Machine Learning for biology

V. Monbet

UFR de Mathématiques
Université de Rennes 1

Outline

Ensemble methods (I)

Ensemble methods (II)

Outline

Ensemble methods (I)

Ensemble methods (II)

- Boosting
- Interlude: optimization algorithms
- Gradient boosting
- Boosting is a particular way of forward stagewise additive modeling (option)

Outline

Ensemble methods (II)

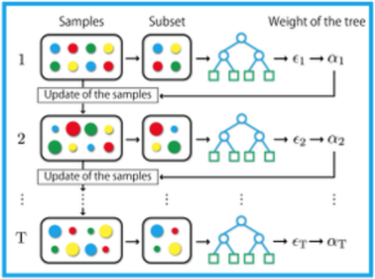
- Boosting
- Interlude: optimization algorithms
- Gradient boosting
- Boosting is a particular way of forward stagewise additive modeling (option)

Boosting

- The first boosting algorithm is due to Freund and Shapire (1996).
- It was developed for classification tasks.
- It consists in building a family of prediction rules which are aggregated.
- The algorithm is recursive: the rule at step m depends on the rule at step $m - 1$.

Boosting

- Boosting is one of the most powerful learning ideas introduced in the last twenty years.
- The first objective of boosting is to reduce bias.
- Boosting fits **sequentially** *m* models (weak learners) using weighted observations. At each step, examples that are badly predicted gain weight and examples that are classified correctly lose weight. Thus future models focus more on the examples that previous models have badly predicted.

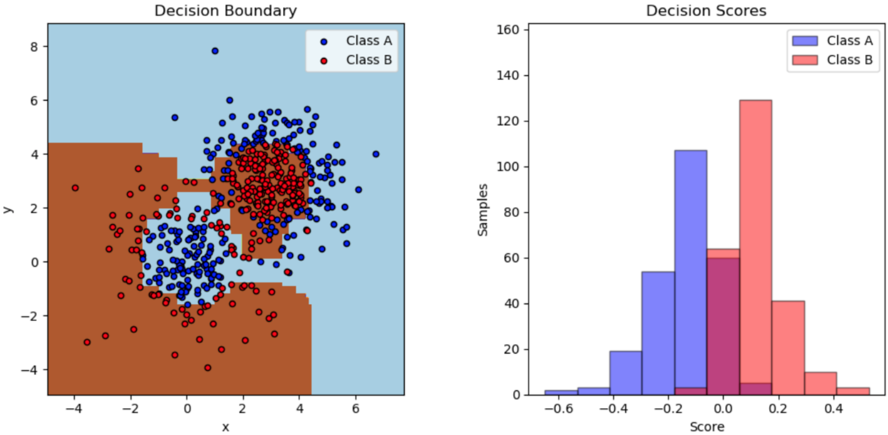

Samples
Subset
Weight of the tree
1
$\epsilon_1 \rightarrow \alpha_1$
Update of the samples
2
$\epsilon_2 \rightarrow \alpha_2$
Update of the samples
T
$\epsilon_T \rightarrow \alpha_T$

Learning Process

General comments

The main idea of boosting is to iteratively build models such that the model at iteration m characterize what the previous models have not described.

Boosting algorithm are local models and they are able to capture complex boundaries!



Example from sklearn documentation (200 trees, max depth=1)

Boosting loss functions

- Regression problems ($y \in \mathbb{R}$): Mean Square Error (MSE)

$$L(y, \mathbf{x}) = \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i))^2$$

  as usual (see the linear regression).
- Classification problems ($y \in \{-1, 1\}$): exponential loss function

$$L(y, \mathbf{x}) = \exp\left(-y_i f(\mathbf{x}_i)\right)$$

For instance, for knn ou decision trees,
the regular loos function is the classification error $L(y, \mathbf{x}) = \sum_{i=1}^{n} \mathbf{1}_{y_i \neq f(\mathbf{x}_i)}$.

This choice is discussed below.

Boosting algorithm

Let us denote L the loss function: RMSE for regression and $\exp(-yf(\mathbf{x}))$ for classification with $y \in \{-1, 1\}$.

Algorithm: boosting

- Initialization of observation weights $\omega_i^{(1)} = 1/n$, $i = 1, \cdots, n$
- For m=1 to M
    - (a) Fit a weak learner[1] $\hat{f}_m$ with weights $\omega_i^{(m)}$, $i = 1, \cdots, n$
    - (b) Compute the loss fonction for each observation and the global error:

$$\ell_m(i) = L\left(y_i, \hat{f}_m(\mathbf{x}_i)\right) \quad \text{and} \quad \text{err}_m = \sum_{i=1}^{n} \omega_i^{(m)} \ell_m(i)$$

    - (c) Compute a relative error $\alpha_m = \text{err}_m / (\sup_i \ell_m(i) - \text{err}_m)$
    - (d) Update the weigths $\omega_i^{(m+1)} \leftarrow \omega_i^{(m)} \alpha_m^{1-\ell_m(i)/\sup_i \ell_m(i)}$, $i = 1, \cdots, n$
    - (e) Normalise the updated weights $\omega_i^{(m+1)} = \frac{\omega_i^{(m+1)}}{\sum_{i=1}^n \omega_i^{(m+1)}}$, $i = 1, \cdots, n$
- Return $\hat{f}(x) = \sum_{m=1}^{M} \log\left(\frac{1}{\alpha_m}\right) \hat{f}_m(\mathbf{x})$

[1] for instance a low depth decision tree

Remark about the weights

- **Step (a)** *"Fit a weak learner $\hat{f}_m$ with weights"* of Adaboost algorithm requires the prediction rule/model to allow weighting

  For example, it is easy to consider weights for linear model by changing the MSE:

  $$L(y, \mathbf{x}) = \sum_{i=1}^{n} \omega_i \left(y_i - f(\mathbf{x}_i)\right)^2$$

  This can not be done for "local" models sucha as decision trees or k-nn because there is no global criteria to optimize.

