Lecture V Boosting and multiplicative updates algorithms

Marina Meilă
mmp@stat.washington.edu

Department of Statistics
University of Washington

CSE 547/STAT 548
Winter 2022

Outline

1. Definitions

2. Boosting
   - There is more than one way to average predictors
   - Bayesian averaging
   - Bagging
   - Stacking
   - Mixtures of Experts
   - Forward Fitting and Backfitting
   - AdaBoost

3. Boosted predictors are additive models
   - AdaBoost is steepest descent on training set
   - A statistical view of boosting

4. More surrogate losses, more boosting algorithms
   - Why the e^{-yf} loss? other surrogate losses
   - GradientBoost
   - Practicalities

5. [Multiplicative updates algorithms]

Definitions – Supervised learning

- Data set $\mathcal{D} = \{(x^1, y^1), \ldots (x^n, y^N)\}$
- Loss function $L(y, \hat{y})$ – to be minimized during training
- Predictor $f_\theta(x)$
- Loss on training set (empirical loss) $\hat{L}(f_\theta) = \frac{1}{N} \sum_{i=1}^{N} L(y^i, f_\theta(x^i))$

We will also talk about

- Decision/Regression Tree
- Random forest
- Logistic regression
- Gradient descent/Stochastic Gradient Descent (SGD)
- Bias and variance, overfitting, underfitting
  (known from STAT 535/CSE 546 or other ML classes)

Stochastic Gradient Descent

Goal: find minimum of function f_θ w.r.t $\theta \in \mathbb{R}^d$

STOCHASTIC GRADIENT DESCENT (SGD)

Input

for $k = 1, 2, \ldots K$

1. get $d^k \in \mathbb{R}^d$ direction of descent: $d^k = \nabla f_{\theta^k}$+noise
2. get step size $\eta^k > 0$
3. update θ:

$$\theta^{k+1} \leftarrow \theta^k - \eta^k d^k \tag{1}$$

Output θ^K (or better $\bar{\theta}$ average over last steps)

Note

- When f_θ is $\hat{L}(\mathcal{D}; \theta)$ an empirical loss function, it is typical to set $d^k = \frac{\partial L(y^i, f_{\theta^k}(x^i))}{\partial \theta}$
- If data is streaming, then SGD becomes on-line algorithm (never stops learning θ)
- In gradient descent η^k is optimized to approximately maximize the decrease in f (or L)

On-line learning

data streaming, θ changes at each time step

for $t = 0, 1, \ldots$

1. x^t arrives
2. predict $\hat{y}^t = f_{\theta^{t-1}}(x^t)$
3. observe y^t (if y^t not observed we would be doing Reinforcement Learning)
4. pay loss $L(y^t, \hat{y}^t)$
5. update $\theta^{t+1} \leftarrow$ Learning Alg$(x^t, y^t, \ldots)$

How to measure performance of on-line algorithm?

- denote $\mathcal{D}^t = \{(x^{1:t}, y^{1:t})\}$
- θ^t_* is the result of off-line learning θ from $\mathcal{D}^t$.
- off-line we can learn at least as well as on-line (from the same data, "by the same algorithm"/optimizing the same L)
  Assume $E[L(y^{t+1}, f_{\theta^t_*}(x^{t+1})] \leq E[L(y^{t+1}, f_{\theta^t}(x^{t+1})]$
- (instantaneous) **regret at time** t

$$r(t) \;=\; L(y^{t+1}, f_{\theta^t}(x^{t+1}) - L(y^{t+1}, f_{\theta^t_*}(x^{t+1})$$

- Average **regret**

$$R(t) \;=\; \frac{1}{t}\sum_{s=1}^{t} r(s)$$

Wanted

Best: $R(t) \to 0$ for $t \to \infty$ asymptotically we are doing as well as processing all the data at once

stronger possible too $\sum_{s=1} tr(t) \leq M$, finite for all t (true for Perceptron algorithm: makes finite number of mistakes)

Okay: $R(t) \leq \epsilon$ for t sufficiently large

Multiplicative updates algorithms

Area developed at the frontier between game theory and computer science, and in machine learning. For a general and clear discussion of these algorithms see **?** "The Multiplicative updates method".

Weighted majority We have "experts" $b^{1:M}$ who can predict the stock market (with some error). The predictions in this problem are binary, i.e $\{up, down\}$ We want to predict the stock market by combining their predictions in the function $f = \sum^k w^k b^k$. The following algorithm learns f by optimizing the weights w^k.

WEIGHTED MAJORITY ALGORITHM

Initialize $w_i^0 \leftarrow 1$

for $t = 1, 2, \ldots$

1. $w_i^t \leftarrow w_i^{t-1}(1-\epsilon)$ if expert i makes a mistake at time t
2. predict the outcome that agrees with the weighted majority of the experts

It can be shown that the number of mistakes m^t of f up to time t is bounded by

$$m^t \leq \frac{2 \ln M}{\epsilon} + 2(1+\epsilon) m_j^t \tag{2}$$

where m_j^t is the number of mistakes of any expert j. Thus, asymptotically, the number of mistakes of the algorithm is about twice those of the best expert.

For a more general algorithm, that includes the above case, **?** prove that to achieve a tolerance δ w.r.t to the optimal average loss, one needs to make $\mathcal{O}(\ln M/\delta^2)$ updates.

Feasibility problem for LP, with oracle

The problem is to find a point $x \in \mathbb{R}^n$ satisfying M linear constraints given by

$$Ax \geq b, \qquad A \in \mathbb{R}^{M \times n},\ b \in \mathbb{R}^M \tag{3}$$

The **oracle** is a blackbox which, given a single constraint $c^T x \geq d$ returns a point x satisfying it whenever the constraint is feasible. It is assumed that the oracle's responses x satisfy $A_i x - b^i \in [-\rho, \rho]$ for all rows i of A and that ρ is known.

LINEAR PROGRAM WITH ORACLE parameters ρ, δ
Initialize $w_i = 1/M$ the weight of each constraint
for $t = 1, 2, \ldots T$

1. Call ORACLE with $c = \sum_i w_i A_i$, $d = \sum_i w_i b^i$ and obtain x^t
2. Penalty for equation i is $r_i^t = A_i x^t - b^i$
3. Update weights by

$$w_i^{t+1} \leftarrow w_i^t (1 - \epsilon \cdot \mathrm{sgn} r_i^t)^{|r_i^t|} \tag{4}$$

with $\epsilon = \frac{\delta}{4\rho}$ and renormalize the weights.
Output $x = \sum_t x^t / T$

The number of steps $T \propto \rho^2$.
In **?** it is shown (Exercise prove it based on the initial assumptions!) that (1) if ORACLE returns a feasible x^t at all steps, then x satisfies $A_i x - b + \delta \geq 0$ i.e the system is satisfied with tolerance δ; (2) if ORACLE declares infeasibility in some step, then the program is infeasible.

