Gradient Boosting

David S. Rosenberg

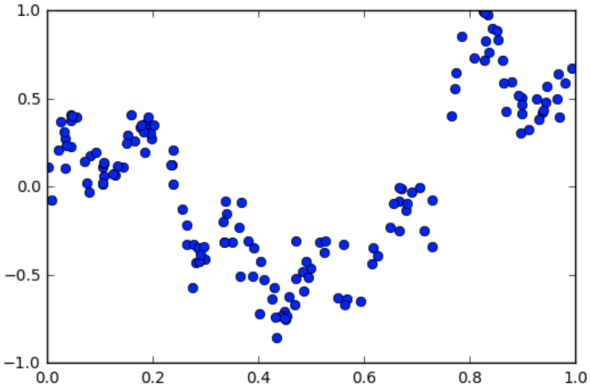
Bloomberg ML EDU

November 28, 2017

Introduction

Nonlinear Regression

- Suppose we have the following regression problem:



- What are some options?
- basis functions, kernel methods, trees, neural nets, ...

Linear Model with Basis Functions

- Choose some basis functions on input space $\mathcal{X}$:

$$g_1, \ldots, g_M : \mathcal{X} \to \mathbf{R}$$

- Predict with linear combination of basis functions:

$$f(x) = \sum_{m=1}^{M} \nu_m g_m(x)$$

- Can fit this using standard methods for linear models (e.g. least squares, lasso, ridge, etc.)
- In ML parlance, basis functions are called **features** or **feature functions.**

Not Limited to Regression

- Linear combination of basis functions:

$$f(x) = \sum_{m=1}^{M} \nu_m g_m(x)$$

- $f(x)$ is a number — for regression, it's exactly what we're looking for.
- Otherwise, $f(x)$ is often called a **score** function.
- It can be
  - thresholded to get a classification
  - transformed to get a probability
  - transformed to get a parameter of a probability distribution (e.g. Poisson regression)
  - used for ranking search results

Adaptive Basis Function Model

- Let's "learn" the basis functions.
- **Base hypothesis space** $\mathcal{H}$ consisting of functions $h : \mathcal{X} \to \mathbf{R}$.
  - We will choose our "basis functions" or "features" from this set of functions.
- An **adaptive basis function expansion** over $\mathcal{H}$ is

$$f(x) = \sum_{m=1}^{M} \nu_m h_m(x),$$

where $v_m \in \mathbf{R}$ and $h_m \in \mathcal{H}$ are chosen based on training data.

Adaptive Basis Function Model

- **Base hypothesis space**: $\mathcal{H}$ of **real-valued functions**
- **Combined hypothesis space:** $\mathcal{F}_M$:

$$\mathcal{F}_M = \left\{ \sum_{m=1}^{M} v_m h_m(x) \mid v_m \in \mathbf{R},\, h_m \in \mathcal{H},\, m = 1, \ldots, M \right\}$$

- Suppose we're given some data $\mathcal{D} = ((x_1, y_1), \ldots, (x_n, y_n))$.
- Learning is choosing $v_1, \ldots, v_M \in \mathbf{R}$ and $h_1, \ldots, h_M \in \mathcal{H}$ to fit $\mathcal{D}$.

Empirical Risk Minimization

- We'll consider learning by **empirical risk minimization**:

$$\hat{f} = \arg\min_{f \in \mathcal{F}_M} \frac{1}{n} \sum_{i=1}^{n} \ell\left(y_i, f(x_i)\right),$$

  for some **loss function** $\ell(y, \hat{y})$.
- Write ERM objective function as

$$J(v_1, \ldots, v_M, h_1, \ldots, h_M) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(y_i, \sum_{m=1}^{M} v_m h_m(x)\right).$$

- How to optimize J? i.e. how to learn?

