Review on boosting algorithms

VU Tuan Hung and DO Quoc Khanh

10 mai 2012

1 Introduction

2 Two-class classification

In this first part, we present an overview on boosting methods in the two-class classification framework. From a *training* set $(\mathbf{x}_i, y_i)_{i=1,\ldots,N}$ in which $\mathbf{x}_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$, we try to construct a function $F : \mathcal{X} \to \mathcal{Y}$ so that when a new value $\mathbf{x}$ is randomly introduced, we have the highest probability to predict correctly the value y corresponding to this value of $\mathbf{x}$. Formally, we want to minimize the probability :

$$\mathbb{P}_{(\mathbf{x},y)} \left(y \neq F(\mathbf{x}) \right)$$

The variable $\mathbf{x}$ is called explanatory variables ($\mathbf{x}$ may be multi-variational) and y is called response variable. In the two-class classification framework, $\mathcal{Y} = \{-1, 1\}$.

2.1 Boosting and optimization in function space

We exploit the point of view presented in [2] by considering this problem as an estimation and optimization in function space. Indeed, if there exists a function F^* which minimizes the above error :

$$\begin{aligned} F^* &= arg \min_F \mathbb{P}_{(\mathbf{x},y)}(y \neq F(\mathbf{x})) \\ &= arg \min_F \mathbb{E}_{(\mathbf{x},y)} \left[1(y \neq F(\mathbf{x})) \right] \end{aligned}$$

then we are trying to estimate F^* by a function $\hat{F}$ through the training set $(\mathbf{x}_i, y_i)_{i=1,\ldots,N}$.

Base classifiers. An approach frequently employed by classification algorithms is to suppose F^* belongs to a function class parameterized by $\theta \in \Theta$:

$$F^* \in \mathcal{Q} = \{F(.,\theta) | \theta \in \Theta\}$$

so that the problem of estimating F^* becomes an optimization of the parameters on Θ :

$$\hat{\theta} = arg \min_{\theta \in \Theta} \mathbb{E}_{(\mathbf{x},y)} \left[1(y \neq F(\mathbf{x}, \theta)) \right]$$

and then we will take $\hat{F} = F(., \hat{\theta}) \in \mathcal{Q}$. For example, with regression tree algorithms, we have :

$$\mathcal{Q} = \left\{ F(x, \theta) = \sum_{k=1}^{K} \lambda_k 1(\mathbf{x} \in R_k) | (\lambda_1, \ldots, \lambda_K) \in \mathbb{R}^K, (R_1, \ldots, R_K) \in \mathcal{P}_{\mathcal{X}} \right\}$$

in which $\theta = (\lambda_{1:K}, R_{1:K})$ and $\mathcal{P}_{\mathcal{X}}$ is the set of all partitions of $\mathcal{X}$ into K disjoint subsets by hyperplans which are orthogonal to axes. Similarly for support vector machines, K disjoint subsets R_1,...,R_K are divided by hyperplans in the reproducing kernel Hilbert space of $\mathcal{X}$ corresponding to some kernel.

We can see that a classifier is characterized by its function sub-space $\mathcal{Q}$ and the corresponding parameter space. Having the base classifiers $\mathcal{Q}_{1:M}$ with parameter spaces $\Theta_{1:M}$, instead of considering each of these classifiers separately, boosting methods consider functions of the following additive form :

$$\hat{F} \in \mathcal{F}_{\mathcal{Q}_1,...,\mathcal{Q}_M} = \left\{ \sum_{m=1}^{M} \beta_m f(.,\theta_m) | \theta_m \in \Theta_m, \forall m = 1,...,M \right\}$$

so that the optimization problem becomes :

$$\left\{ \hat{\beta}_{1:M}, \hat{\theta}_{1:M} \right\} = arg \min_{\beta \in \mathbb{R}^M, \theta_{1:M} \in \Theta_{1:M}} \mathbb{E}_{(\mathbf{x},y)} \left[1(y \neq F(\mathbf{x}; \beta_{1:M}, \theta_{1:M})) \right] \tag{1}$$

Friedman, J. and Hastie, T. in [1] explained boosting as a forward stepwise algorithm for resolve the optimization problem (1). Friedman, J. in [2] considered boosting like optimization algorithm in function space. We will try to adopt the latter to explain all mentioned boosting algorithms. Before going into greater details, we remark that as an classification algorithm, the boosting algorithm has its corresponding function subset which is $\mathcal{F} = \mathcal{F}_{\mathcal{Q}_1,...,\mathcal{Q}_M} = \sum\limits_{m=1}^{M} \mathcal{Q}_m$ so much larger than function subset of all base classifiers, explaining the dominating performance of boosting compared to its base classifiers.

Loss function. We remark that the binary loss function $1(y \neq F(\mathbf{x}))$ is not the only function that reflects the difference between y and $F(\mathbf{x})$. In machine learning, one has other loss functions that are continuous, convex and then easier to do the optimization. For the rest of the report, we use in general $L(y, F(\mathbf{x}))$ to indicate this function.

Optimization on training set. One difficulty is that we can not evaluate the distribution of $(\mathbf{x}, y)$ and calculate the expectation in the right hand side formula of (1). Instead, we only want to optimize on the training data, which means the following optimization problem :

$$F^* = arg \min_F \sum_{i=1}^{N} L(y_i, F(x_i))$$

Put $Q(F) = \sum\limits_{i=1}^{N} L(y_i, F(\mathbf{x}_i))$, we remark that $Q(F)$ depends only on N values of the function F at $(\mathbf{x}_1,...,\mathbf{x}_N)$. We denote for each function F in the function space, a corresponding vector $\overline{F} \in \mathbb{R}^N$ so that $\overline{F} = (F(\mathbf{x}_1),...,F(\mathbf{x}_N))$, and we consider the relaxation problem on vector space $\mathbb{R}^N$: $\overline{F^*} = arg \min\limits_{\overline{F} \in \mathbb{R}^N} Q(\overline{F})$. We try to resolve this problem by recursive numerical methods. Suppose that at $m - 1^{th}$ step we obtain a value $\overline{F}_{m-1}$. By numerical methods (Newton-Raphson, algorithm of gradient descent etc.) we find a direction of descent $d_m \in \mathbb{R}^N$ and a coefficient c_m so that if we put $\overline{F}_m = \overline{F}_{m-1} + c_m.d_m$, then $Q(\overline{F}_m) \leq Q(\overline{F}_{m-1})$. But we can not use the direction d_m directly in the original problem with functions F_m because $\overline{F}_m$ identifies the values of functions only at N points. Instead, we have to find a regression function near to the direction d_m ; it means if we use a function subspace $\mathcal{Q}_m$ at m^{th} step, then we have to solve :

$$\{f_m, c\} = arg \min_{f_m \in \mathcal{Q}_m, c \in \mathbb{R}_+} \|d_m - c.\overline{f_m}\|^2$$

in which $\overline{f_m}$ is the vector of values of f_m at $(\mathbf{x}_1,...,\mathbf{x}_N)$. After that, we have to look for a coefficient β_m so that, if F_{m-1} is the function obtained at the precedent step, then $Q(F_m) \leq Q(F_{m-1})$

with $F_m = F_{m-1} + \beta_m f_m$. If we start with $F_0 = 0$ then we obtain at M^{th} the additive form $F_M = \sum_{m=1}^{M} \beta_m f_m \in \mathcal{F} = \sum_{m=1}^{M} \mathcal{Q}_m$. We summary this generic algorithm in the following table.