There is more than one way to average predictors

Classification will be the running exaple here, but most results hold for other prediction problems as well.
Denote $\mathcal{B} = \{b\}$ a **base classifier** family
Averaging:

$$f(x) = \sum_{k=1}^{M} \beta^k b^k(x) \tag{5}$$

f is real-valued even if the b^k's are ± 1 valued
Why average predictors?

- to reduce bias
- to compensate for local optima (a form of bias)
- to reduce variance
- if $b_1, b_2, \ldots b_M$ make independent errors, averaging reduces expected error (loss). We say that b_1 and b_2 make **independent errors** $\Leftrightarrow P(b_1 \text{ wrong} \,|\, x) = P(b_1 \text{ wrong} \,|\, x,\, b_2 \text{ wrong})$
- because the b^k functions are given: real world domain experts (weather prediction), a set of black-box classifiers (from a software package), a set of [expert designed] features (speech recognition, car/human recognition) each of them weakly informative of y
- because $\mathcal{B}$ is a set of (simple) "basis functions" and we need a more complex predictor in our task

Averaging is not always the same thing

Depending how we choose $\mathcal{B}$, $b^1, b^2, \ldots b^M$ and $\beta^1, \beta^2, \ldots \beta^M$, we can obtain very different effects.
We will examine

- Bayesian averaging (briefly)
- Mixtures of experts (briefly)
- Bagging (briefly)
- Backfitting (briefly)
- Stacking (briefly))
- Boosting

Bayesian averaging

Assume any predictor $b \in \mathcal{B}$ could be the "true" predictor with some **prior** probability $P_0(b)$.
Learning means changing the probability distribution of b after seeing the data.

Before seeing data	$P_0(b)$	prior probability of b
After seeing $\mathcal{D}$	$P(b\|\mathcal{D})$	posterior probability of b
Bayes formula	$P(b\|\mathcal{D}) = \frac{P_0(b)P(\mathcal{D}\|b)}{\sum_{b' \in \mathcal{B}} P_0(b')P(\mathcal{D}\|b')}$	

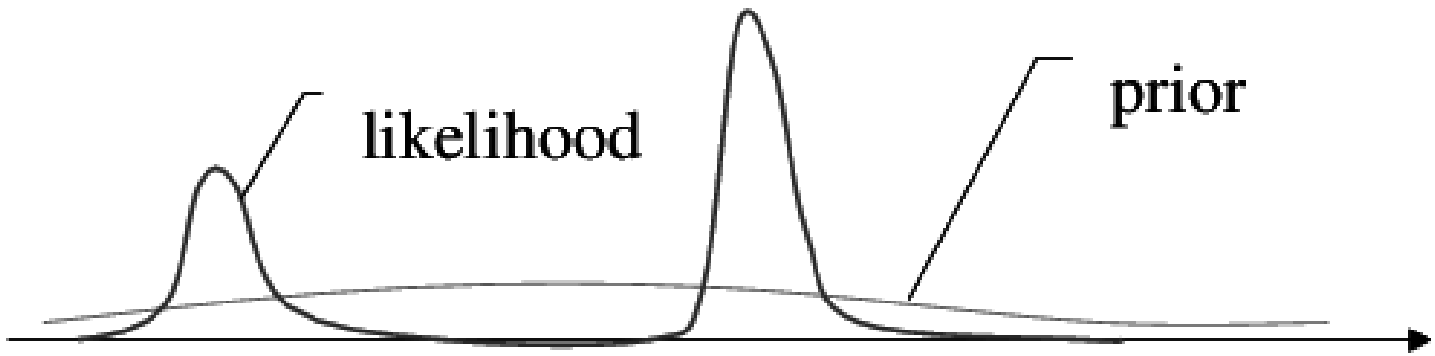
Classification of a new instance by Bayesian averaging:

$$f(x) = \int_{\mathcal{B}} b(x) dP(b|\mathcal{D})$$

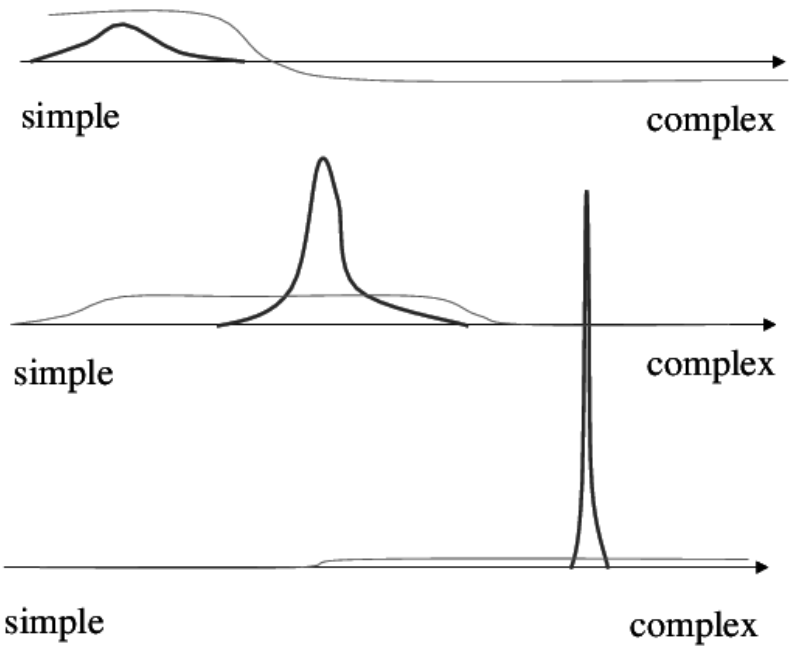
or

$$P(y|x, \mathcal{D}) = \int_{\mathcal{B}} 1_{b(x)=y} dP(b|\mathcal{D})$$

Hence classifiers (or more generally predictors) are weighted by their posterior probability.
Intuition: The likelihood becomes more concentrated when N increases



Bayesian averaging and model complexity



- model too simple: likelihood low, prior high
- model about right: likelihood high, prior not too low
- model too complex: likelihood high, prior very low

Bayesian averaging in practice.

$$\hat{P}(b^k|\mathcal{D}) = \frac{P_0(b^k)P(\mathcal{D}|b^k)}{\sum_{k'=1}^{M} P_0(b_{k'})P(\mathcal{D}|b_{k'})} \equiv \beta_k$$

$b^{1:M}$ are either sampled from $\mathcal{B}$, or trained separately (e.g local minima of $\hat{L}$, models of different complexities)

Priors in practice

- non-informative
- complexity penalizing, sparsity inducing, etc

Reducing variance: Bagging

What if we had several (independently sampled) trainig sets $\mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_M$?