Gradient-Based Methods

- **Suppose** our base hypothesis space is parameterized by $\Theta = \mathbf{R}^b$:

$$J(v_1, \ldots, v_M, \theta_1, \ldots, \theta_M) = \frac{1}{n} \sum_{i=1}^{n} \ell \left(y_i, \sum_{m=1}^{M} v_m h(x; \theta_m) \right).$$

- Can we can differentiate J w.r.t. v_m's and θ_m's? Optimize with SGD?
- For **some** hypothesis spaces and typical loss functions, yes!
- Neural networks fall into this category! ($h_1, \ldots, h_M$ are neurons of last hidden layer.)

What if Gradient Based Methods Don't Apply?

- What if base hypothesis space $\mathcal{H}$ consists of decision trees?
- Can we even parameterize trees with $\Theta = \mathbf{R}^b$?
- Even if we could for some set of trees,
  - predictions would not change continuously w.r.t. $\theta \in \Theta$,
  - and so certainly not differentiable.

- Today we'll discuss **gradient boosting**. It applies whenever
  - our loss function is [sub]differentiable w.r.t. training predictions $f(x_i)$, and
  - we can do regression with the base hypothesis space $\mathcal{H}$ (e.g. regression trees).

Overview

- Forward stagewise additive modeling (FSAM)
  - example: L^2 Boosting
  - example: exponential loss gives AdaBoost
  - Not clear how to do it with many other losses, including logistic loss
- Gradient Boosting
  - example: logistic loss gives BinomialBoost
- Variations on Gradient Boosting
  - step size selection
  - stochastic row/column selection
  - Newton step direction
  - XGBoost

Forward Stagewise Additive Modeling

Forward Stagewise Additive Modeling (FSAM)

- FSAM is an iterative optimization algorithm for fitting adaptive basis function models.
- Start with $f_0 \equiv 0$.
- After $m-1$ stages, we have

$$f_{m-1} = \sum_{i=1}^{m-1} \nu_i h_i.$$

- In m'th round, we want to find
  - **step direction** $h_m \in \mathcal{H}$ (i.e. a basis function) and
  - **step size** $\nu_i > 0$
- such that

$$f_m = f_{m-1} + \nu_i h_m$$

  improves objective function value by as much as possible.

Forward Stagewise Additive Modeling for ERM

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M:
   1. Compute:
   $$(\nu_m, h_m) = \underset{\nu \in \mathbf{R}, h \in \mathcal{H}}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \ell \left(y_i, f_{m-1}(x_i) \underbrace{+ \nu h(x_i)}_{\text{new piece}} \right).$$
   2. Set $f_m = f_{m-1} + \nu_m h$.
3. Return: f_M.

Example: L^2 Boosting

Example: L^2 Boosting

- Suppose we use the **square loss**. Then in each step we minimize

$$J(v, h) = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \left[f_{m-1}(x_i) \underbrace{+ v h(x_i)}_{\text{new piece}} \right] \right)^2$$

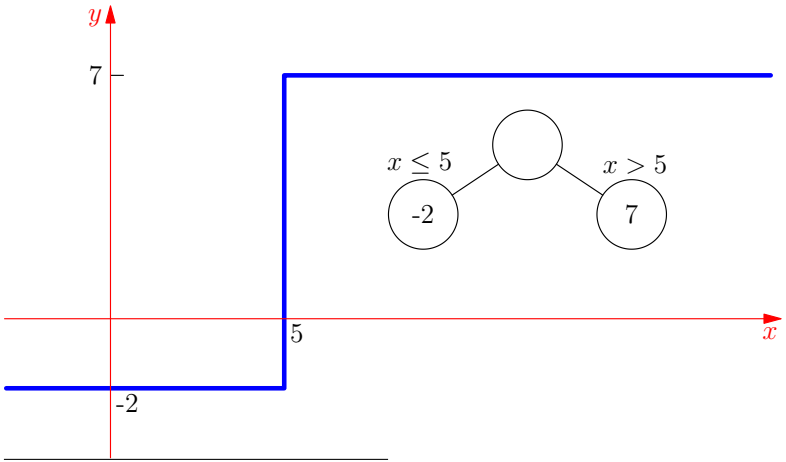
- If $\mathcal{H}$ is closed under rescaling (i.e. if $h \in \mathcal{H}$, then $vh \in \mathcal{H}$ for all $h \in \mathbf{R}$), then don't need v.
- Take $v = 1$ and minimize

$$J(h) = \frac{1}{n} \sum_{i=1}^{n} \left([y_i - f_{m-1}(x_i)] - h(x_i) \right)^2$$

- This is just fitting the residuals with least-squares regression!
- If we can do regression with our base hypothesis space $\mathcal{H}$, then we're set!