  However, the model can also be fit on a sample drawn according to the weights $\omega_1, \cdots, \omega_n$: some observations are duplicated and other removed according to the value of the weights.

Some justification of Adaboost

Theorem (Hastie et al (2009), chap. 10)
Let $\mathcal{F}$ be a class of weak classifiers. It is assumed that step 2(a) of Adaboost algorithm consists in searching $f \in \mathcal{F}$ which minimize

$$\frac{1}{n}\sum_{i=1}^{n} \omega_i \mathbf{1}_{y_i \neq f(x_i)}.$$

Then Adaboost estimator at step m is given by

$$\hat{f}_b = \hat{f}_{b-1} + 2\beta_b F_b$$

with

$$(\beta_b, F_b) = \arg \min_{(\beta,f)\in\mathbb{R}\times\mathcal{F}} \sum_{i=1}^{n} \exp\left(-y_i(\hat{f}_{b-1}(x_i) + \beta f(x_i))\right)$$

.

Remark:

- Adaboost build a sequence of additive models.
- It minimize an empirial loss defined from $\exp(-yf(x))$

Theoretical and empirical loss functions

Here, we will illustrate why boosting use the function $\exp(-yf(x))$ as loss function. It is a general concept in statistical learning.

- Let $(\mathbf{X}, Y)$ be a pair of random variables in $\mathbb{R}^d \times \{-1, 1\}$
- Given $\mathcal{F}$ a family of models (or rules), we look for the best model in $\mathcal{F}$.
- In theory, we want to choose the model which minimizes a loss fonction, for instance

$$L(f) = P\big(Y \neq f(\mathbf{X})\big)$$

Problem: the loss function is intractable.

- Idea: Choose the model which minimizes the empirical loss function

$$L_n(f) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{f(\mathbf{x}_i)\neq y_i}$$

Convex risk

- **Problem**
  Function

$$\begin{array}{ccc} \mathbb{R}^n & \rightarrow & \mathbb{R} \\ (f(\mathbf{x}_1), \cdots, f(\mathbf{x}_n)) & \mapsto & \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{f(\mathbf{x}_i)\neq y_i} \end{array}$$

  is usually difficult to minimize by an optimization algorithm.

- **Idea**
  Find another loss function ℓ such that

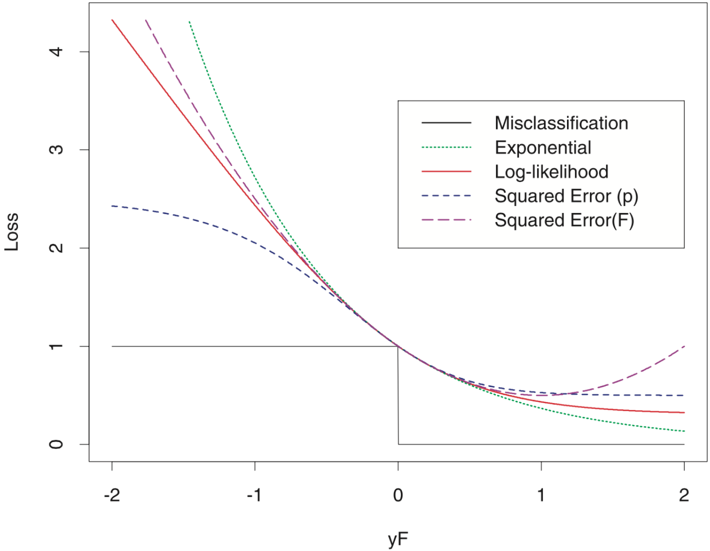
$$\begin{array}{ccc} \mathbb{R}^n & \rightarrow & \mathbb{R} \\ (f(\mathbf{x}_1), \cdots, f(\mathbf{x}_n)) & \mapsto & \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\mathbf{x}_i)) \end{array}$$

  is "easy" to minimize (ex: $v \mapsto \ell(u, v)$ convex).