- we could train classifiers $\{b^1, b^2, \ldots b^M\}$ on the respective $\mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_M$
- we could estimate $E_{P(b)}[b] \sim f = \frac{1}{M} \sum_{k=1}^{M} b^k$
- f has always lower variance than b^k

Idea of **bagging**: sample $\mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_M$ from the given $\mathcal{D}$ and estimate b^k on $\mathcal{D}_k$

$$f(x) = \operatorname{sgn} \frac{1}{M} \sum_{k=1}^{M} b^k(x)$$

Thus, bagging is a form of boostrap.


sample $N' \leq N$ samples $\times$ M times

$\mathcal{D}_1$ $\mathcal{D}_2$ $\mathcal{D}_3$

Resulting "bagged" classifier $F = \frac{1}{3}(b_1 + b_2 + b_3)$

Bagging reduces variances

It was shown theoretically and empirically that bagging reduces variance.
Bagging is good for

- base classifiers with high variance (complex)
- unstable classifiers (decision trees, decision lists, neural networks)
- noisy data

Example

Random Forests A large ensemble of decision trees is fitted to the same data set, introducing randomness in various ways, such as (1) resampling the data set, (2) taking random splits, with probabilities that favor "good" splits, etc. The output predictor is the (unweighted) average of all the trees.

- can be extended to more than classification (regression, feature selection)
- training can be done in parallel
- computing $f(x)$ on new example is fast enough
- VERY POPULAR in industry

Stacking

- very general method (any kind of predictors or costs)
- for complex base classifiers

1. Fit predictors $b^1, \ldots b^M$ to the data $\mathcal{D}$. The predictors can be from different model classes (i.e neural networks, CART, nearest neighbors, logistic regressions) or use different sets of features.
   For $k = 1 : M$, for data point $i = 1 : N$
   - train b^k_{-i} from the model class of b^k on $\mathcal{D} \setminus \{(x^i, y^i)\}$
2. Fit coefficients $\beta_{1:M}$ by minimizing the leave-one-out (loo) empirical cost $\hat{L}^{loo}$

$$\hat{L}^{loo}(\beta_{1:M} = \frac{1}{N} \sum_{i=1}^{N} L(y^i, \sum_{k=1}^{M} \beta_k b^k_{-i}(x^i)) \tag{6}$$

   (or by cross-validation).

Note that if L is a non-linear function, the minimization in (**??**) is a non-linear minimization, and in particular for convex losses this is a convex optimization problem in $\beta_{1:M}$.

Mixtures of Experts

Here $\beta_k = \beta_k(x)$, and $\sum_k \beta^k(x) = 1$ for all x in the domain of the inputs.
Idea each b^k is an "expert" in one region of the input space.
Wanted $f \approx b^k(x)$ in the region of expertise of expert b^k
The vector function $\beta(x) = [\beta^1(x) \ \ldots \ \beta^M(x)]$ is sometimes called the gating function.

Example

Suppose the true function to learn is $f^*(x) = |x|$, $x \in \mathbb{R}$. This can be well approximated by two linear experts $f^1(x) = x$, $f^2(x) = -x$ with weights $\beta^1 = \phi(x), \beta^1 = \phi(-x)$, where ϕ is the sigmoid function (hence $\beta^1 + \beta^2 = 1$ everywhere.

The example highlights that, by using a mixture of experts we can construct a more complex classifier by from simple classifiers (linear). "Simple" can mean easy to fit, or low complexity (aka simple decision region), or both. The effective sample size for each b^k is smaller than N, and corresponds to the data for which $\beta^k(x^i)$ is away from 0.

For more than two experts, a natural gating function is the softmax function

$$\beta^k(x) = \frac{e^{v_k^T x}}{\sum_{l=1}^{M} e^{v_l^T x}} \text{ with } v_l \in \mathbb{R}^n \text{ a vector of parameters} \tag{7}$$

Training By descent methods. Often the experts $f^{1:M}$ and gating functions $\beta^{1:M}$ are trained simultaneously (to maximize log-likelihood) by steepest descent, or EM algorithm (which we'll study later).

Forward Fitting and Backfitting

(b^k, β^k) are fitted iteratively (sequentially), one k at a time.
In some cases, the weights β^k can be absorbed into b^k.
f^t is f at iteration t The residual $r \in \mathbb{R}^N$ is defined as $r^t(x^i) = y^i - f^t(x^i)$.

FORWARDFITTING ALGORITHM

Input M, labeled training set $\mathcal{D}$
Initialize $f = 0$
repeat
  for $k = 1, 2, \ldots M$
    fit k-th predictor $\beta^k, b^k = \text{argmin} \hat{L}(f + \beta b)$
    update $f = f + b^k \beta^k$
until change in $\hat{L}$ small enough (or, change in b^k small enough)
Output $f(x) = \sum_{k=1}^{M} \beta^k b^k(x)$

Note that M does not have to be set in advance.

Backfitting

Set M at the beginning, and cycle through the M predictors, updating predictor k while keeping the others fixed. Denote by

$$f^{-k}(x) = \sum_{l \neq k} \beta^k b^k(x) \tag{8}$$

the combined predictor f "minus" the k-th base predictor b^k.

BackFitting Algorithm

Input M, labeled training set $\mathcal{D}$

Initialize $b^{1:M} = 0$, [$\beta^{1:M} = 0$ if there are coefficients β]

repeat

  for $k = 1, 2, \ldots M$

    calculate $r^k(x^i) = y^i - f^{-k}(x^i)$, $i = 1 : N$

    optimize $\hat{L}$ w.r.t k-th base predictor $\beta^k, b^k = \text{argmin} \hat{L}(r^k + \beta b)$

until change in $\hat{L}$ small enough (or, change in $b^{1:M}$ small enough)

Output $f(x) = \sum_{k=1}^{M} \beta^k b^k(x)$

Often these methods are used when b^k are assumed to be simple, weak, and the predictor f is built from the cooperation of several b's. Thus, they are generally bias reduction method.