Recall: Regression Stumps

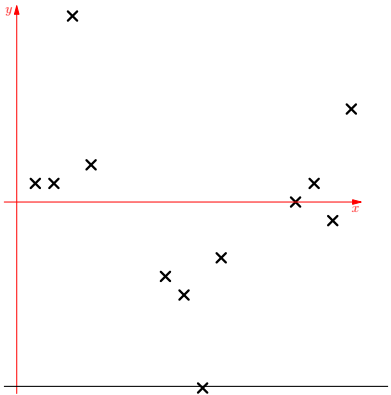
- A **regression stump** is a function of the form $h(x) = a1(x_i \leqslant c) + b1(x_i > c)$



Plot courtesy of Brett Bernstein.

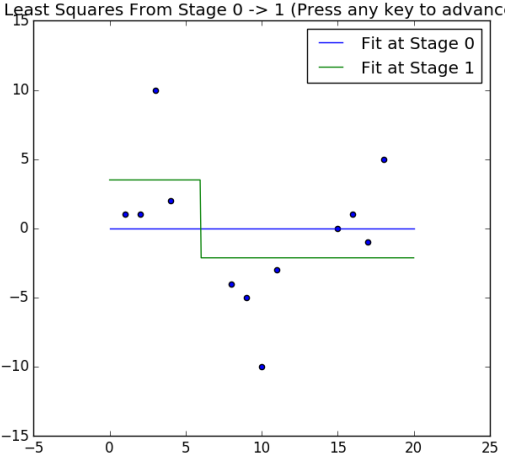
L^2 Boosting with Decision Stumps: Demo

- Consider FSAM with L^2 loss (i.e. L^2 Boosting)
- For base hypothesis space of **regression stumps**
- Data we'll fit with code:



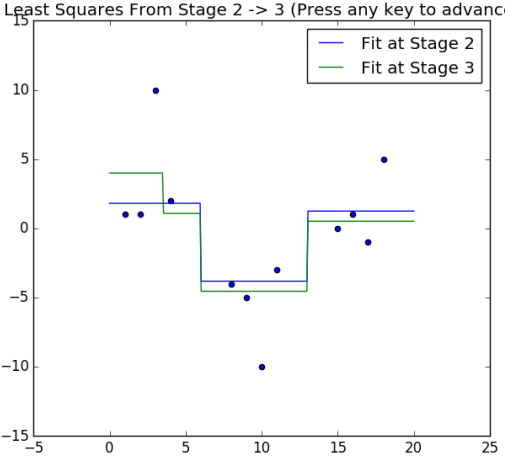
Plot courtesy of Brett Bernstein.

L^2 Boosting with Decision Stumps: Results



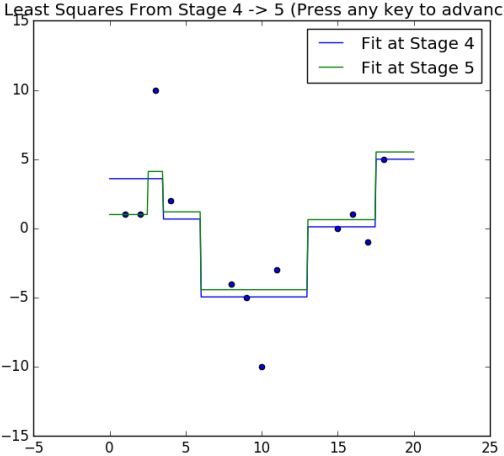
Least Squares From Stage 1 -> 2 (Press any key to advance)

L^2 Boosting with Decision Stumps: Results



Least Squares From Stage 3 -> 4 (Press any key to advance)