Generic Algorithm for Boosting
1. Start with $F_0(\mathbf{x}) = 0$.
2. Repeat for $m = 1,2,...,M$:
(a) Search for a descent direction $d_m \in \mathbb{R}^N$ by some Newton-like numerical algorithm of optimization in $\mathbb{R}^N$.
(b) Solve $\{f_m, c\} = arg \min_{f_m \in \mathcal{Q}_m, c \in \mathbb{R}_+} \|d_m - c.\overline{f_m}\|^2$. The least-square criterion is not mandatory.
(c) Search for a coefficient β_m so that $Q(F_{m-1} + \beta_m f_m) \leq Q(F_{m-1})$, a line-search strategy can be used.
(d) $F_m = F_{m-1} + \beta_m f_m$.
3. Conclude with $F(\mathbf{x})$.

In most cases, the direction of descent will be calculated from the gradient of $Q(\overline{F})$ at $\overline{F} = \overline{F}_{m-1}$ according to Newton-like optimization algorithms in $\mathbb{R}^N$:

$$d_m = -\frac{\partial Q}{\partial \overline{F}}(\overline{F}_{m-1}) = -\left(\frac{\partial L}{\partial \overline{F}_1}(y_1, \overline{F}_1), ..., \frac{\partial L}{\partial \overline{F}_N}(y_N, \overline{F}_N)\right) |_{\overline{F}_{1:N} = \overline{F}_{m-1}}$$

After having found f_m, the coefficient β_m is often evaluated by line-search procedure $\beta_m = arg \min_{\beta \in \mathbb{R}} Q(F_{m-1} + \beta f_m)$ in which F_{m-1} and f_m are functions from $\mathcal{X}$ to $\mathcal{Y}$. In the next sections, we will present and explain boosting algorithms following the same paradigm described above.

2.2 $L(y, F(\mathbf{x})) = e^{-yF(\mathbf{x})}$

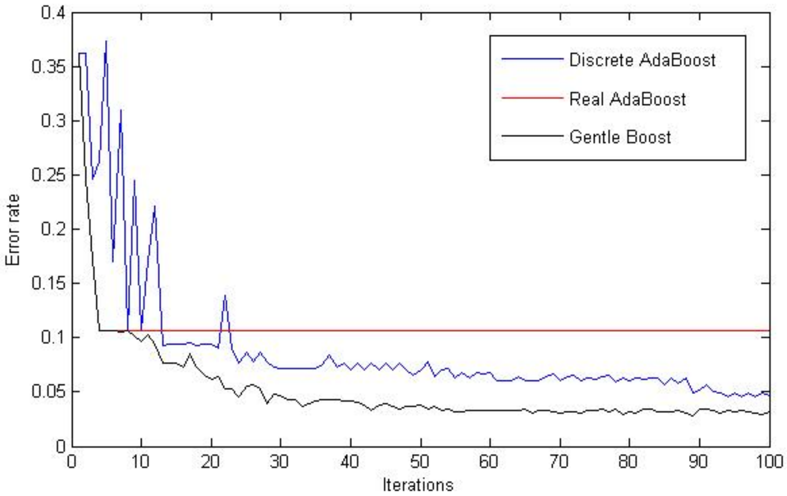

FIGURE 1 – Error rates corresponding to *Discrete AdaBoost*, *Real AdaBoost* and *Gentle AdaBoost*.

This loss function is used in some of the most popular boosting algorithms like *Discrete AdaBoost* or *Real AdaBoost* ; we use it as an example for our model. Firstly the direction descent at m^{th} step $d_m = \nabla_{\overline{F}} Q(\overline{F})|_{\overline{F}=\overline{F}_{m-1}} = \left(y_i e^{-y_i F_{m-1}(x_i)}\right)_{i=1:N}$. Following 2(b), we look for $f_m \in \mathcal{Q}_m$ and $c > 0$ which minimize :

$$S(f_m, c) = \sum_{i=1}^{N} \left(y_i e^{-y_i F_{m-1}(\mathbf{x}_i)} - c f_m(\mathbf{x}_i)\right)^2 \tag{2}$$

If we precise $\mathcal{Q}_m$ so that $f_m(\mathbf{x}) \in \{-1, 1\}, \forall \mathbf{x}$, we have :

$$S(f_m, c) = Nc^2 + \sum_{i=1}^{N} e^{-2y_i F_{m-1}(\mathbf{x}_i)} - 2c \sum_{i=1}^{N} e^{-y_i F_{m-1}(\mathbf{x}_i)} + c \sum_{i=1}^{N} e^{-y_i F_{m-1}(\mathbf{x}_i)} (y_i - f_m(\mathbf{x}_i))^2$$

so if we put $w_i = e^{-y_i F_{m-1}(\mathbf{x}_i)}$, we have to solve $arg \min\limits_{f_m \in \mathcal{Q}_m} \sum\limits_{i=1}^{N} w_i (y_i - f_m(\mathbf{x}_i))^2$ which is equivalent to classification of $\mathbf{x}_i$ using weights w_i. The line 2(c) is equivalent to :

$$\begin{aligned}
\beta_m &= arg \min_{\beta \in \mathbb{R}} \sum_{i=1}^{N} e^{-y_i (F_{m-1}(\mathbf{x}_i) + \beta f_m(\mathbf{x}_i))} \\
&= arg \min_{\beta \in \mathbb{R}} \sum_{i=1}^{N} w_i e^{-\beta y_i f_m(\mathbf{x}_i)} \\
&= arg \min_{\beta \in \mathbb{R}} \left(e^{-\beta} \times \sum_{y_i = f_m(\mathbf{x}_m)} w_i + e^{\beta} \times \sum_{y_i \neq f_m(\mathbf{x}_i)} w_i \right) \\
&= \frac{1}{2} \log \left(\frac{\sum\limits_{y_i = f_m(\mathbf{x}_i)} w_i}{\sum\limits_{y_i \neq f_m(\mathbf{x}_i)} w_i} \right)
\end{aligned}$$

we obtain Discrete AdaBoost algorithm described in the table below.

```
Discrete AdaBoost
1. Start with weights $w_i = 1/N, i = 1,...,N$.
2. Repeat for $m = 1,2,...,M$ :
(a) Fit the classifier $f_m(x) \in \{-1, 1\}$ using weights $w_i$ on the training data.
(b) Compute $err_m = \mathbb{E}_w \left[1(y \neq f_m(\mathbf{x}))\right]$, and $\beta_m = \log\left(\frac{1 - err_m}{err_m}\right)$.
(c) Set $w_i \leftarrow w_i \times e^{\beta_m \times 1(y_i \neq f_m(\mathbf{x}_i))}$, $i = 1,...,N$, and renormalize $w$.
3. Output the classifier $sign\left[\sum_{m=1}^{M} \beta_m f_m(\mathbf{x})\right]$.
```