Example

Least squares regression In this problem, it is useful to denote

$$r^k(x^i) \;=\; y^i - f^{-k}(x^i) \tag{9}$$

the residual of f^{-k} at data point x^i. Then, $L_{LS}(y^i, f(x^i)) = y^i - f^{-k}(x^i) - \beta^k b^k(x^i)$. Hence, optimizing $\hat{L}_{LS}$ w.r.t. β^k, b^k can be expressed as

$$\beta^k, b^k \;=\; \operatorname*{argmin}_{\beta, b} \sum_{i=1}^{N} (r^i - \beta b(x^i))^2 \,. \tag{10}$$

In other words, each step of backfitting is a least squares regression problem, where the output variable values y^i are replaced with the current residuals r^i.

See also Additive Models

Reducing bias: Boosting

Base classifier family $\mathcal{B}$ has large bias (e.g. linear classifier, decision stumps) but learning always produces b that is better (on the training set) than random guessing.
Preconditions for boosting

1. Learning algorithm accepts **weighted** data sets. Training minimizes

$$\hat{L}_{01}^{w}(b) = \sum_{i=1}^{N} w_i L_{01}(y^i, b(x^i)) \quad \text{with} \quad \sum_{i=1}^{N} w_i = 1.$$

2. $\mathcal{B}$ is a weak classifier family. For any $\mathcal{D}$ and any weights $w_{1:N}$ there can be found $b \in \mathcal{B}$ such that the training error of b on $\mathcal{D}$ is bounded below one half.

$$0 < \hat{L}_{01}^{w}(b) \leq \delta < \frac{1}{2}$$

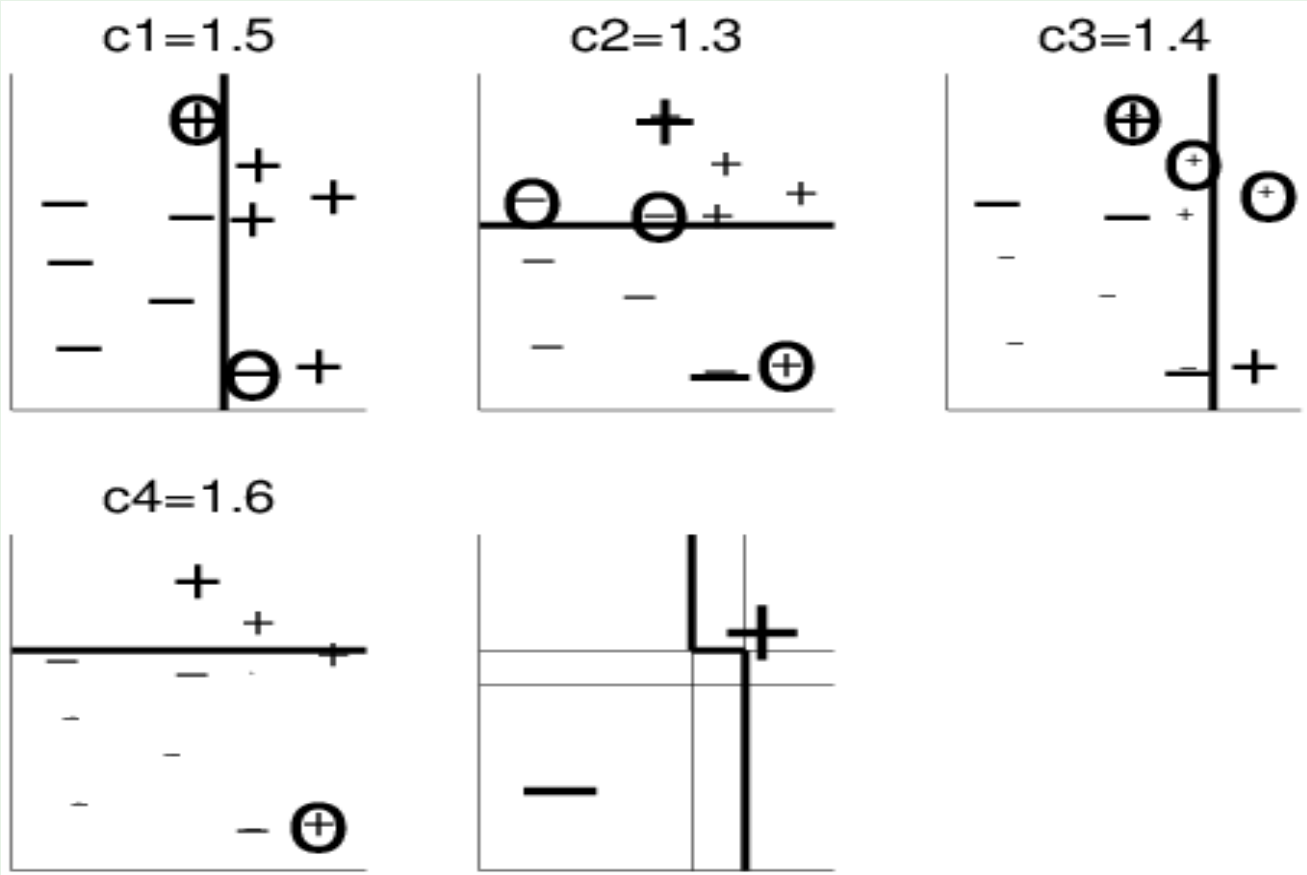
Idea of boosting: train a classifier b^1 on $\mathcal{D}$, then train a b^2 to correct the errors of b^1, then b^3 to correct the errors of b^2, etc.

Example

Boosting with stumps

Stumps are decision trees with a single split.

(below, $c_1 \ldots c_4$ denote the coefficients $\beta_{1:4}$).