L^2 Boosting with Decision Stumps: Results



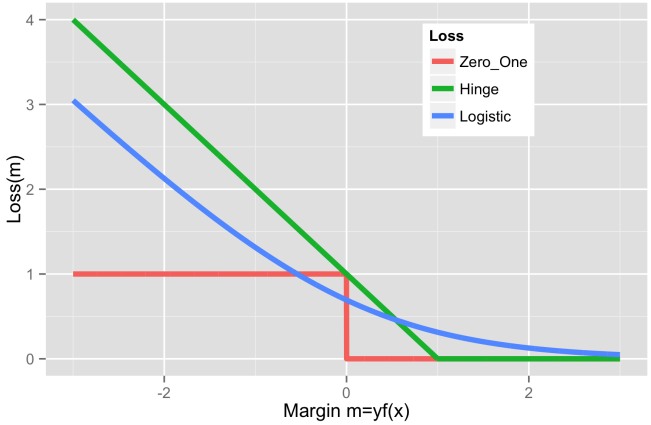
Least Squares From Stage 49 -> 50 (Press any key to advance)

Example: AdaBoost

The Classification Problem

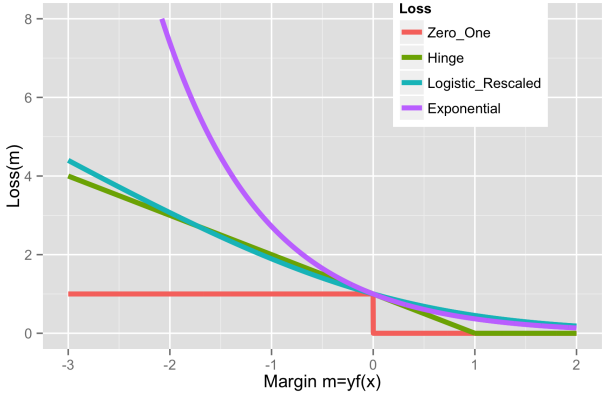
- Outcome space $\mathcal{Y} = \{-1, 1\}$
- Action space $\mathcal{A} = \mathbf{R}$
- Score function $f : \mathcal{X} \to \mathcal{A}$.
- Margin for example (x, y) is $m = yf(x)$.
  - $m > 0 \iff$ classification correct
  - Larger m is better.

Margin-Based Losses for Classification



Exponential Loss

- Introduce the **exponential loss**: $\ell(y, f(x)) = \exp(-yf(x))$.



FSAM with Exponential Loss

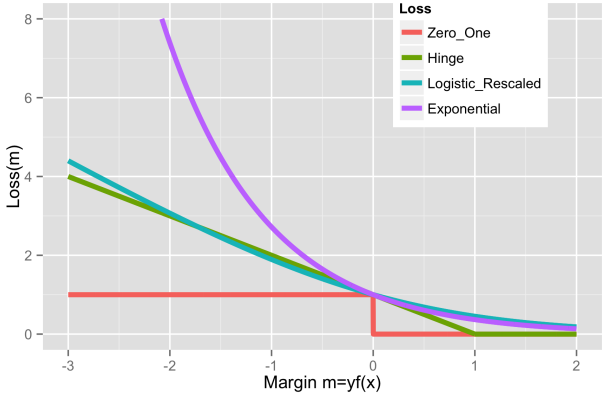
- Consider classification setting: $\mathcal{Y} = \{-1, 1\}$.
- Take loss function to be the **exponential loss**:

$$\ell(y, f(x)) = \exp(-yf(x)).$$

- Let $\mathcal{H}$ be a base hypothesis space of classifiers $h : \mathcal{X} \to \{-1, 1\}$.
- Then Forward Stagewise Additive Modeling (FSAM) reduces to a version of **AdaBoost**.
- Proof on Spring 2017 Homework #6, Problem 4 (and see HTF Section 10.4).

Exponential Loss

- Note that exponential loss puts a very large weight on bad misclassifications.