Loss function

- The loss function $\ell(y, f(\mathbf{x}))$ measures the gap between $y \in \{-1, 1\}$ and $f(\mathbf{x})$.
- Thus, it has to be
  - large if $yf(\mathbf{x}) < 0$ (both have opposite signs → badly classified sample.))
  - small if $yf(\mathbf{x}) > 0$ (both have same sign → well classified sample.)
- Examples:
  1. $\ell(y, f(\mathbf{x})) = \mathbf{1}_{yf(\mathbf{x})<0}$
  2. $\ell(y, f(\mathbf{x})) = \exp(-yf(\mathbf{x}))$ (this function is convex according to its second argument)

Choice of loss function is important



Loss functions and robustness

- The principal attraction of **exponential loss** in the context of additive modeling is computational.
- The exponential loss function is a monotone decreasing function of the "margin" $yf(x)$.
- In classification (with a -1/1 response) the margin plays a role analogous to the residuals $(y - f(x))$ in regression.
  Observations with positive margin $y_i f(x_i) > 0$ are classified correctly whereas those with negative margin $y_i f(x_i) < 0$ are misclassified.
- The goal of the classification algorithm is to produce positive margins as frequently as possible. Any loss criterion used for classification should penalize negative margins more heavily than positive ones since positive margin observations are already correctly classified.
- **Misclassification loss** gives unit penalty for negative margin values, and no penalty at all for positive ones.
- **Exponential loss** continuously penalizes increasingly negative margin values more heavily than they reward increasingly positive ones.

‘

Generalization error & bias/variance tradeoff

Ideally, one wants to choose a model that both accurately captures the information in training data, but also generalizes well to unseen data.

- **Property** [Freund and Shapire, 1999]

$$L(\hat{f}_M) \leq L_n(\hat{f}_M) + O\left(\sqrt{\frac{MV}{n}}\right)$$

  where $L(\hat{f}_M)$ is the theoretical error, $L_n(\hat{f}_M)$ is the empirical error and V a variance. The theoritical error of boosting model $\hat{f}_M$ is bounded by the empirical error plus a variance term depending on the number of iterations M.

- The bias/variance tradeoff or approximation error/estimation tradeoff is controlled by the number of iterations M:
  - M small: the first term $L_n(\hat{f}_M)$(approximation) dominates
  - M big: the second term $\sqrt{\frac{MV}{n}}$ (estimation) dominates

- If M is too large, Adaboost leads to overfitting (see next slide).

- The choice of M is important.

Empirical error

- Let us denote err_m the error rate computed on the sample $\{x_1, \cdots, x_n\}$ for the model $\hat{f}_m$

$$err_m = \frac{\sum_{i=1}^n \omega_i^{(m)} \ell_m(i)}{\sum_{i=1}^n \omega_i^{(m)}}$$

  and γ_m the improvement of model $\hat{f}_m$ compared to the purely random model, then

$$err_m = 1/2 - \gamma_m$$

  for a classification task.

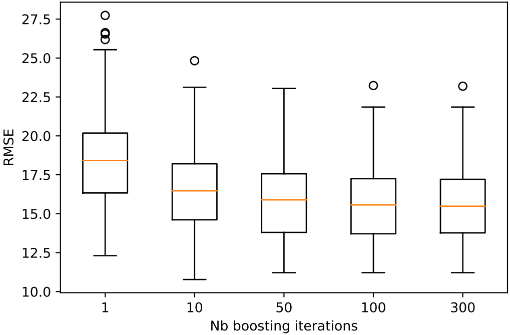
- **Property** [Freund and Shapire, 1997]

$$L_n(\hat{f}_M) \leq \exp\left(-2 \sum_{m=1}^{M} \gamma_m^2\right)$$

  Thus, the empirical error $L_n(\hat{f}_M)$ (computed on the data) tends to 0 when the number of iterations M increases.

Example for AdaBoost: Ozone (regression)

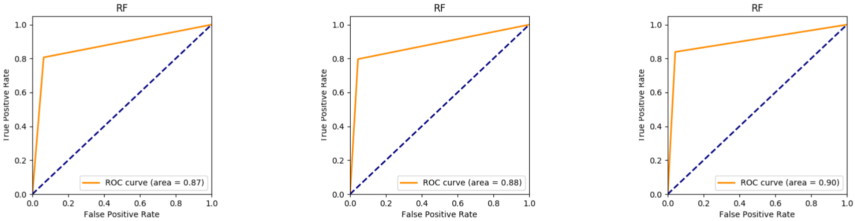
- AdaBoost
- Weak learner: tree, maximum depth=2
- Boosting decreases RMSE
- RMSE = VAR+BIAS^2.
  It is mainly the bias which is decreased (not shown).



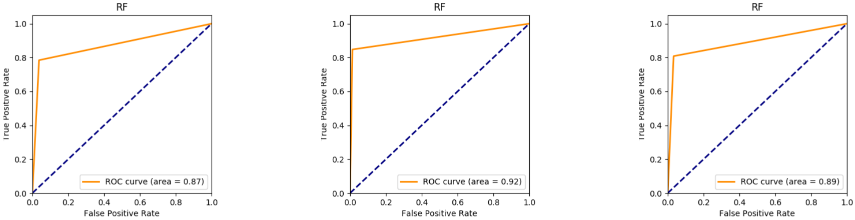
Adaboost performances, Leukemia (classification)

Trees depth = 2 (+ default parameters)

10 trees 100 trees 500 trees



Trees depth = 6



Outline

Ensemble methods (II)

Boosting

Interlude: optimization algorithms

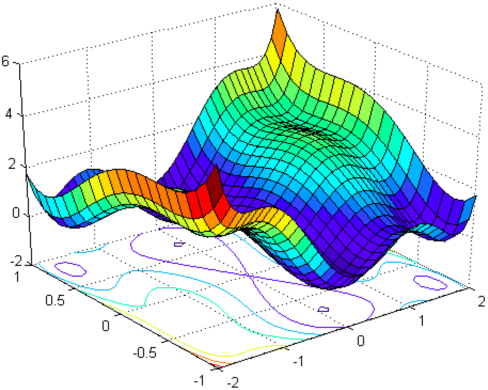
Gradient boosting

Boosting is a particular way of forward stagewise additive modeling (option)

Optimization problem

In Machine Learning, what ever the method we choose, we have to minimize a loss function.