AdaBoost Algorithm

ADABOOST ALGORITHM

Assume $\mathcal{B}$ contains functions b taking values in $[-1, 1]$ or $\{\pm 1\}$

Input M, labeled training set $\mathcal{D}$

Initialize $f = 0$

$w_i^1 = \frac{1}{N}$ weight of datapoint x_i

for $k = 1, 2, \ldots M$

1. "learn classifier for $\mathcal{D}$ with weights w^k" $\Rightarrow$ b^k
2. compute error $\varepsilon^k = \sum_{i=1}^N w_i^k \frac{1-y_i b^k(x_i)}{2}$
3. set $\beta^k = \frac{1}{2} \ln \frac{1-\varepsilon^k}{\varepsilon^k}$
4. compute new weights $w_i^{k+1} = \frac{1}{Z^k} w_i^k e^{-\beta^k y_i b^k(x_i)}$ where Z^k is the normalization constant that makes $\sum_i w_i^{k+1} = 1$

Output $f(x) = \sum_{k=1}^M \beta^k b^k(x)$

Remarks

1. If $b(x) \in \{\pm 1\}$ then $y^i b(x^i) \in \{\pm 1\}$, and $\frac{1-y^i b(x^i)}{2} = 1$ if an error occurs and 0 otherwise. Thus, ε^k in step 2 adds up the weights of the errors.
   If $b(x) \in [-1, 1]$ then the errors contribute different amounts to the loss depending on their margin.
2. In both cases, $\varepsilon^k \in [0, \delta]$, $\delta < 0.5$ by the weak learner property of $\mathcal{B}$
3. $\beta^k > 0$ whenever $\varepsilon^k < 1/2$.
4. If $b \in \{\pm 1\}$, then step 4 can be written equivalently (up to a multiplicative constant)

$$w_i^{k+1} = \begin{cases} \frac{1}{Z^k} w_i^k & \text{if } b^k(x_i) = y_i \\ \frac{1}{Z^k} w_i^k e^{2\beta^k} & \text{if } b^k(x_i) \neq y_i \end{cases} \tag{11}$$

   This form corresponds to the DiscreteAdaBoost algorithm, the first AdaBoost algorithm published, which assumed $b(x) \in \{\pm 1\}$. As we shall see later, modern boosting algorithms dispense with the assumption $b \in [-1, 1]$ too.

An interpretation of the weights

$$w_i^{k+1} = \frac{1}{N} \prod_{k' \leq k} \frac{e^{-\beta^{k'} y_i b^{k'}(x_i)}}{Z_{k'}} = \frac{e^{-y_i f^k(x_i)}}{N \prod_{k' \leq k} Z^k} \tag{12}$$

- weight of example i at step k is proportional to $e^{-y_i f^{k-1}(x_i)}$ the exponential of its negative margin
- Examples that have been hard to classify get exponentially high weights.
- Examples that are classified with high margins get vanishingly small weights.

The normalization constant is an average loss

If we sum both sides of (**??**) over i we obtain

$$1 = \frac{\sum_i e^{-y_i f^k(x_i)}}{N \prod_{k' \leq k} Z^k}, \tag{13}$$

or

$$\prod_{k' \leq k} Z^k = \frac{\sum_i e^{-y_i f^k(x_i)}}{N} \equiv \hat{L}_\phi(f^k) \tag{14}$$

where

$$\phi(z) = e^{-z}. \tag{15}$$

and

$$L_\phi(y, f(x)) = \phi(yf(x)) \tag{16}$$

Hence, the r.h.s of (**??**) is the average over the data set of the **exponential loss** L_ϕ.
The function ϕ decreases with the margin, thus decreasing $\hat{L}_\phi$ will produce a better classifier (on the training set). In this sense, L_ϕ is an alternative loss function for classification.

$\hat{L}_\phi$ decreases exponentially with M

For simplicity, we show this in the special case $b(x) \in \{\pm 1\}$ for all $b \in \mathcal{B}$.

$$Z^k = \sum_{i=1}^{n} w_i^k e^{-\beta^k (y_i b^k(x_i))} \tag{17}$$

$$= e^{\beta^k} \underbrace{\sum_{i=err} w_i^k}_{\varepsilon^k} + e^{-\beta^k} \underbrace{\sum_{i=corr} w_i^k}_{1-\varepsilon^k} \tag{18}$$

$$= e^{\beta^k} \varepsilon^k + (1-\varepsilon^k) e^{-\beta^k} \tag{19}$$

$$= \sqrt{\frac{1-\varepsilon^k}{\varepsilon^k}} \varepsilon^k + \sqrt{\frac{\varepsilon^k}{1-\varepsilon^k}} (1-\varepsilon^k) = 2\sqrt{(1-\varepsilon^k)\varepsilon^k} \leq \gamma \tag{20}$$

where $\gamma < 1$ depends on δ the maximum error. It follows that

$$\hat{L}_\phi(f^k) = \prod_{k=1}^{k} Z^{k'} \leq \gamma^k \tag{21}$$

The training set error $\hat{L}_{01}$ decreases exponentially with M

Note that $\phi(z) \geq 1_{z<0}$ for all z (see also figure **??**). Therefore

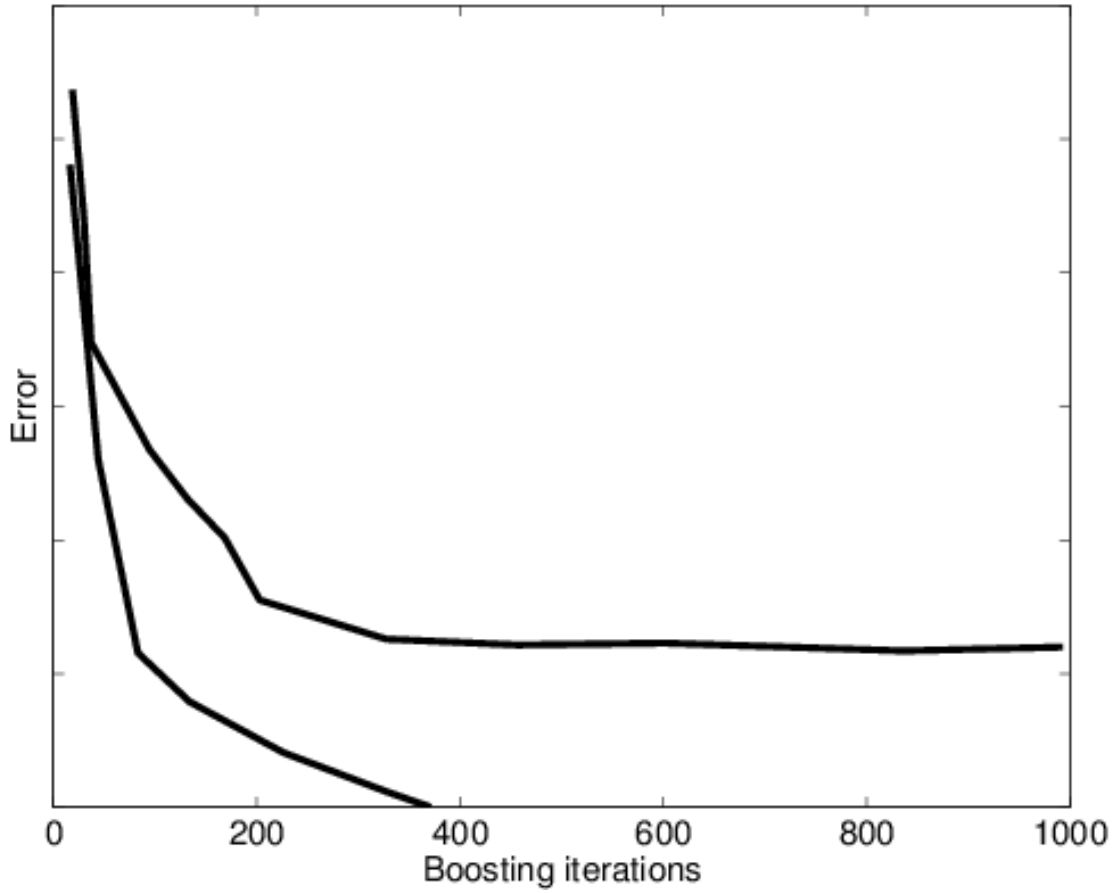
$$\hat{L}(f^k) = \frac{1}{N}\sum_{i=1}^{N} 1_{[y_i f^k(x_i)<0]} \tag{22}$$