Now we extend the subset $\mathcal{Q}_m$ to contain f_m of real values and not only the functions with discrete values $\{-1, 1\}$. *Real AdaBoost* is a boosting algorithm that uses such base classifiers. In order to understand this algorithm, we note that *Real AdaBoost* does not use gradient as the direction of descent d_m, but find it directly from an optimization problem. The coefficient of descent β_m is taken value 1. Firstly, d_m is calculated by :

$$d_m = arg \min_{d \in \mathbb{R}^N} Q(\overline{F}_{m-1} + d) = arg \min_{d \in \mathbb{R}^N} \sum_{i=1}^{N} e^{-y_i (F_{m-1}(\mathbf{x}_i) + d^{(i)})}$$

In order to calculate exactly d_m as an estimate of some added regression function f_m's values, we remark that in $(\mathbf{x}_i, y_i)$, there may be many $\mathbf{x}_i$ taking a same values, so $(F(\mathbf{x}_i))_{i=1:N}$ and then

d_m may not be real vectors in $\mathbb{R}^N$. By simplicity, we parameterize d_m by $\mathbf{x}$ (which are different values in $(\mathbf{x}_i)_{i=1:N}$) and not by $i = 1 : N$. We have :

$$d_m(\mathbf{x}) = arg \min_{d(\mathbf{x})\in\mathbb{R}} \sum_{\mathbf{x}_i=\mathbf{x}} e^{-y_i(F_{m-1}(\mathbf{x}_i)+d(\mathbf{x}))}$$

We imply that $d_m(\mathbf{x}) = \frac{1}{2}\log\left(\frac{\sum_{\mathbf{x}_i=\mathbf{x},y_i=1} w_i}{\sum_{\mathbf{x}_i=\mathbf{x},y_i=-1} w_i}\right)$. We do not use the least-square criterion to find a regression function in $\mathcal{Q}_m$ that approximates d_m (line 2(b) of the Generic Algorithm). Instead, we use a comment that, $d_m(\mathbf{x})$ is in fact an empirical estimation of the quantity $\frac{1}{2}\log\left(\frac{\mathbb{P}_w(y=1|\mathbf{x})}{\mathbb{P}_w(y=-1|\mathbf{x})}\right)$ after having trained $\mathbf{x}$ and y on training set $(\mathbf{x}_i, y_i)_{i=1:N}$ with weights $w_i, i = 1,...,N$. The coefficient β_m is taken 1. We have the following *Real AdaBoost* algorithm.

```
Real AdaBoost
1. Start with weights $w_i = 1/N, i = 1,...,N$.
2. Repeat for $m = 1,2,...,M$ :
(a) Fit the classifier to obtain a class probability estimate $p_m(\mathbf{x}) = \hat{P}_w(y = 1|\mathbf{x}) \in [0, 1]$, using weights $w_i$ on the training data.
(b) Set $f_m(\mathbf{x}) = \frac{1}{2}\log\frac{p_m(\mathbf{x})}{1-p_m(\mathbf{x})}$.
(c) Update $w_i \leftarrow w_i e^{-y_i f_m(\mathbf{x}_i)}$ and renormalize.
3. Output the classifier $sign\left[\sum_{m=1}^{M} f_m(\mathbf{x})\right]$.
```

The last algorithm that we want to present in this section is *Gentle AdaBoost*. This algorithm use $\beta_m = 1$ but d_m is not calculated from a direct optimization but from the stepping of Newton-Raphson algorithm. Like in *Real AdaBoost*, we parameterize d_m not by $i = 1 : N$ but by $\mathbf{x}$ which are the differents values in $(\mathbf{x}_i)_{i=1:N}$. We note that the algorithms with such parameterization of d_m is called **population version**. More precisely,

$$\begin{aligned} d_m(\mathbf{x}) &= -\left(\frac{\partial^2 Q}{\partial F(\mathbf{x})^2}\right)^{-1} \frac{\partial Q}{\partial F(\mathbf{x})}|_{F(\mathbf{x})=F_{m-1}(\mathbf{x})} \\ &= \frac{\sum_{\mathbf{x}_i=\mathbf{x}} y_i e^{-y_i F(\mathbf{x}_i)}}{\sum_{\mathbf{x}_i=\mathbf{x}} e^{-y_i F(\mathbf{x}_i)}} = \frac{\sum_{\mathbf{x}_i=\mathbf{x}} w_i y_i}{\sum_{\mathbf{x}_i=\mathbf{x}} w_i} \end{aligned}$$

We recognize that $d_m(\mathbf{x})$ is in fact an empirical estimation of the quantity $\mathbb{E}_w(y|\mathbf{x})$; therefore we take $f_m(\mathbf{x})$ to be the regression of y to $\mathbf{x}$ with weights w_i on the training data. We have the following *Gentle AdaBoost* algorithm.

```
Gentle AdaBoost
1. Start with weights $w_i = 1/N, i = 1,...,N$.
2. Repeat for $m = 1,2,...,M$ :
(a) Fit the regression function $f_m(\mathbf{x})$ by weighted least-squares of $y_i$ to $\mathbf{x}_i$ with weights $w_i$.
(b) Update $F(\mathbf{x}) \leftarrow F(\mathbf{x}) + f_m(\mathbf{x})$.
(c) Update $w_i \leftarrow w_i e^{-y_i f_m(\mathbf{x}_i)}$ and renormalize.
3. Output the classifier $sign\,[F(\mathbf{x})] = sign\left[\sum_{m=1}^{M} f_m(\mathbf{x})\right]$.
```

Experiments : Here we compare the performance of these three algorithms on a same simulated dataset. The dataset is simulated following the following rules : $\mathbf{x} \in \mathbb{R}^2$ follows uniform distribution in $(0,1)^2$, and $y_i = 2 * 1((\mathbf{x}_i^{(1)} - 0.5)^2 + (\mathbf{x}_i^{(2)} - 0.5)^2 > 1/6) - 1$. The Bayes error is therefore 0. We can see that the *Gentle AdaBoost* obtain the best performance in this experiment. The *Gentle AdaBoost* corresponds to two-degree optimization in function space (Newton-Raphson algorithm) while the *Discrete AdaBoost* corresponds to one-degree method. We can link this result to the dominating convergence rate of two-degree methods in numerical optimization.