AdaBoost / Exponential Loss: Robustness Issues

- When Bayes error rate is high (e.g. $\mathbb{P}(f^*(X) \neq Y) = 0.25$)
  - e.g. there's some intrinsic randomness in the label
  - e.g. training examples with same input, but different classifications.
- Best we can do is predict the most likely class for each X.
- Some training predictions **should be wrong**,
  - because example doesn't have majority class
  - AdaBoost / exponential loss puts a lot of focus on getting those right
- Empirically, AdaBoost has degraded performance in situations with
  - high Bayes error rate, or when there's
  - high "**label noise**"
- Logistic loss performs better in settings with high Bayes error

FSAM for Other Loss Functions

- We know how to do FSAM for certain loss functions
  - e.g square loss, absolute loss, exponential loss,...
- In each case, happens to reduce to another problem we know how to solve.
- However, not clear how to do FSAM in general.
- For example, logistic loss / cross-entropy loss?

Gradient Boosting / "Anyboost"

FSAM Is Iterative Optimization

- The FSAM step

$$(\nu_m, h_m) = \underset{\nu \in \mathbf{R}, h \in \mathcal{H}}{\arg\min} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i) \underbrace{+\nu h(x_i)}_{\text{new piece}}\right).$$

- Hard part: finding the **best step direction** h.
- What if we looked for the **locally best** step direction?
  - like in gradient descent

"Functional" Gradient Descent

- We want to minimize
$$J(f) = \sum_{i=1}^{n} \ell\left(y_i, f(x_i)\right).$$
- In some sense, we want to take the gradient w.r.t. "f", whatever that means.
- $J(f)$ only depends on f at the n training points.
- Define
$$\mathbf{f} = \left(f(x_1), \ldots, f(x_n)\right)^T$$
and write the objective function as
$$J(\mathbf{f}) = \sum_{i=1}^{n} \ell\left(y_i, \mathbf{f}_i\right).$$

Functional Gradient Descent: Unconstrained Step Direction

- Consider gradient descent on

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

- The **negative gradient step direction** at $\mathbf{f}$ is

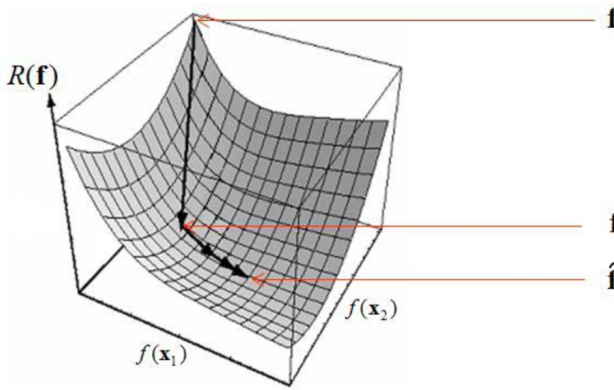
$$\begin{aligned} -\mathbf{g} &= -\nabla_{\mathbf{f}} J(\mathbf{f}) \\ &= -(\partial_{\mathbf{f}_1} \ell(y_1, \mathbf{f}_1), \ldots, \partial_{\mathbf{f}_n} \ell(y_n, \mathbf{f}_n)) \end{aligned}$$

  which we can easily calculate.

- $-\mathbf{g} \in \mathbf{R}^n$ is the direction we want to change each of our n predictions on training data.

- Eventually we need more than just $\mathbf{f}$, which is just predictions on training.

Unconstrained Functional Gradient Stepping



$\mathbf{f}_0$: initial guess

$\mathbf{f}_1 = \mathbf{f}_0 - \rho_1 \nabla R(\mathbf{f})$

$\hat{\mathbf{f}}_M = \sum_{m=0}^{M} \mathbf{f}_m$

$R(\mathbf{f})$ is the empirical risk, where $\mathbf{f} = (f(x_1), f(x_2))$ are predictions on training set.
Issue: $\hat{\mathbf{f}}_M$ only defined at training points.

From Seni and Elder's *Ensemble Methods in Data Mining*, Fig B.1.