$$\leq \frac{1}{N}\sum_{i=1}^{N} e^{-y_i f^k(x_i)} = \hat{L}_\phi(f^k) \leq \gamma^k \tag{23}$$

In other words, the training error $\hat{L}(f^k)$ is bounded by a decaying exponential. Moreover, since $\hat{L}(f^k) \in \{0, 1/N, 2/N, \ldots 1\}$, it follows that after a finite number of steps, when $\gamma^{k^0} < 1/N$, the training error will become 0 and the training data will be perfectly classified!

The test set error and overfitting

- Do NOT take $M = k^0$. The number of steps M for good generalization error is often much larger than k^0 (and sometimes smaller).
- Below is a typical plot of $\hat{L}$ and L (which can be estimated from an independent sample) vs the number of boosting iterations.



Boosted predictors are additive models

An **additive model** (for prediction) has the form

$$f(x) \equiv E[Y|x] = \alpha + b_1(x_1) + b_2(x_2) + \ldots + b_n(x_n) \tag{24}$$

In other words, it is a linear model, where each coordinate has been non-linearly transformed. A generalization of the above definition, which is still called an additive model, is

$$f(x) = \alpha + \beta_1 b_1(x) + \beta_2 b_2(x) + \ldots + \beta_M b_M(x) \tag{25}$$

This is a linear model over a set of new features $b_{1:M}$.

Example (Linear model and neural net)

If $b_j = x_j$, $j = 1 : n$, the model (**??**) is a linear model.
If $b_j \in \{\frac{1}{1+e^{-\gamma^T x}}, \gamma \in \mathbb{R}^n\} = \mathcal{B}$ (the family of logistic functions with parameter $\gamma \in \mathbb{R}^n$) then $f(x)$ is a [two layer] neural network.

Additive Logistic Regression While the predictors above are well suited for regression, for classification one may employ a logistic regression, i.e

$$f(x) \equiv \frac{P(Y=1|x)}{P(Y=-1|x)} = \alpha + \beta_1 b_1(x) + \beta_2 b_2(x) + \ldots + \beta_M b_M(x) \tag{26}$$

[Generalized Additive Models. Link function. See supplementary notes]

How to train an Additive Model?

[Alg 9.2 HTF for Additive Logistic Regression]
An additive model for prediction can be trained in several different ways.

Given base family $\mathcal{B}$, data $\mathcal{D}$, loss function L

- Fix M from the start and optimize over all the parameters and base functions at once.
- Backfitting Fix M from the start but optimize only one b_j, β_j at a time, keeping the others fixed
- Forward fitting Optimize b_k, β_k sequentially, for $k = 1, 2, \ldots$ without refitting previously fit base models. In this case, M need not be fixed in advance.
  It turns out that this is what boosting does.

AdaBoost is steepest descent on training set

We will show that boosting is a form of (stochastic) gradient descent on the surrogate loss $\hat{L}_\phi$ (we already know from Part I that AdaBoost pushes $\hat{L}_\phi$ asymptotically towards 0).
Assume we want to minimize the surrogate loss $\hat{L}_\phi$ on the training set. For any finite $\mathcal{D}$, f and $b \in \mathcal{B}$ affect $\hat{L}_\phi$ only via the N-dimensional vectors of their values on $\mathcal{D}$ (which we will abusively denote by f, b)

$$f = \begin{bmatrix} f(x^1) \\ f(x^2) \\ \dots \\ f(x^N) \end{bmatrix} \quad b = \begin{bmatrix} b(x^1) \\ b(x^2) \\ \dots \\ b(x^N) \end{bmatrix} \tag{27}$$

Thus, $\hat{L}_\phi(f)$ is a function of N variables, with partial derivatives

$$\frac{\partial \hat{L}_\phi}{\partial f(x^i)} = \frac{\partial}{\partial f(x^i)} \left[\frac{1}{N} \sum_{i=1}^{N} \phi(y^i f(x^i)) \right] = \frac{1}{N} y^i \phi'(y^i f(x^i)) = -\frac{1}{N} y^i e^{-y^i f(x^i)}, \tag{28}$$

since $\phi'(z) = -e^{-z}$. Imagine a boosting step as trying to find a change βb in f which minimizes the loss $\hat{L}_\phi(f + \beta b)$. This minimization is equivalent to maximizing the decrease in loss $\hat{L}_\phi(f) - \hat{L}_\phi(f + \beta b)$.

The direction of descent

The change in L_ϕ along "direction" b with step size β is approximately

$$\hat{L}_\phi(f) - \hat{L}_\phi(f+\beta b) \approx -\left(\nabla_f \hat{L}_\phi(f)\right)^T(\beta b) = \sum_i \left[\left(\frac{1}{N} y^i e^{-y^i f(x^i)}\right)(\beta b(x^i))\right] \propto \sum_i y^i b(x^i) w_i \quad (29)$$

(denoting/recalling $w_i \propto e^{-y_i f(x^i)}$).
The direction of steepest descent b is therefore the maximizer of

$$\operatorname*{argmax}_{b \in \mathcal{B}} \sum_i w_i y_i b(x^i) \quad (30)$$

where in the sum on the r.h.s we recognize the r of AdaBoost.