In other words, during the training process, we tweak and change the parameters (weights) of our model to try and minimize that loss function, and make our predictions as correct and optimized as possible. But how exactly do you do that? How do you change the parameters of your model, by how much, and when?



Source site:

Gradient descent

Generally, if we want to find the minimum of a function, we set the derivative to zero and solve for the parameters.
It is done for instance solving a linear regression problem.

It turns out, however, it is impossible to get a closed form solution in many Machine Learning methods.

This is where optimizers come in. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by futzing with the weights. The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.

We iteratively search for a minimum using a method called gradient descent.

Gradient descent image

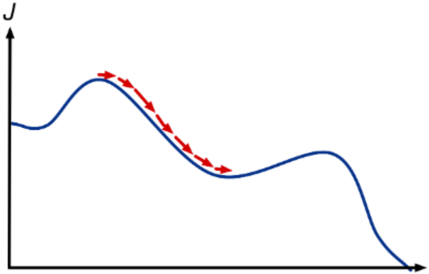
As a visual analogy, you can think of a hiker trying to get down a mountain with a blindfold on. It's impossible to know which direction to go in, but there's one thing she can know: if she's going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she'll reach the base.

Source site: `https://www.deepideas.net/deep-learning-from-scratch-iv-gradient-descent-and-backpropagation/`

Gradient descent ideas

Gradient descent operates in a similar way when trying to find the minimum of a function: It starts at a random location in parameter space and then iteratively reduces the error J until it reaches a local minimum.

At each step of the iteration, it determines the direction of steepest descent and takes a step along that direction. This process is depicted for the 1-dimensional case in the following image.



Source site: `https://www.deepideas.net/deep-learning-from-scratch-iv-gradient-descent-and-backpropagation/`

Gradient descent algorithm

As you might remember, the direction of steepest ascent of a function at a certain point is given by the gradient at that point. Therefore, the direction of steepest descent is given by the negative of the gradient.

So now we have a rough idea how to minimize J:

1. Start with random values for the parameters (or weights) β
2. Calculate what a small change in each individual weight would do to the loss function

$\xrightarrow{math}$ Compute the gradient of J with respect to β.

3. Adjust each individual weight based on its gradient

$\xrightarrow{math}$ Take a small step along the direction of the negative gradient

4. Go back to 2

The tricky part of this algorithm (and optimizers in general) is understanding gradients, which represent what a small change in a weight or parameter would do to the loss function. Gradients are partial derivatives, and are a measure of change.

$$\nabla J(\beta_j) \simeq \frac{J(\beta_j + h) - J(\beta_j)}{h} \quad \text{for } h \text{ small}$$

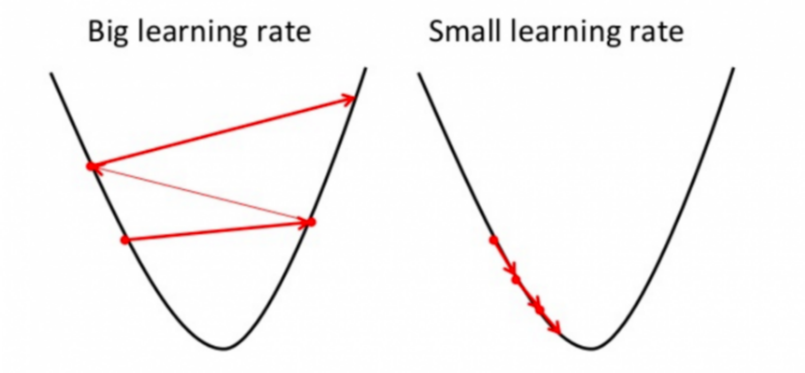
Local/global maxima

One hiccup that you might experience during optimization is getting stuck on local minima. When dealing with high dimensional data sets (lots of variables) it's possible you'll find an area where it seems like you've reached the lowest possible value for your loss function, but it's really just a local minimum.

In the vein of the hiker analogy theme, this is like finding a small valley within the mountain you're climbing down. It appears that you've reached bottom – getting out of the valley requires, counterintuitively, climbing – but you have'nt.

Learning rate

To avoid getting stuck in local minima, we make sure we use the proper learning rate.