Functional Gradient Descent: Projection Step

- Unconstrained step direction is

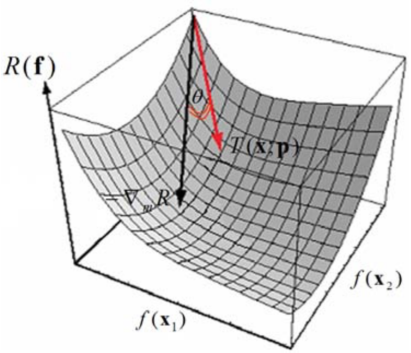
$$-\mathbf{g} = -\nabla_{\mathbf{f}} J(\mathbf{f}) = -\left(\partial_{\mathbf{f}_1} \ell(y_1, \mathbf{f}_1), \ldots, \partial_{\mathbf{f}_n} \ell(y_n, \mathbf{f}_n)\right).$$

- Also called the "**pseudo-residuals**"
  - (for square loss, they're exactly the residuals)
- Find the closest base hypothesis $h \in \mathcal{H}$ (in the ℓ^2 sense):

$$\min_{h \in \mathcal{H}} \sum_{i=1}^{n} \left(-\mathbf{g}_i - h(x_i)\right)^2.$$

- This is a least squares regression problem over hypothesis space $\mathcal{H}$.
- Take the $h \in \mathcal{H}$ that best approximates $-\mathbf{g}$ as our step direction.

"Projected" Functional Gradient Stepping



$T(x; p) \in \mathcal{H}$ is our actual step direction – like the projection of -$\mathbf{g}$=-$\nabla R(\mathbf{f})$ onto $\mathcal{H}$.

From Seni and Elder's *Ensemble Methods in Data Mining*, Fig B.2.

Functional Gradient Descent: Step Size

- Finally, we choose a stepsize.
- Option 1 (Line search):

$$\nu_m = \arg\min_{\nu > 0} \sum_{i=1}^{n} \ell\{y_i, f_{m-1}(x_i) + \nu h_m(x_i)\}.$$

- Option 2: (Shrinkage parameter – **more common**)
  - We consider $\nu = 1$ to be the full gradient step.
  - Choose a fixed $\nu \in (0, 1)$ – called a **shrinkage parameter.**
  - A value of $\nu = 0.1$ is typical – optimize as a hyperparameter .

The Gradient Boosting Machine Ingredients (Recap)

- Take any [sub]differentiable loss function.
- Choose a base hypothesis space for regression.
- Choose number of steps (or a stopping criterion).
- Choose step size methodology.
- Then you're good to go!

Example: BinomialBoost

BinomialBoost: Gradient Boosting with Logistic Loss

- Recall the logistic loss for classification, with $\mathcal{Y} = \{-1, 1\}$:

$$\ell(y, f(x)) \quad = \quad \log\left(1 + e^{-yf(x)}\right)$$

- Pseudoresidual for i'th example is negative derivative of loss w.r.t. prediction:

$$\begin{aligned} r_i &= -\partial_{f(x_i)}\left[\log\left(1 + e^{-y_i f(x_i)}\right)\right] \\ &= \frac{y_i e^{-y_i f(x_i)}}{1 + e^{-y_i f(x_i)}} \\ &= \frac{y_i}{1 + e^{y_i f(x_i)}} \end{aligned}$$

BinomialBoost: Gradient Boosting with Logistic Loss

- Pseudoresidual for ith example:

$$r_i \quad = \quad -\partial_{f(x_i)} \left[\log \left(1 + e^{-y_i f(x_i)} \right) \right] = \frac{y_i}{1 + e^{y_i f(x_i)}}$$

- So if $f_{m-1}(x)$ is prediction after $m-1$ rounds, step direction for m'th round is

$$h_m = \arg\min_{h \in \mathcal{H}} \sum_{i=1}^{n} \left[\left(\frac{y_i}{1 + e^{y_i f_{m-1}(x_i)}} \right) - h(x_i) \right]^2.$$

- And $f_m(x) = f_{m-1}(x) + \nu h_m(x)$.