- If $b(x^i) = \pm 1$ values, then $1 - y_i b(x^i) = 1_{[i \text{ error}]}$, and maximizing (**??**) is the same as minimizing the weighted training error $\hat{L}_{01}^w$.
- If b takes real values, then $y_i b(x^i)$ is the margin of example i, and maximizing (**??**) is a natural objectiv for many training algorithm. Exercise Can you find examples of algorithms/predictors which do/don't maximize the loss in (**??**)?

More generally (we will use this later), the direction b maximizes

$$\sum_i y_i b(x^i)[-\phi'(y_i f(x^i))] \quad (31)$$

Finding the direction b is equivalent with step 1 of the AdaBoost algorithm, training a weak classifier on the weighted data. The resulting b can be seen as the best approximate of the gradient of L_ϕ in $\mathcal{B}$.

The line minimization

Now let us do line minimization: find the optimal step size β in direction b. For this we take the derivative of $\hat{L}_\phi(f + \beta b)$ w.r.t β and set it to 0.

$$\frac{d\hat{L}_\phi(f+\beta b)}{d\beta} = \sum_i y_i b(x^i)\phi'(y_i f(x^i)) = -\sum_i y_i b(x^i) e^{-y_i f(x^i) - \beta y_i b(x^i)} \tag{32}$$

β is the (unique) root of

$$\sum_i w_i y_i b(x^i) e^{-\beta y_i b(x^i)} = 0 \tag{33}$$

If
- $b(x) \in \{-1, 1\}$ then line optimization gives β^k from AdaBoost
- $b(x) \in [-1, 1]$ then line optimization gives β^k from AdaBoost approximately
- $b(x) \in (-\infty, \infty)$ then β amounts to a rescaling of b and is redundant.

Calculating β^k for binary b's

Assume $b(x) \in \{\pm 1\}$.
In this case $y^i b(x^i) = \pm 1$ and we obtain

$$\frac{d\hat{L}_\phi(f + \beta b)}{d\beta} = \sum_{i \text{ corr}} w_i e^{-\beta} - \sum_{i \text{ err}} w_i e^{\beta} = 0 \tag{34}$$

$$0 = (1 - \sum_{i \text{ err}} w_i) - \underbrace{(\sum_{i \text{ err}} w_i)}_{\varepsilon^k} e^{2\beta} \tag{35}$$

$$\beta = \frac{1}{2} \ln \frac{1 - \varepsilon^k}{\varepsilon^k} \tag{36}$$

This is the β^k coefficient of step 4 of AdaBoost

Hence, the AdaBoost algorithm can be seen as minimizing the loss $L_\phi(f)$ by steepest descent in the function space span $\mathcal{B}$.

RealAdaBoost

The third case corresponds to the RealAdaBoost in the FHT variant, described here for completeness

Real AdaBoost Algorithm (in the FHT variant)

Assume $\mathcal{B}$ contains real-valued functions

Input M, labeled training set $\mathcal{D}$

Initialize $f = 0$

$w_i^1 = \frac{1}{N}$ weight of datapoint x^i

for $k = 1, 2, \ldots M$

"learn classifier for $\mathcal{D}$ with weights $w^k \Rightarrow b^k$"

compute new weights $w_i^{k+1} = w_i^k e^{-y^i b^k(x^i)}$ and normalize them to sum to 1

Output $f(x) = \sum_{k=1}^{M} b^k(x)$

A statistical view of boosting

It has been shown **?** (FHT) that boosting can also be seen as noisy gradient descent in function space when we replace the finite training set with the true data distribution. The loss function and gradient can be given a probabilistic interpretation. This point of view is useful in two ways:

1. It shows that boosting is asymptotically minimizing a reasonable loss function, so that we can expect the performace/and algorithm behavior on finite samples to be a good predictor on its behaviour with much larger samples.
2. It is an interpretation that allows on to create a very large variety of boosting algorithms, like the LogitBost, Gentle AdaBoost and GradientBoost, presented hereafter.

Assume

- we do boosting "at the distribution level", i.e using P_{XY} instead of the empirical distribution given by $\mathcal{D}$.
- The loss function is $L_\phi(f) = E[e^{-yf(x)}]$.
  The notation $E[]$ denotes expectation w.r.t the joint P_{XY} distribution.
- learning a classifier means "find the best possible minimizer to $L_\phi(f)$"

Is L_ϕ a good loss?

Denote $p_x = P_{XY}(y = 1|x)$. The loss $L_\phi(f)$ is minimized by

$$f^*(x) = \frac{1}{2} \ln \frac{P_{XY}(y = 1|x)}{P_{XY}(y = -1|x)} = \frac{1}{2} \ln \frac{p_x}{1 - p_x}$$

And $p_x = \frac{e^{f(x)}}{e^{f(x)} + e^{-f(x)}}$ the logistic function.

Exercise Does the expresion of p_x look familiar? What is the connection?

Proof Since we are minimizing over all possible f's with no restrictions, we can minimize separately for every $f(x)$. Hence, let x be fixed

$$E_{P_{Y|X=x}}[e^{-yf(x)}] = P(y = 1|x)e^{-f(x)} + P(y = -1|x)e^{f(x)}$$

and the gradient is

$$\frac{\partial E[e^{-yf(x)}|x]}{\partial f(x)} = -P(y = 1|x)e^{-f(x)} + P(y = -1|x)e^{f(x)}$$

By setting this to 0 the result follows. □

In summary f^* is the Bayes optimal predictor for L_ϕ. But by the Proposition, f^* is also Bayes optimal for L_{01}. **(Good!)**

Steepest descent on $L_\phi(f)$ is (like) RealAdaBoost

The Real AdaBoost (with "learn a classifier" defined at the distribution level) fits an additive logistic regression model f by iterative descent on $L_\phi(f)$.

Proof The proof is similar to that for the training set case.
Suppose we have a current estimate $f(x)$ and seek to improve it by minimizing $L_\phi(f+b)$ over b.
In the proof we assume that b is an arbitrary function, while in practice b will be chosen to best approximate the ideal f within the class $\mathcal{B}$.
Denote by $p_x = P[y=1|x]$ (the true value) and by $\hat{p}_x$ the "estimate"

$$\hat{p}_x = \frac{e^{f(x)}}{e^{f(x)} + e^{-f(x)}} \tag{37}$$

Assume again x is fixed. Then,

$$\begin{aligned} L_\phi(f+b) &= E[e^{-yf(x)-yb(x)}] \\ &= e^{-f(x)}e^{-b(x)}p_x + (1-p_x)e^{f(x)}e^{b(x)} \end{aligned}$$

Taking the derivative and setting it to 0 we obtain the new step:

$$b(x) = \frac{1}{2}\ln\frac{p_x e^{-f(x)}}{(1-p_x)e^{f(x)}} = \frac{1}{2}\left[\ln\frac{p_x}{1-p_x} - \ln\frac{\hat{p}_x}{1-\hat{p}_x}\right] \tag{38}$$

Note that if one could exactly obtain the b prescribed by (**??**) the iteration would not be necessary.

