosi

Ensemble methods

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)

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

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

Bootstrap

- 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 $\hat{\mathcal{F}}$. 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. Bootstrap sample from training set
    2. Grow a decision tree T_b on such data by performing the following operations for each node:
        1. select m variables at random
        2. pick the best variable among them
        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

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

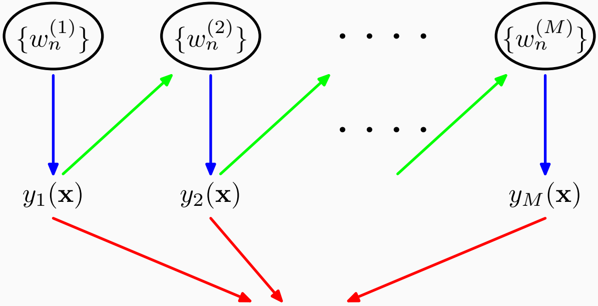
- Each $\alpha_j > 0$ is computed by the boosting algorithm and reflects how accurately y_m classified the data.

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



$$Y_M(\mathbf{x}) = \operatorname{sign}\left(\sum_{m}^{M} \alpha_m y_m(\mathbf{x}) \right)$$

Adaboost

Binary classification, dataset $(\mathbf{X}, \mathbf{t})$ of size n, with $t_i \in \{-1, 1\}$. The algorithm maintains a 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 $i = 1, \ldots, n$
- For $j = 1, \ldots, m$:
  - Train a weak learner $y_j(\mathbf{x})$ on $\mathbf{X}$ in such a way to minimize the 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 $\frac{w_i^{(j)}}{\sum_i w_i^{(j)}}$

Adaboost

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

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

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

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

Adaboost

The overall prediction is

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

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.

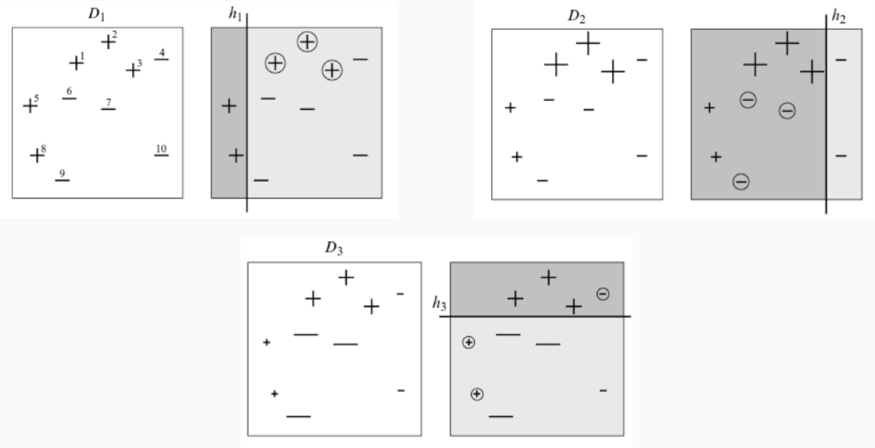
Adaboost

- Observe that a weak learner confidence is inversely related to the probability of misclassification. Moreover,

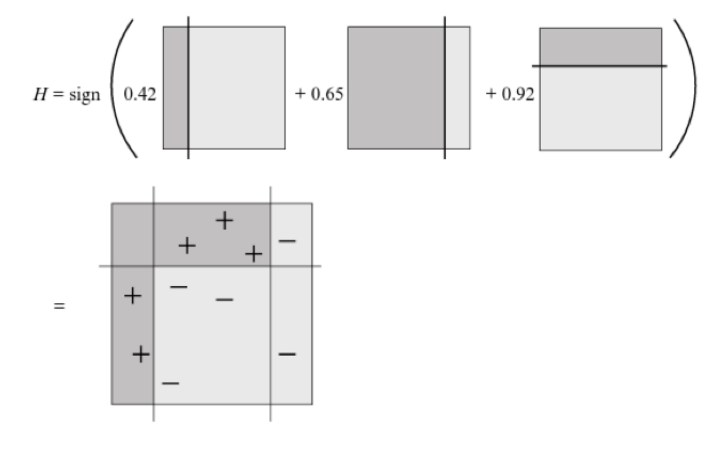
$$w_i^{(t)} = \frac{1}{n}\prod_{j\in\mathcal{B}_i}\frac{1-e^{(j)}}{e^{(j)}}$$

  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.

Adaboost



Adaboost



Additive models

- Additive models are defined as the additive composition of simpler "base" predictors

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

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

Fitting additive models

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

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

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

Forward stagewise additive modeling

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

- Set $y_0(\mathbf{x}) = 0$
- For $k = 1, \ldots, m$:
    - Compute
    $$(\hat{c}_k, \hat{\mathbf{w}}_k) = \underset{c_k, \mathbf{w}_k}{\operatorname{argmin}} \sum_{i=1}^{n} L(t_i, y_{k-1}(\mathbf{x}_i) + c_k \overline{y}(\mathbf{x}_i; \mathbf{w}_k))$$
    - Set $y_k(\mathbf{x}) = y_{k-1}(\mathbf{x}) + \hat{c}_k \overline{y}(\mathbf{x}; \hat{\mathbf{w}}_k)$

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

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

that is, minimizing

$$\underset{\alpha_j, \mathbf{w}_j}{\operatorname{argmin}} \sum_{i=1}^{n} e^{-t_i \sum_{k=1}^{m} \alpha_k \overline{y}(\mathbf{x}_i; \mathbf{w}_k)}$$

Adaboost as additive model

Applying forward stagewise additive modeling, at each step k it computes

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

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

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

Gradient boosting

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.

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 $t_i - y^{(1)}(\mathbf{x}_i) = t_i - y_i^{(1)}$
- A new dataset $(\mathbf{x}_i, t_i - y_i^{(1)})$, $i = 1, \ldots, n$ can be defined, 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?

- 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 $y_i, \ldots, y_n$
- 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 $\overline{y}(\mathbf{x})$ can then be derived by considering the dataset

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

Gradient boosting for regression

The following algorithm results

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

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

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

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

Gradient boosting for classification

Consider a K-class framework.

Again, let $R = \sum_{i=1}^{n} L(t_i, y_1(\mathbf{x}_i), \ldots, y_K(\mathbf{x}_i))$ for a given loss function

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

$$-g_j^{(k)}(\mathbf{x}_i) = -\frac{\partial R}{\partial y_j^{(k)}(\mathbf{x}_i)} = -\frac{\partial}{\partial y_i} L(t_i, y_1(\mathbf{x}_i), \ldots, y_K(\mathbf{x}_i))\Big|_{y_i = y^{(k)}(\mathbf{x}_i)}$$
$$= -\frac{\partial}{\partial y_j^{(k)}(\mathbf{x}_i)} L(t_i, y_1(\mathbf{x}_i), \ldots, y_K(\mathbf{x}_i))$$

    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 $y_{k+1}^{(j)}(\mathbf{x}) = y_k^{(j)}(\mathbf{x}) + \overline{y}_k^{(j)}(\mathbf{x})$

Which weak learners?

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