Learning rate

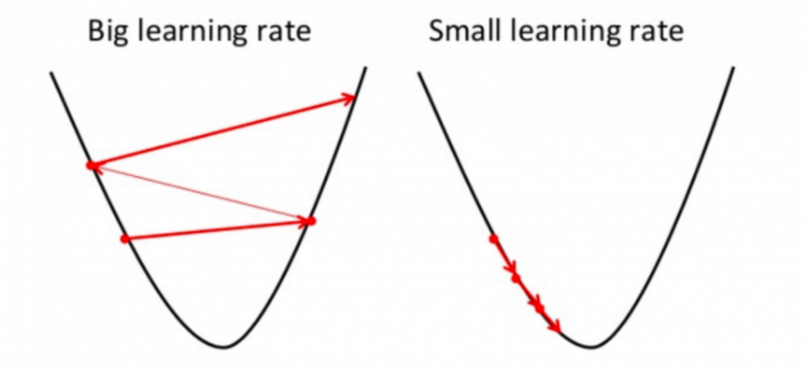
Changing our weights too fast by adding or subtracting too much (i.e. taking steps that are too large) can hinder our ability to minimize the loss function. We don't want to make a jump so large that we skip over the optimal value for a given weight.

To make sure that this doesn't happen, we use a variable called "the learning rate." This thing is just a very small number, usually something like 0.001, that we multiply the gradients by to scale them. This ensures that any changes we make to our weights are pretty small. In math talk, taking steps that are too large can mean that the algorithm will never converge to an optimum.

At the same time, we don't want to take steps that are too small, because then we might never end up with the right values for our weights. In math talk, steps that are too small might lead to our optimizer converging on a local minimum for the loss function, but not the absolute minimum.

For a simple summary, just remember that the learning rate ensures that we change our weights at the right pace, not making any changes that are too big or too small.

Learning Rate



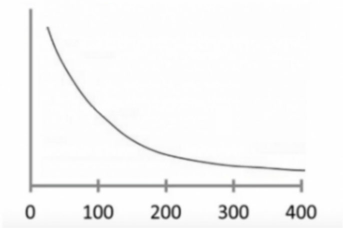
Gradient descent algorithm with learning rate α

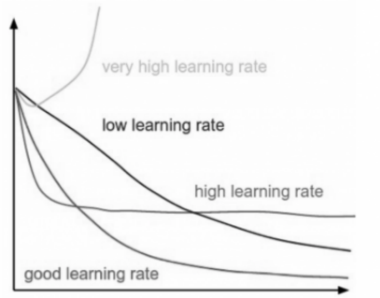
1. Sample randomly an initial value $\beta_{[0]}$ for the weights
2. Compute the gradient $\nabla J(\beta_{[r-1]})$ of the loss function J at the current value of the weights $\beta_{[r-1]}$
3. Update the weights:

$$\beta_{[r]} = \beta_{[r-1]} - \alpha \nabla J(\beta_{[r-1]})$$

4. Go back to 2. until convergence.

A good way to make sure gradient descent runs properly is by plotting the cost function as the optimization runs. Put the number of iterations on the x axis and the value of the cost-function on the y axis.





When gradient descent can't decrease the cost-function anymore and remains more or less on the same level, it has converged. The number of iterations gradient descent needs to converge can sometimes vary a lot. It can take 50 iterations, 60,000 or maybe even 3 million, making the number of iterations to convergence hard to estimate in advance.

Let's practice!

Just copy the url below, paste it on your favorite web navigator

`https://developers.google.com/machine-learning/crash-course/reducing-loss/playground-exercise`

and follow the instructions.

Outline

Ensemble methods (II)

- Boosting
- Interlude: optimization algorithms
- Gradient boosting
- Boosting is a particular way of forward stagewise additive modeling (option)

Ideas

- $(\mathbf{X}, Y)$ in $\mathbb{R}^d \times \{-1, 1\}$, a loss function ℓ and we have to approximate

$$f^* = \arg\min_{f \in \mathcal{F}} E\left(\ell(Y, f(\mathbf{X}))\right)$$

- Strategy: given a sample $\mathcal{S}_n = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_n, y_n)\}$, the empirical version of $E\left(\ell(Y, f(\mathbf{X}))\right)$ is minimized:

$$\frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\mathbf{x}_i))$$

- Recursive approach: f^* is approximated by

$$\hat{f} = \sum_{m=1}^{M} f_m(\mathbf{x})$$

where the models f_m are fit in a recursive way.

- Method: use a numerical approach (gradient descent)

Gradient descent

- Let us denote $\mathbf{f_m} = (f_m(\mathbf{x}_1), \cdots, f_m(\mathbf{x}_n))$ and

$$J(\mathbf{f_m}) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, \mathbf{f_m}(\mathbf{x}_i))$$

- The recurrence formula of gradient descent algorithms is

$$\mathbf{f_m} = \mathbf{f_{m-1}} - \alpha \nabla J(\mathbf{f_m})$$

  where α is the learning rate.

Drawbacks of this algorithm

- This algorithm allows to compute the estimator only on the observed points $\mathbf{x}_1, \cdots, \mathbf{x}_n$
- It does not take advantage of specific regularity properties of the function to be estimated

Gradient descent

- At each step, approximated values are obtained:

$$f_m(\mathbf{x}_i) = f_{m-1}(\mathbf{x}_i) - \alpha U_i$$

  with, for all $i = 1, \cdots, n$

$$U_i = -\frac{\partial}{\partial f(\mathbf{x_i})} \ell(y_i, f(\mathbf{x}_i))_{|f(\mathbf{x}_i)=f_{m-1}(\mathbf{x_i})}$$

  For example, if $\ell(y_i, f(\mathbf{x})) = \exp(-yf(\mathbf{x}))$, then $U_i = -y_i \exp(-y_i f_{m-1}(\mathbf{x}_i))$
- To compute f_m at a given $\mathbf{x}$, a regression can be performed on the points $(\mathbf{x}_1, U_1), \cdots, (\mathbf{x}_n, U_n)$.