(Proof, continued)
More interesting than the exact form of b above is the optimization problem that leads to it. Denote $w(x, y) = e^{-yf(x)}$. Then, b is the solution of

$$b = \operatorname*{argmin}_{b \in \mathcal{B}} E_{P_{XY} w(X,Y)}[e^{-Yb}] \tag{39}$$

where $P_{XY} w(X, Y)$ denotes the (unnormalized) **twisted distribution** obtained by multiplying the original data distribution with $w(x, y)$. (Of course, one may have to put some restrictions on P_{XY} and $\mathcal{B}$ in order to obtain a proper distribution.) Finally, note that the new f is $f + b$ and the new weights are $w(x, y)e^{-yb(x)}$ which finishes the proof.
Hence, the REAL ADABOOST algorithm can be seen as a form of "noisy gradient" algorithm at the distribution level. (Note that the minimization in equation (**??**) is over both direction and scale of f.)

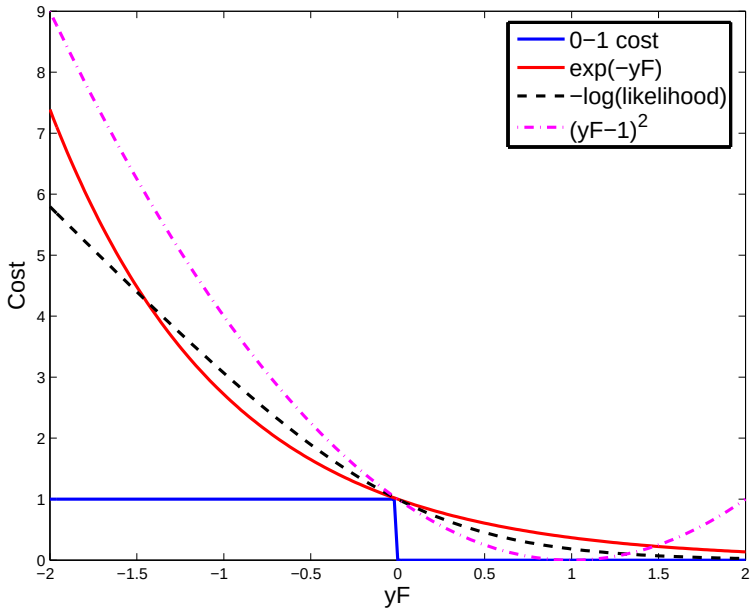
Why the e^{-yf} loss? and other L_ϕ losss

- We saw that L_ϕ is statistically motivated. Now we will see that it is computationally motivated as well.
- Recall: The "true" classification loss L_{01} is nonsmooth (has 0/no derivatives), non-convex. For training, one uses **surrogate losses** of the form $L_\phi(y, f) = \phi(yf)$.
- Want the following properties for ϕ
  - $\phi(z)$ is an upper bound of the 0–1 loss
  - $\phi(z)$ is smooth (has continuous derivatives of any order if f has them); (this lets us use continuous optimization techniques to fit the classifier)
  - $\phi(z)$ is convex (this leads to global optimization, which has been recognized as beneficial in practice; it also allows to prove bounds, rates of convergence and so on)
  - $\phi(z)$ is monothone (decreasing) (thus, when $z > 0$, driving the margins to increase even if the classification is correct).

  These properties are satisfied by $\phi(z) = e^{-z}$

Surrogate losses and boosting algorithms

A cornucopia of loss functions



(sometimes good to have $L(z)$ decrease for all $z < 0$, and sometimes bad – causes overfitting)

. . . and of boosting algorithms

- GentleAdaBoost: approx Newton, $\phi = e^{-z}$
- Least-SquaresBoost: $\phi = (1 - z)^2$ many operations in closed form
- LogitBoost $\phi = \ln(1 + e^{-z})$ slower (almost linear) decrease for $z \ll 0$
- AnyBoost, GradientBoost work with any ϕ

GradientBoost

GradientBoost Algorithm

Given $\mathcal{B}$ contains real-valued functions, loss L_ϕ, ϕ differentiable

Input M, labeled training set $\mathcal{D}$

Initialize $f^0(x) = \beta_0 = \text{argmin}_{\beta \in \mathbb{R}} \hat{L}(\beta)$

for $k = 0, 1, 2, \ldots M-1$

1. compute $r_i = -y^i \phi'(y^i f(x^i))$
2. fit $b^k(x)$ to outputs r_i
3. find $\beta_k = \text{argmin}_{\beta \in \mathbb{R}} \hat{L}(f^k + \beta b_k)$ (univariate optimization)

update $f^{k+1}(x) = f^k(x) + \beta^k b^k(x)$

Ouput $f^M(x)$

- Can be used for either classification or regression
- Works with any L
- If L convex, step 3 is convex optimization (efficient)
- Proposed first as AnyBoost, later specialized for $\mathcal{B}$ =decision/regression trees, with other tweaks and new name GradientBoost
- When $\mathcal{B}$ =CART
  - step 3 optimizes over every leaf separately
  - depth of trees J represents maximum number of interactions in f; should not be too large ($\mathcal{B}$ must be weak)

Practical aspects

Overfitting in noise When the classes overlap much (many examples in $\mathcal{D}$ hard/impossible to classify correctly) boosting algorithms tend to focus too much on the hard examples, at the expense of overall classification accuracy. The same happens for outliers. Observe also that the loss function(s) in the previous figure, which penalize more as the margin becomes more negative.

Choice of features Often times, the base class $\mathcal{B}$ consists of function of the form $b(x) = x_j - a$, which perform a split on coordinate x_j at point $x_j = a$. They have the advantage that they can be learned and evaluated extremely fast. One can also augment the coordinate vector x with functions of the coordinates (e.g. $x \rightarrow [x_1 \ \ldots \ x_d \ x_1x_2 \ x_1x_3 \ \ldots]$) essentially creating a large set of features, which corresponds to finite but very large $\mathcal{B}$. In such a situation, the number of features d can easily be larger than M the number of b's in the final f. Thus, boosting will be implicitly performing a feature selection task.