Gradient Tree Boosting

Gradient Tree Boosting

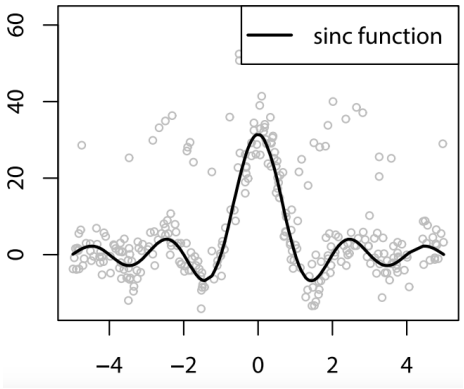
- One common form of gradient boosting machine takes

$$\mathcal{H} = \{\text{regression trees of size } J\},$$

  where J is the number of terminal nodes.
- $J = 2$ gives decision stumps
- HTF recommends $4 \leqslant J \leqslant 8$ (but more recent results use much larger trees)
- Software packages:
  - Gradient tree boosting is implemented by the **gbm package** for R
  - as `GradientBoostingClassifier` and `GradientBoostingRegressor` in **sklearn**
  - **xgboost** and **lightGBM** are state of the art for speed and performance

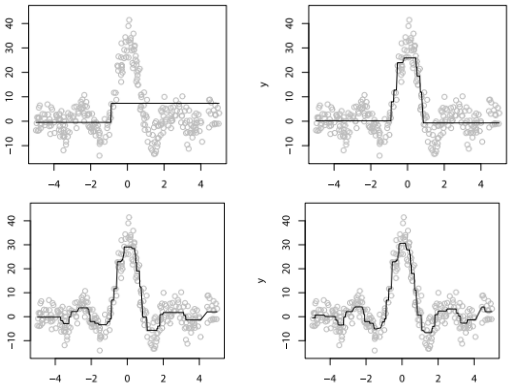
GBM Regression with Stumps

Sinc Function: Our Dataset



From Natekin and Knoll's "Gradient boosting machines, a tutorial"

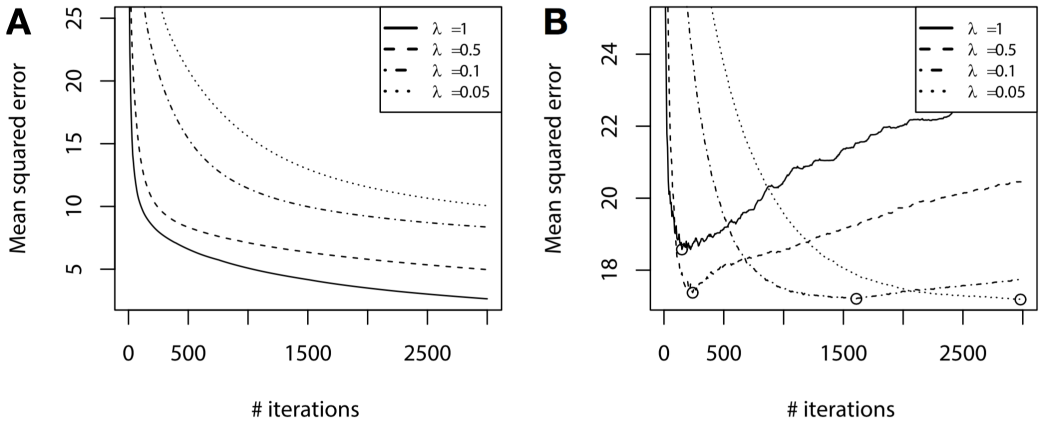
Minimizing Square Loss with Ensemble of Decision Stumps



Decision stumps with $1, 10, 50$, and 100 steps, step size $\lambda = 1$.

Figure 3 from Natekin and Knoll's "Gradient boosting machines, a tutorial"

Step Size as Regularization



Performance vs rounds of boosting and step size. (Left is training set, right is validation set)

Figure 5 from Natekin and Knoll's "Gradient boosting machines, a tutorial"

Rule of Thumb

- The smaller the step size, the more steps you'll need.
- But never seems to make results worse, and often better.
- So set your step size as small as you have patience for.