Gradient Boosting Algorithm (Friedman, 2001)

- $\mathcal{S}_n = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_n, y_n)\}$ and $0 < \alpha \leq 1$ a regularization parameter
- h a simple regression model

- Initialization: $f_0(.) = \arg\min_c \frac{1}{n}\ell(y_i, c)$
- For $m = 1, \cdots, M$:
  (a) Compute $-\frac{\partial}{\partial f(\mathbf{x_i})}\ell(y_i, f(\mathbf{x}_i))$ at the points $f_m(\mathbf{x}_i)$
  (b) Fit the simple regression model h_m on the sample $(\mathbf{x}_1, U_1), \cdots, (\mathbf{x}_n, U_n)$
  (c) Update $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \alpha h_m(\mathbf{x})$

Gradient Boosting Algorithm, comments

- The output of the Gradient Boosting Algorithm is an aggregated function

$$\hat{f}_M = f_0 + \alpha \sum_{m=1}^{M} \hat{f}_m$$

- If $\hat{f}_M(\mathbf{x})$ is a real; in classification task, $\hat{y} = sign(\hat{f}_M(\mathbf{x}))$
- When $\alpha = 1$ and $\ell(y, f(\mathbf{x})) = exp(-yf(\mathbf{x}))$, the gradient boosting is very close to Adaboost.
- Choice of α is linked to the choice of M; they control the rate at which the function

$$\frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\mathbf{x}_i))$$

  is minimized.
- If α is too low, the computational time is long.
- If M is too large, overftting occurs.
  M can be estimated using the out-of-bag samples.

Gradient Boosting Algorithm, comments

- As for Adaboost, the model used in gradient boosting should be a weak classifier.
- Usually the algorithm is more performant if the weak learner has a large bias but a low variance.
- If trees are used, their depth should be low.
- To decrease variance (and increase bias), only a sample of features can be used at each step (like in bagging algorithms).

Gradient Boosting for regression

- For a regression task,

$$\ell(y, f(\mathbf{x})) = \frac{1}{2}\,(y - f(\mathbf{x}))^2$$

  is a convex function with respect to $f(\mathbf{x})$.

- The gradients are

$$U_i = -\frac{\partial}{\partial f(\mathbf{x_i})}\ell(y_i, f(\mathbf{x}_i))_{|f(\mathbf{x}_i)} = y_i - f_{m-1}(x_i).$$

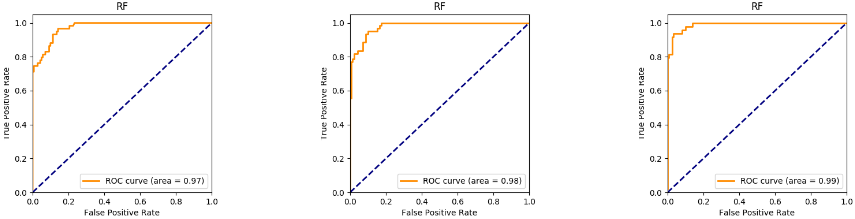
- They are the residuals of the regression at step $m-1$, so that f_m tries to explain the residual information of f_{m-1}.

- It can be shown, that under some assumptions, at each step the bias decreases but the variance tends to increase.

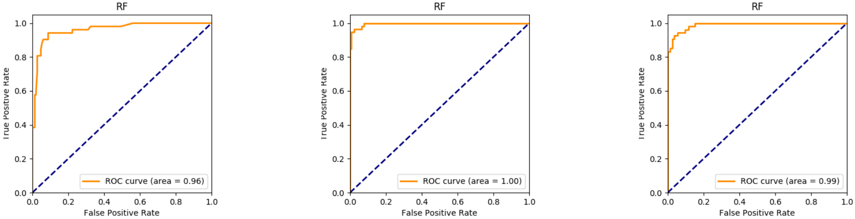
Gradient boosting performances, Leukemia (classification)

Trees depth = 2, Exponential loss, Max features=3, default learning rate

10 trees 100 trees 500 trees



Trees depth = 6, Exponential loss, Max features=10, default learning rate

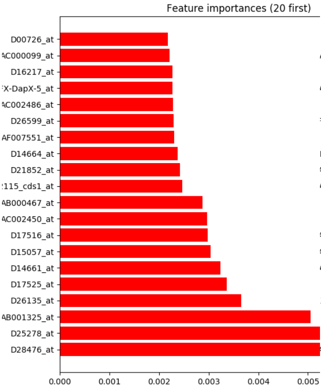


Variable importance, Leukemia (classification)

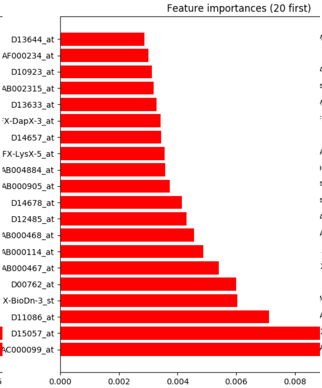
Let us compare the importance of variables for Gradient Boosting and for a unique tree.