2.3 $L(y, F(\mathbf{x})) = (y - F(\mathbf{x}))^2/2$

In this section, we will use least-square as the loss function. The updating terms of F_m in the line 2(d) of our generic algorithm will be specifically derived. The least-square loss function is defined as : $L(y,F) = \dfrac{(y-F)^2}{2}$. Now, we can determine the negative gradient, or, the descent direction $d_m = -\dfrac{\partial Q}{\partial \overline{F}}|_{\overline{F}=F_{m-1}} = y - F_{m-1}$. If we apply line search for 2(c), it turns out that $\beta_m = c$. Indeed, with finite data, to do the line search is to find β_m such that :

$$\begin{aligned}
\beta_m &= arg \min_\beta \sum_{i=1}^N Q(y_i, F_{m-1}(x_i) + \beta \overline{f_m}) \\
&= arg \min_\beta \sum_{i=1}^N \frac{(y_i - F_{m-1}(x_i) - \beta\overline{f_m})^2}{2} \\
&= arg \min_\beta \sum_{i=1}^N \frac{(d_m - \beta\overline{f_m})^2}{2} \\
&= arg \min_\beta \|d_m - \beta\overline{f_m}\|^2 = c
\end{aligned}$$

So, with least-square loss function, the generic algorithm ends up the following LS_Boost algorithm

```
Least-Square Boost
1. Start with $F_0(\mathbf{x}) = 0$.
2. Repeat for $m$ = 1,2,...,$M$ :
(a) $\overline{y}_i = y_i - F_{m-1}(\mathbf{x}_i), i = 1 : N$
(b) Solve $\{f_m, \beta_m\} = arg\_\min_{f_m \in \mathcal{Q}_m, \beta\in\mathbb{R}} \|\overline{y} - \beta\overline{f_m}\|^2$.
(c) $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \beta_m f_m(\mathbf{x})$.
3. Output the classifier $sign\,[F(\mathbf{x})]$.
```

2.4 $L(y, F(\mathbf{x})) = |y - F(\mathbf{x})|$

With $L(y,F(\mathbf{x})) = |y - F(\mathbf{x})|$, we choose $d_m^{(i)} = \dfrac{\partial Q}{\partial \overline{F}_i}|_{\overline{F}=\overline{F}_{m-1}} = sign(y_i - F_{m-1}(\mathbf{x}_i)), \forall i = 1 : N$. We can always follow the generic algorithm presented in 2.1 with all kinds of base classifiers. Here we discuss on how to adapt regression tree to our boosting algorithm, because it will be the only base classifier used in our experiments.

Firstly, we remark that the coefficient $c > 0$ in the line 2(b) of our generic algorithm does not change fondamentally the properties of regression trees f_m ; it suffices to get regression tree f_m to approximate the direction d_m. A regression tree f_m divide $\mathcal{X}$-space into K disjoint subspace $R_{m,k}$ so that $\bigcup\limits_{k=1}^{K} R_{m,k} = \mathcal{X}$. For $\mathbf{x}$ belonging to each of these areas, $f_m(\mathbf{x})$ takes a value $\lambda_{m,k}$. We can rewrite $f_m(\mathbf{x}) = \sum\limits_{k=1}^{K} \lambda_{m,k} 1(\mathbf{x} \in R_{m,k})$. The constant β_m in the line 2(c) changes these values $\lambda_{m,k}$ by a same way. Another method which gives a better optimization than line 2(c) is to modify all regression values $\lambda_{m,k}$ in this phase and not only the parameter β_m, that means :

$$\{\lambda_{m,k}\}_{k=1:K} = arg \min_{\lambda \in \mathbb{R}^K} Q(F_{m-1} + \sum_{k=1}^{K} \lambda_k 1(\mathbf{x} \in R_{m,k}))$$

This method is clearly more powerful than optimization only on β_m and is in fact operated separately in each $R_{m,k}$: For each $k = 1 : K$,

$$\lambda_{m,k} = arg \min_{\lambda \in \mathbb{R}} \sum_{\mathbf{x}_i \in R_{m,k}} L(y_i, F_{m-1}(\mathbf{x}_i) + \lambda) \tag{3}$$

Finally, the areas $R_{m,k}$ are divided accordint to the regression in line 2(b), which tries to approximate d_m by f_m (as explained, we have taken $c = 1$).

For the absolute loss function, according to (3), we have $\lambda_{m,k} = arg \min\limits_{\lambda \in \mathbb{R}} |y_i - F_{m-1}(\mathbf{x}_i) - \lambda| = \text{median}\{y_i - F_{m-1}(\mathbf{x}_i)\}_{i, \mathbf{x}_i \in R_{m,k}}$, $\forall k = 1 : K$. We obtain *LAD TreeBoost* algorithm described below.

```
LAD TreeBoost
1. Start with $F_0(\mathbf{x}) = 0$.
2. Repeat for $m = 1,2,...,M$ :
(a) $\overline{y}_i = sign(y_i - F_{m-1}(\mathbf{x}_i)), i = 1 : N$.
(b) $\{R_{m,k}\}_{k=1:K}$ is the $K$-terminal node tree regression of $\overline{y}_i$ on $\mathbf{x}_i, i = 1 : N$.
(c) $\lambda_{m,k} = \text{median}\{y_i - F_{m-1}(\mathbf{x}_i)\}_{i, \mathbf{x}_i \in R_{m,k}}$ for $k = 1 : K$.
(d) $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \sum_{k=1}^{K} \lambda_{m,k} 1(\mathbf{x} \in R_{m,k})$.
3. Output the classifier $sign\,[F(\mathbf{x})]$.
```

2.5 M-regression loss function

We consider here the Huber loss function, which is defined as :

$$L(y, F(\mathbf{x})) = \begin{cases} \frac{(y - F(\mathbf{x}))^2}{2} & \text{if } |y - F(\mathbf{x})| \leq \delta \\ \delta(|y - F(\mathbf{x})| - \delta/2) & \text{if } |y - F(\mathbf{x})| > \delta \end{cases}, \delta \in \mathbb{R}^+$$

«««< .mine

Then, we have the negative gradient :

$$d_m^{(i)} = -\frac{\partial Q}{\partial \overline{F}_i}|_{\overline{F} = F_{m-1}} = \begin{cases} y_i - F_{m-1} & \text{if } |y_i - F_{m-1}(\mathbf{x}_i)| \leq \delta \\ \delta.\text{sign}(y_i - F_{m-1}(\mathbf{x}_i)) & \text{if } |y_i - F_{m-1}(\mathbf{x}_i)| > \delta \end{cases}, \forall i = 1 : N$$

As a significant parameter of the loss function, δ should be chosen considerately. Before going into detail, we should observe that $d_m^{(.)}$ is constrained by δ. The meaning of using δ here is to remove outliers. Indeed, those points, which have big difference between target output y and estimated output $F_{m-1}(\mathbf{x})$, will be excluded from updating process. From now on, we should remark M-Regression's characteristic of resisting to long-tailed error distribution and outliers. The value of δ will be redetermined iteratively, based on the distribution of residuals $y - F^*$, with F^* as the true target function. Normally, at m^{th} iteration, we often choose $\delta = \delta_m$ as α-quantile of current approximate absolute residual distribution $|y - F_{m-1}|$. Then, the factor is determined by $\delta_m = \text{quantile}_\alpha(|y - F_{m-1}|)$.