The "selection" is not the same.

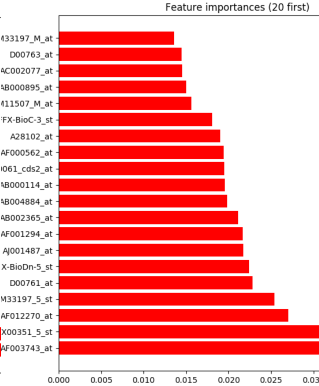
1000 trees, depth=2



1000 trees, depth=6

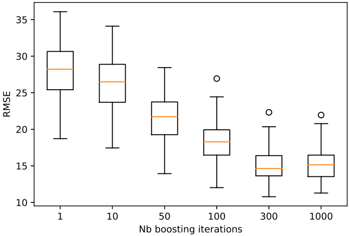


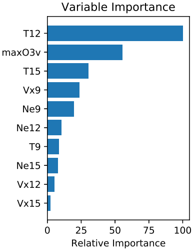
1 tree



Example for Gradient Boosting: Ozone (Regression)

- Gradient Boosting algorithm is more efficient than AdaBoost.
- Weak learner: tree, maximum depth=2





eXtreme Gradient Boosting

eXtreme Gradient Boosting is an optimized version of gradient boosting.

- Model Features
  - Stochastic Gradient Boosting with sub-sampling at the row, column and column per split levels.
  - Regularized Gradient Boosting with both L1 and L2 regularization.
- System Features
  - Parallelization of tree construction using all of your CPU cores during training.
  - Distributed Computing for training very large models using a cluster of machines.
  - Out-of-Core Computing for very large datasets that don't fit into memory.
  - Cache Optimization of data structures and algorithm to make best use of hardware.
- Algorithm Features
  - Sparse Aware implementation with automatic handling of missing data values.
  - Block Structure to support the parallelization of tree construction.
  - Continued Training so that you can further boost an already fitted model on new data.

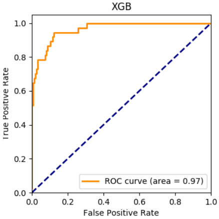
eXtreme Gradient boosting performances, Leukemia (classification)

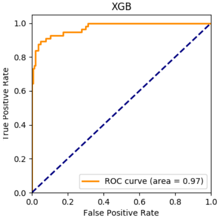
Trees depth = 2, column sample by tree=.3

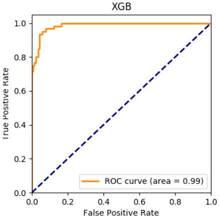
10 trees

100 trees

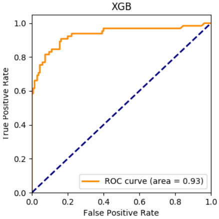
500 trees

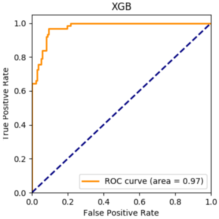


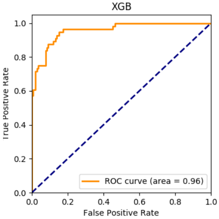




Trees depth = 6: to much trees may lead to over-fitting





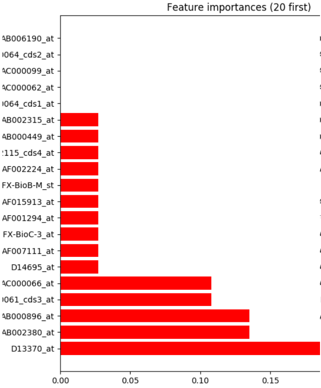


Variable importance, Leukemia (classification)

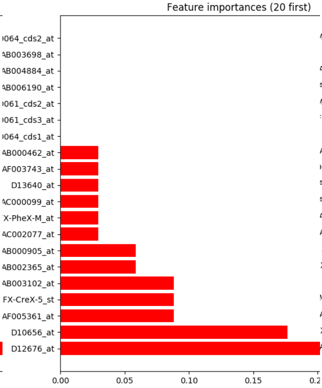
Let us compare the importance of variables for eXtreme Gradient Boosting and for a unique tree.

The "selection" is not the same.

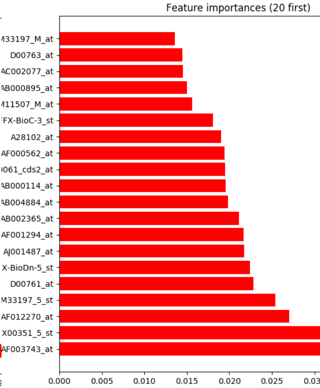
1000 trees, depth=2



1000 trees, depth=6

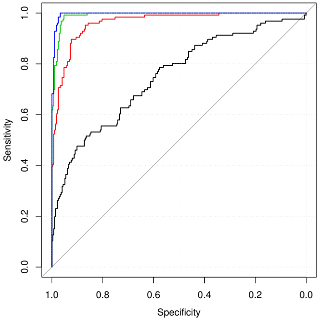


1 tree



Example for Gradient Boosting: Leukemia

- Gradient Boosting algorithm is more efficient than AdaBoost.
- Weak learner: tree, maximum depth=2



n.trees = 100, AUC = 0.750

n.trees = 250, AUC = 0.963

n.trees = 500, AUC = 0.992

n.trees = 1000, AUC = 0.996

Outline

Ensemble methods (II)

Boosting

Interlude: optimization algorithms

Gradient boosting

Boosting is a particular way of forward stagewise additive modeling (option)

Boosting as a particular way of forward stagewise additive modeling

- Idea: fit nested models such that, at step m, a model is fit to the residuals of the previous step

$$\text{res}_{m-1}(i) = y_i - \hat{f}_{m-1}(x_i) = \beta_m^* b(x_i;\gamma_m^*) + \epsilon_i$$

where $b(.;\gamma)$ is a basis function

$$(\beta_m^*,\gamma_m^*) = \arg\min_{\beta_{m+1},\gamma_{m+1}} \sum_{i=1}^n L\left(y_i - \hat{f}_{m-1}(x_i), \beta_m b(x_i;\gamma_m)\right)$$

- More generally, the models are fit by solving

$$\min_{\{\beta_m,\gamma_m\}_{m=1}^M} \sum_{i=1}^n L\left(y_i, \sum_{m=1}^M \beta_m b(x_i;\gamma_m)\right)$$

- The boosting *AdaBoost* algorithm (with $y \in \{-1,1\}$) is a particular case of Forward Stagewise Additive Modeling for the exponential loss function.
- In *AdaBoost*, the basis functions are weak classifiers[2] $G(\mathbf{x}) \in \{-1,1\}$. And, at step m,

$$(\beta_m, G_m) = \arg\min_{\beta,G} \sum_{i=1}^n \exp\left(-y_i(f_{m-1}(\mathbf{x}_i) + \beta G(\mathbf{x}_i))\right)$$

[2] See Hastie, p.343

Boosting as a particular forward stagewise additive modeling (continue)

- The boosting *AdaBoost* algorithm (with $y \in \{-1, 1\}$) is a particular case of Forward Stagewise Additive Modeling for the exponential loss function.
- In *AdaBoost*, the basis functions are weak classifiers $G(\mathbf{x}) \in \{-1, 1\}$. And, at step m,

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{n} \exp\left(-y_i(f_{m-1}(\mathbf{x}_i) + \beta G(\mathbf{x}_i))\right)$$

- With $\omega_i^{(m)} = \exp(-y_i f_{m-1}(\mathbf{x}_i))$, it leads to

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{n} \omega_i^{(m)} \exp\left(-y_i \beta G(\mathbf{x}_i)\right)$$

  Since $\omega_i^{(m)}$ does not depend on β nor G it can be interpreted as a wieght applied to each observation.
  The weight depends on f_{m-1} so that it will be updated at each iteration.

Boosting as a particular forward stagewise additive modeling (continue)

- Now, the minimizing problem

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{n} \exp\left(-y_i(f_{m-1}(\mathbf{x}_i) + \beta G(\mathbf{x}_i))\right)$$

can be solved in 2 steps. Firstly, since $-y_i G(\mathbf{x}_i) = \mathbb{I}(y_i \neq G(\mathbf{x}_i))$,

$$G_m = \arg\min_{G} \sum_{i=1}^{n} \omega_i^{(m)} \mathbb{I}(y_i \neq G(\mathbf{x}_i))$$

which is the classifier that minimizes the weighted error.

- If the exponential loss fonction is considered,

$$\begin{aligned}\sum_{i=1}^{n} \omega_i^{(m)} \exp\left(-y_i \beta G(\mathbf{x}_i)\right) &= e^{-\beta} \sum_{y_i = G(\mathbf{x}_i)} \omega_i^{(m)} + e^{\beta} \sum_{y_i \neq G(\mathbf{x}_i)} \omega_i^{(m)} \\ &= (e^{\beta} - e^{-\beta}) \sum_{i=1}^{n} \omega_i^{(m)} \mathbb{I}(y_i \neq G(\mathbf{x}_i)) + e^{\beta} \sum_{i=1}^{n} \omega_i^{(m)}\end{aligned}$$

and minimizing the last line in β leads to

$$\beta = \frac{1}{2} \log \frac{1 - \mathrm{err}_m}{\mathrm{err}_m} \quad \text{where} \quad \mathrm{err}_m = \frac{\sum_{i=1}^{n} \omega_i^{(m)} \mathbb{I}(y_i \neq G(\mathbf{x}_i))}{\sum_{i=1}^{n} \omega_i^{(m)}}$$

Boosting as a particular forward stagewise additive modeling (continue)

- The approximation of the forward stagewise additive modeling is updated

  $$f_m = f_{m-1} + \beta G_m$$

  which causes the weights for the next iteration to be

  $$\omega_i^{(m)} = \omega_i^{(m-1)} \exp\left(-y_i \beta_m G(\mathbf{x}_i)\right)$$

  Using that $-y_i G(\mathbf{x}_i) = \mathbb{I}(y_i \neq G(\mathbf{x}_i))$, it becomes

  $$\omega_i^{(m)} = \omega_i^{(m-1)} e^{-\beta_m} e^{\alpha_m \mathbb{I}(y_i \neq G(\mathbf{x}_i))}$$

  where $\alpha_m = 2\beta_m$ is the quantity defined in the algorithm.
- Finally, Adaboost is a forward stagewise additive model associated to the exponential loss function.