If we use regression trees as base classifiers, the results in 2.4 could be exploited. Let's consider $(R_{m,1}, R_{m,2},..., R_{m,K})$ as the partition of $\mathcal{X}$-space, determined by the regression tree at m^{th} iteration. The updating equation then becomes $F_m = F_{m-1} + \sum_{k=1}^{K} \lambda_{m,k} 1(\mathbf{x} \in R_{m,k})$ with

$$\lambda_{m,k} = arg \min_{\lambda} \sum_{\mathbf{x}_i \in R_{m,k}} L(y_i, F_{m-1}(\mathbf{x}_i + \lambda))$$

One approximation of $\lambda_{m,k}$ is given in [ref], which requires the median of the current residual $\tilde{r}_{m,k} = median_{\mathbf{x}_i \in R_{m,k}} (\underbrace{y_i - F_{m-1}(\mathbf{x}_i)}_{r_{m-1}(\mathbf{x}_i)}), \forall k = 1 : K$. The approximate $\lambda_{m,k}, \forall k = 1 : K$ is then computed by :

$$\lambda_{m,k} = \tilde{r}_{m,k} + \frac{1}{N_{m,k}} \sum_{\mathbf{x}_i \in R_{m,k}} sign(r_{m-1}(\mathbf{x}_i) - \tilde{r}_{m,k}) \cdot min(\delta_m, |r_{m-1}(\mathbf{x}_i) - \tilde{r}_{m,k}|)$$

Now, we obtain an boosting algorithm using regression tree and Huber loss function, which is called M-TreeBoost

```
M-TreeBoost
1. Start with $F_0(\mathbf{x}) = 0$.
2. Repeat for $m = 1,2,...,M$ :
(a) $r_{m-1}(\mathbf{x}_i) = y_i - F_{m-1}(\mathbf{x}_i), \forall i = 1 : N$.
(b) $\delta_m = \text{quantile}_\alpha |r_{m-1}(\mathbf{x}_i)|, \forall i = 1 : N$.
(c) $\overline{y}_i = \begin{cases} y_i - F_{m-1}(\mathbf{x}_i) & \text{if} |y_i - F_{m-1}(\mathbf{x}_i)| \leq \delta_m \\ \delta_m.\text{sign}(y_i - F_{m-1}(\mathbf{x}_i)) & \text{if} |y_i - F_{m-1}(\mathbf{x}_i)| > \delta_m \end{cases}$
(d) $\{R_{m,k}\}_{k=1:K}$ is the $K$-terminal node tree regression of $\overline{y}_i$ on $\mathbf{x}_i, i = 1 : N$.
(e) $\tilde{r}_{m,k} = median_{\mathbf{x}_i \in R_{m,k}} (r_{m-1}(\mathbf{x}_i))$
(f) $\lambda_{m,k} = \tilde{r}_{m,k} + \frac{1}{N_{m,k}} \sum_{\mathbf{x}_i \in R_{m,k}} sign(r_{m-1}(\mathbf{x}_i) - \tilde{r}_{m,k}) \cdot min(\delta_m, |r_{m-1}(\mathbf{x}_i) - \tilde{r}_{m,k}|)$.
(g) $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \sum_{k=1}^{K} \lambda_{m,k} 1(\mathbf{x} \in R_{m,k})$.
3. Output the classifier $sign\,[F(\mathbf{x})]$.
```

2.6 Negative Binomial log-likelihood loss function

The binomial log-likelihood loss function is $L(y, F(\mathbf{x})) = \log\left(1 + e^{2F(\mathbf{x})}\right) - 2y^* F(\mathbf{x})$ in which $y^* = \frac{y+1}{2}$. In Bernouilli model, the probability that y^* taking value 1 is $p(\mathbf{x}) = \frac{e^{F(\mathbf{x})}}{e^{F(\mathbf{x})} + e^{-F(\mathbf{x})}}$. The negative log-likelihood is calculated by :

$$\begin{aligned} L(y, F(\mathbf{x})) &= -y^* \log p(\mathbf{x}) - (1 - y^*) \log(1 - p(\mathbf{x})) \\ &= \log\left(1 + e^{2F(\mathbf{x})}\right) + y^* \log \frac{1 + e^{-2F(\mathbf{x})}}{1 + e^{2F(\mathbf{x})}} \end{aligned}$$

We replace the last term simply by $-2y^* F(\mathbf{x})$, and obtain the loss function above.

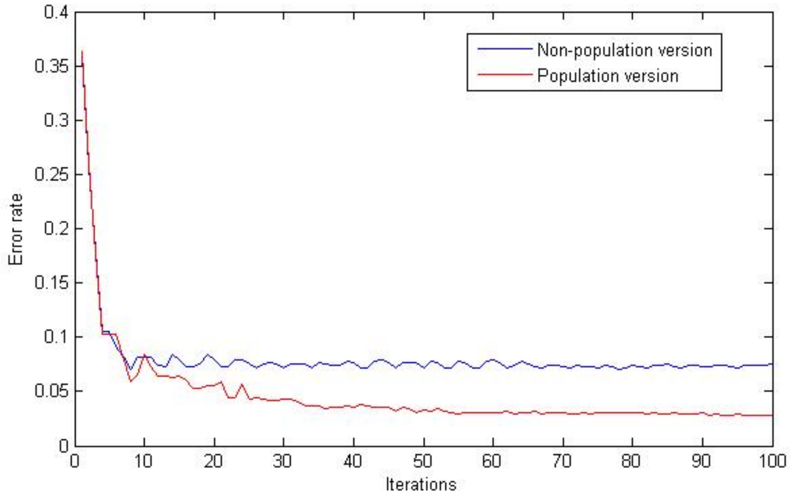

FIGURE 2 – Comparison of the performance of population and non-population version of *Logit Boost* algorithm on simulated data.

The direction of descent d_m is taken to be the Newton-Raphson steps in the **population version** (d_m is parameterized by $\mathbf{x}$). By denoting $p_m(\mathbf{x}_i) = \dfrac{e^{F_{m-1}(\mathbf{x}_i)}}{e^{F_{m-1}(\mathbf{x}_i)} + e^{-F_{m-1}(\mathbf{x}_i)}}$ and by ignoring some multiplying constants in *Hessian* matrix, we have :

$$\begin{aligned} d_m(\mathbf{x}) &= -\left(\frac{\partial^2 Q}{\partial F(\mathbf{x})^2}\right)^{-1} \frac{\partial Q}{\partial F(\mathbf{x})}|_{F=F_{m-1}} \\ &= \frac{\sum\limits_{\mathbf{x}_i=\mathbf{x}} (y_i^* - p_m(\mathbf{x}_i))}{\sum\limits_{\mathbf{x}_i=\mathbf{x}} p_m(\mathbf{x}_i)(1-p_m(\mathbf{x}_i))} \end{aligned}$$

If we put $w_i = p_m(\mathbf{x}_i)(1-p_m(\mathbf{x}_i))$ on the training data, we have $d_m(\mathbf{x}) = \dfrac{\sum\limits_{\mathbf{x}_i=\mathbf{x}} \dfrac{y_i^* - p_m(\mathbf{x}_i)}{p_m(\mathbf{x}_i)(1-p_m(\mathbf{x}_i))} w_i}{\sum\limits_{\mathbf{x}_i=\mathbf{x}} w_i}$. We recognize that this is an empirical estimation of the quantity $\mathbb{E}_w\left[\dfrac{y^* - p(\mathbf{x})}{p(\mathbf{x})(1-p(\mathbf{x}))}|\mathbf{x}\right]$. The coefficient β_m is taken value $\dfrac{1}{2}$. Then we have the following population version of *Logit Boost* algorithm.

```
**Logit Boost (2 classes)**
1. Start with weights $w_i = 1/N$, $F(\mathbf{x}) = 0$ and probability estimates $p(\mathbf{x}_i) = 1/2$.
2. Repeat for $m = 1,2,...,M$ :
(a) Compute $z_i = \dfrac{y_i^* - p(\mathbf{x}_i)}{p(\mathbf{x}_i)(1-p(\mathbf{x}_i))}$ and $w_i = p(\mathbf{x}_i)(1-p(\mathbf{x}_i))$.
(b) Fit the function $f_m(\mathbf{x})$ by a weighted least-squares regression of $z_i$ to $\mathbf{x}_i$ using weights $w_i$.
(c) Update $F(\mathbf{x}) \leftarrow F(\mathbf{x}) + \dfrac{1}{2} f_m(\mathbf{x})$, and $p(\mathbf{x}) = \dfrac{e^{F(\mathbf{x})}}{e^{F(\mathbf{x})} + e^{-F(\mathbf{x})}}$.
3. Output the classifier $sign\,[F(\mathbf{x})] = sign\,[F(\mathbf{x})]$.
```

In the derivation of this *Logit Boost* algorithm, we comment that although $\mathbf{x}_i = \mathbf{x}$, the quantity $p_m(\mathbf{x}_i)(1-p_m(\mathbf{x}_i))$ is not constant. Indeed, in our implementation $p_m(\mathbf{x}_i)$ depends on

$F_{m-1}(\mathbf{x}_i)$ with parameterization by $i = 1 : N$ and not by $\mathbf{x}$. The interpretation of the population version is, although all F_m and f_m is calculated on $\{\mathbf{x}_i\}_{i=1:N}$, their right parameterization in function space is by $\mathbf{x} \in \mathcal{X}$ and not by $i = 1 : N$. The vector d_m parameterized by $\mathbf{x} \in \mathcal{X}$ may be easier to be used in regression of function f_m. This is an advantage of the point of view that considers boosting as an optimization in the function space.

Comparison between population and non-population versions. There exists a **non-population version** of *Logit Boost* that consists on doing regression of f_m **without weights** w. We can derive it by parameterizing vector d_m by $d_m^{(i)}|_{i=1:N}$. We use the same simulated dataset than one used in 2.2 to compare these two versions. The results in Fig. 2 prove the good performance of population version compared to the non-population one.

2.7 $L(y, F(\mathbf{x})) = \log(1 + e^{-2yF(\mathbf{x})})$

For the negative binomial log-likelihood loss function $L(y, F) = log(1 + e^{-2yF})$, we obtain the descent direction $d_m = -\frac{\partial Q}{\partial \overline{F}}|_{\overline{F}=F_{m-1}} = \frac{2y}{1 + e^{2yF_{m-1}}}$. If regression trees are used as base classifiers, we can also inherit results from 2.4, in which the $\lambda_{m,k}$ is now approximated as :

$$\lambda_{m,k} = \frac{\sum\limits_{\mathbf{x}_i \in R_{m,k}} d_m^{(i)}}{\sum\limits_{\mathbf{x}_i \in R_{m,k}} |d_m^{(i)}|(2 - |d_m^{(i)}|)}$$

And, the boosting algorithm is described as in Algorithm **??**

```
L2-TreeBoost
1. Start with $F_0(\mathbf{x}) = 0$.
2. Repeat for $m$ = 1,2,...,$M$ :
(a) $\overline{y}_i = \frac{2y_i}{1 + e^{2y_i F(\mathbf{x}_i)}}$.
(b)$\{R_{m,k}\}_{k=1:K}$ is the $K$-terminal node tree regression of $\overline{y}_i$ on $\mathbf{x}_i, i = 1 : N$.
(c) $\lambda_{m,k} = \frac{\sum_{\mathbf{x}_i \in R_{m,k}} \overline{y}_i}{\sum_{\mathbf{x}_i \in R_{m,k}} |\overline{y}_i|(2 - |\overline{y}_i|)}$
(d) $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \sum_{k=1}^{K} \lambda_{m,k} 1(\mathbf{x} \in R_{m,k})$.
3. Output the classifier $sign\,[F(\mathbf{x})]$.
```

3 Multi-class classification and some generalizations

In this part, we present some generalizations of two-class classification algorithms to the case of multi-classes. In a multi-class classification problem, there are $J > 2$ labels for each line of explanatory variables $\mathbf{x}$, denoted by 1,2,...,J. The response variable y is no longer a vector $\{y_i\}_{i=1:N}$ but a matrix $\{y_{i,j}\}_{1 \le i \le N, 1 \le j \le J}$ defined by :

$$y_{i,j} = \begin{cases} 1 & \text{if label of } \mathbf{x}_i \text{ is } j, \\ -1 & \text{if otherwsise} \end{cases}$$

If we combine the explanatory variables $\mathbf{x}$ with each column $y(:, j)$ of y, we will obtain J separate two-class classification problems. The algorithms for multi-class case will be based on these J problems that we have tried to resolve in the precedent parts of this report.

Another advantage of this representation of y as a matrix $N \times J$ rather than as a N-vector $\in \{1,2,...,J\}^N$ is that we can use it (or some of its variants) to solve more general problems. Schapire, R.E. & Singer, Y. in [3] discussed about some of these generalizations, for example multi-labels problem (in which each $\mathbf{x}$-data can corresponds to more than one labels), and label-ranking problem (in which we try to assess a rank on all labels so that the first one is the most likely to be a label of data $\mathbf{x}$). We also realize that the methods presented below can be adapted to resolve multi-label problems, because that only consists on deciding value of $y_{i,j}$ for $i = 1,...,N$ and $j = 1,...,J$ which is also the goal of multi-class problem. However, in scope of this report, we restrict our discussion inside the multi-single class framework.

3.1 A traditional approach

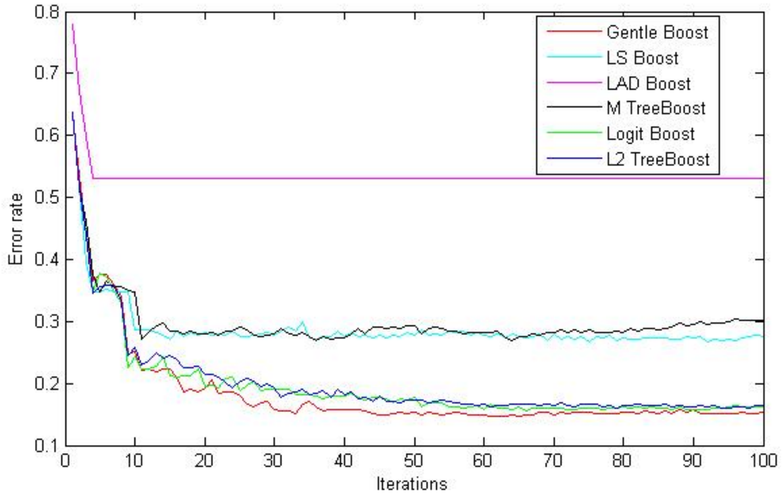

FIGURE 3 – Comparison of the performances of *AdaBoost.MH* using 6 two-class boosting methods, experiments on simulated data with 0% Bayes error.

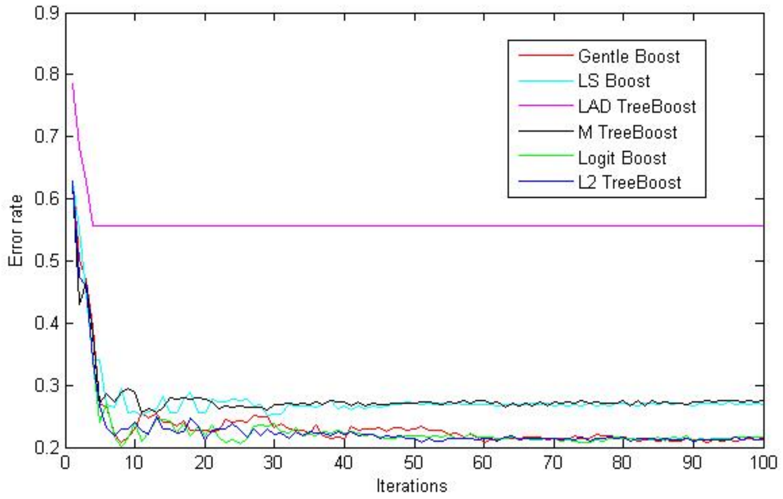

FIGURE 4 – Comparison of the performances of *AdaBoost.MH* using 6 two-class boosting methods, experiments on simulated data with 5% Bayes error.

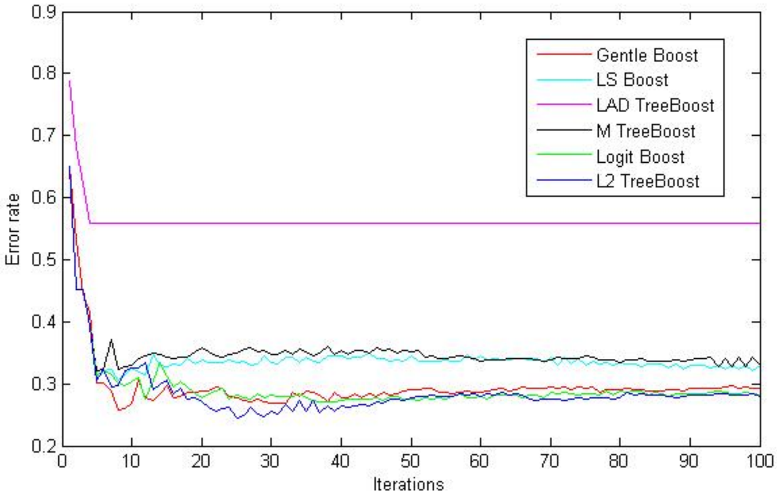

FIGURE 5 – Comparison of the performances of *AdaBoost.MH* using 6 two-class boosting methods, experiments on simulated data with 10% Bayes error.

The *One-Against-All* strategy consists on resolve these J classification problems (described above) separately, then for each data $\mathbf{x}_i$, we choose the result of the problem in which the score of $\mathbf{x}$ is the highest. The *AdaBoost.MH* algorithm is formalized in the following table.

AdaBoost.MH for multi classes
1. Construct response variables y under suitable form ($N \times J$-matrix) using *One-Agaisnt-All* strategy.
2. For $j = 1,...,J$, we output a boosting classifier F_j for problem P_j.
3. Label of $\mathbf{x} = arg \max\limits_{1 \leq j \leq J} F_j(\mathbf{x})$.

Experiments : We can use all two-class boosting methods in the line 2 of the algorithm. Here we use successively *GentleBoost*, *LS Boost*, *LAD TreeBoost*, *M TreeBoost*, *Logit Boost* and *L2 TreeBoost* and compare their performance in multi-class framework on simulated data. The data of J classes is simulated following the rules : $\mathbf{x}_i$ is chosen uniformly in $[0,1]^2$, r_i^2 is the squared distance from $\mathbf{x}_i$ to $(1/2, 1/2)$, $\mathbf{x}_i$ corresponds to label j if $\frac{j-1}{J} \leq \pi r_i^2 < \frac{j}{J}, j =$ 1,2,...,J. We take $J = 4$ and the Bayes error to be 0%, 5% and 10%. The results in Fig. 3 , Fig. 4 and Fig. 5 show that the performance of *AdaBoost.MH* is the best when combining *AdaBoost.MH* with *Gentle Boost*, *Logit Boost* and *L2 TreeBoost*.

3.2 Some generalization of two-class algorithms

Some of two-class algorithms have their generalizations in multi-class case. In this section, we present the usage of the negative Binomial likelihood loss function in multi-class framework.

$$L(y, F(\mathbf{x})) = -\sum_{j=1}^{J} y_j^* \log p_j(\mathbf{x})$$

in which $p_j(\mathbf{x}) = \frac{e^{F_j(\mathbf{x})}}{\sum\limits_{l=1}^{J} e^{F_l(\mathbf{x})}}, j = 1 : J.$

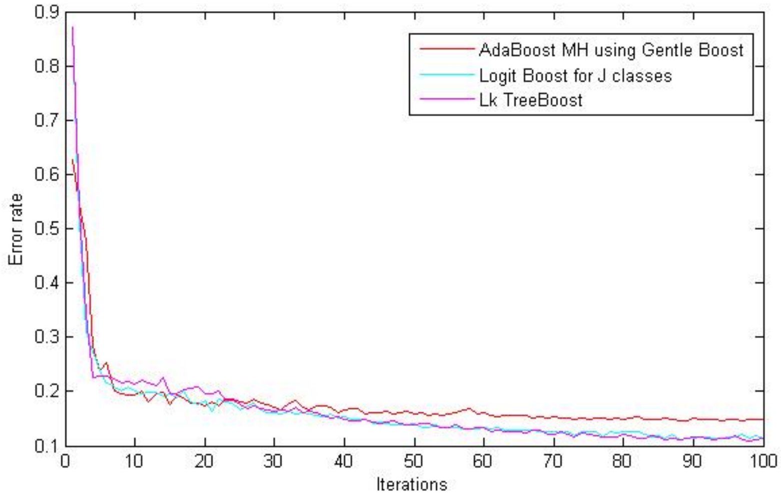

FIGURE 6 – Multi-class classification on simulated data, $J = 4$, Bayes error 0%.

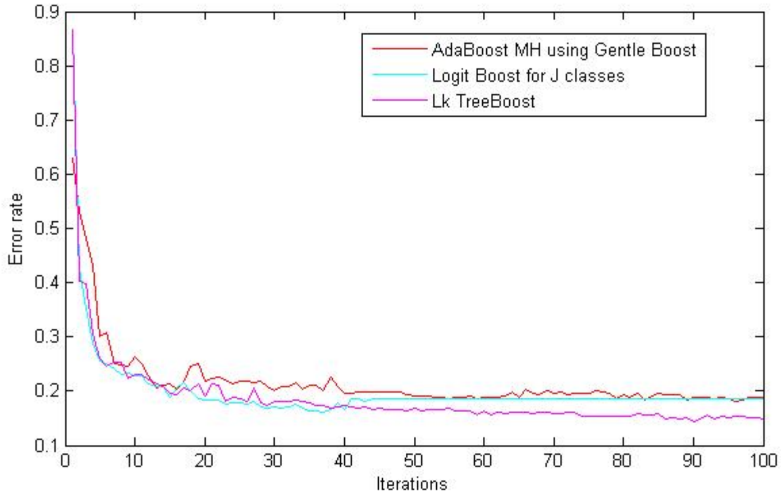

FIGURE 7 – Multi-class classification on simulated data, $J = 4$, Bayes error 5%.

We derive the multi-class version of *Logit Boost* similarly as in two-class case in taking d_m to be Newton-Raphson steps by a **population version** way (parameterize d_m by $\mathbf{x}$). If instead, we only take d_m to be the gradient in a parameterization by i (**non-population version**) and then process the line-search via regression trees like what we have explained in 2.4, we obtain *Lk TreeBoost* algorithm. These two algorithms are summarized below.

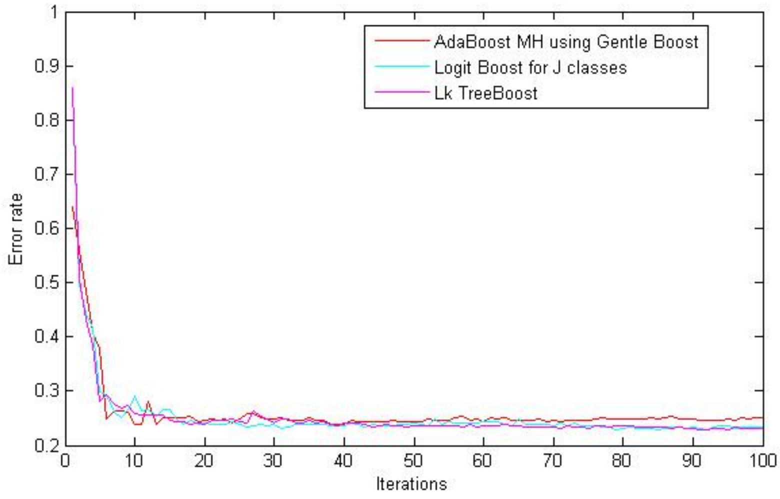

FIGURE 8 – Multi-class classification on simulated data, $J = 4$, Bayes error 10%.

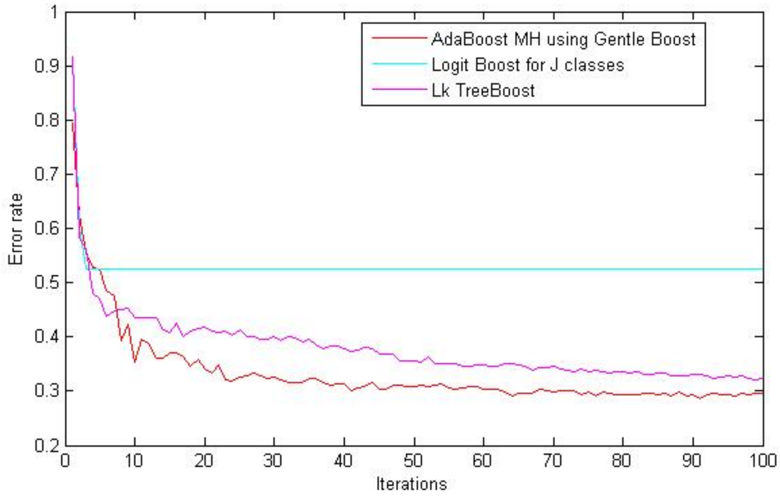

FIGURE 9 – Multi-class classification on simulated data, $J = 6$, Bayes error 0%.

```
LogitBoost for $J$ classes
1. Start with weights $w_{ij} = 1/N, i = 1 : N, j = 1 : J, F_j(\mathbf{x}) = 0$ and $p_j(\mathbf{x}) = 1/J, \forall j = 1 : J$.
2. Repeat for $m = 1,2,...,M$ :
(a) Repeat for $j = 1,...,J$ :
i. Compute $z_{ij} = \dfrac{y^*_{ij} - p_j(\mathbf{x}_i)}{p_j(\mathbf{x}_i)(1 - p_j(\mathbf{x}_i))}$ and $w_{ij} = p_j(\mathbf{x}_i)(1 - p_j(\mathbf{x}_i))$.
ii. Fit the function $f_{mj}(\mathbf{x})$ by a weighted least-squares regression of $z_{ij}$ to $\mathbf{x}_i$ with weights $w_{ij}$.
(b) Set $f_{mj}(\mathbf{x}) \leftarrow \dfrac{J-1}{J} \left( f_{mj}(\mathbf{x}) - \dfrac{1}{J} \sum_{k=1}^{J} f_{mk}(\mathbf{x}) \right)$.
(c) Update $p_j(\mathbf{x})$ via $F_k(\mathbf{x}), k = 1 : J$.
3. Output the classifier $arg \max_{j=1:J} F_j(\mathbf{x})$.
```

```
Lk TreeBoost for $J$ classes
1. Start with $F_j(\mathbf{x}) = 0, \forall j = 1 : J$.
2. Repeat for $m = 1,2,...,M$ :
  (a) Update $p_j(\mathbf{x})$ via $F_j(\mathbf{x}), j = 1 : J$.
  (b) Repeat for $j = 1,...,J$ :
    i. Compute $z_{ij} = y^*_{ij} - p_j(\mathbf{x}_i)$.
    ii. Fit regression tree $\{R_{kjm}\}_{k=1:K}$ a $K$-terminal node from the training $\{z_{ij}, x_i\}_{i=1:N}$.
    iii. Take $\lambda_{kjm} = \frac{J-1}{J} \frac{\sum_{\mathbf{x}_i \in R_{kjm}} z_{ij}}{\sum_{\mathbf{x}_i \in R_{kjm}} |z_{ij}|(1-|z_{ij}|)}, k = 1 : K$.
    iv. $F_{jm}(\mathbf{x}) = F_{j,m-1}(\mathbf{x}) + \sum_{k=1}^{K} \lambda_{kjm} 1(\mathbf{x} \in R_{kjm})$.
3. Output the classifier $arg \max_{j=1:J} F_j(\mathbf{x})$.
```

Experiments : We use the same simulated multi-class dataset with $J = 4$ and Bayes error is 0%, 5% and 10%. The results are shown in Fig. 6, 7 and 8. With $J = 4$, we see that the two generalizations of negative binomial loss function algorithms in multi-class case perform better than all versions of *AdaBoost MH*. But an disadvantage of *Logit Boost for J classes*, as mentioned in [1] and [2], is the numerical instability when one of weights $w(\mathbf{x}) = p(\mathbf{x})(1-p(\mathbf{x}))$ tends to zero. Indeed, when we try with $J = 6$, the result in Fig. 9 shows that *Logit Boost* encounters a numerical problem. In this experiment, *AdaBoost MH* performs the best among these three algorithms.

4 Experiments

4.1 Experiments with simulated data

4.2 Experiments with real data

5 Conclusion

Références

[1] Friedman, J., Hastie, T. & Tibshirani, R. *Additive Logistic Regression : a Statistical View of Boosting*, 2000.

[2] Friedman, J. *Greedy Function Approximation : A Gradient Boosting Machine*, IMS 1999 Reitz Lecture, 2001.

[3] Schapire, R.E. & Singer, Y. *Improved Boosting Algorithms : Using Confidence-rated Predictions*, 1998.