Variations on Gradient Boosting

Stochastic Gradient Boosting

- For each stage,
  - choose random subset of data for computing projected gradient step.
  - "Typically, about 50% of the dataset size, can be much smaller for large training set."
  - Fraction is called the **bag fraction**.
- Why do this?
  - Subsample percentage is additional regularization parameter – may help overfitting.
  - Faster.
- We can view this is a **minibatch method**.
  - we're estimating the "true" step direction (the projected gradient) using a subset of data

Introduced by Friedman (1999) in Stochastic Gradient Boosting.

Bag as Minibatch

- Just as we argued for minibatch SGD,
  - sample size needed for a good estimate of step direction is independent of training set size
- Minibatch size should depend on
  - the complexity of base hypothesis space
  - the complexity of the target function (Bayes decision function)
- Seems like an interesting area for both practical and theoretical pursuit.

Column / Feature Subsampling for Regularization

- Similar to random forest, randomly choose a subset of features for each round.
- XGBoost paper says: "According to user feedback, using column sub-sampling prevents overfitting even more so than the traditional row sub-sampling."
- Zhao Xing (top Kaggle competitor) finds optimal percentage to be 20%-100%

Newton Step Direction

- For GBM, we find the closest $h \in \mathcal{F}$ to the negative gradient

$$-\mathbf{g} = -\nabla_{\boldsymbol{f}} J(\mathbf{f}).$$

- This is a "first order" method.

- Newton's method is a "second order method":
  - Find 2nd order (quadratic) approximation to J at $\boldsymbol{f}$.
    - Requires computing gradient and Hessian of J.
  - Newton step direction points towards minimizer of the quadratic.
  - Minimizer of quadratic is easy to find in closed form
- Boosting methods with projected Newton step direction:
  - LogitBoost (logistic loss function)
  - XGBoost (any loss – uses regression trees for base classifier)

Newton Step Direction for GBM

- Generically, second order Taylor expansion of J at $\mathbf{f}$ in direction $\mathbf{r}$

$$J(\mathbf{f}+\mathbf{r}) = J(\mathbf{f}) + [\nabla_{\mathbf{f}} J(\mathbf{f})]^T \mathbf{r} + \frac{1}{2}\mathbf{r}^T \left[\nabla^2_{\mathbf{f}} J(\mathbf{f})\right] \mathbf{r}$$

- For $J(\mathbf{f}) = \sum_{i=1}^n \ell(y_i, \mathbf{f}_i)$,

$$J(\mathbf{f}+\mathbf{r}) = \sum_{i=1}^n \left[\ell(y_i, \mathbf{f}_i) + g_i \mathbf{r}_i + \frac{1}{2} h_i \mathbf{r}_i^2\right],$$

  where $g_i = \partial_{\mathbf{f}_i} \ell(y_i, \mathbf{f}_i)$ and $h_i = \partial^2_{\mathbf{f}_i} \ell(y_i, \mathbf{f}_i)$.
- Can find $\mathbf{r}$ that minimizes $J(\mathbf{f}+\mathbf{r})$ in closed form.
- Can take step direction to be "projection" of $\mathbf{r}$ into base hypothesis space $\mathcal{H}$.

XGBoost: Objective Function with Tree Penalty Term

- Adds explicit penalty term on tree complexity to the empirical risk:

$$\Omega(r) = \gamma T + \frac{1}{2}\lambda \sum_{i=1}^{T} w_j^2,$$

  where $r \in \mathcal{H}$ is a regression tree from our base hypothesis space and
  - T is the number of leaf nodes and
  - w_j is the prediction in the j'th node
- Objective function at step m:

$$J(r) = \sum_{i=1}^{n} \left[g_i r(x_i) + \frac{1}{2} h_i r(x_i)^2 \right] + \Omega(r)$$

- In XGBoost, they also use this objective to decide on tree splits
- See XGBoost Introduction for a nice introduction.

XGBoost: Rewriting objective function

- For a given tree, let $q(x_i)$ be x_i's node assignment and w_j the prediction for node j.
- In each step of XGBoost we're looking for a tree that minimizes

$$\sum_{i=1}^{n}\left[g_i w_{q(x_i)} + \frac{1}{2}h_i w_{q(x_i)}^2\right] + \gamma T + \frac{1}{2}\lambda\sum_{i=1}^{T} w_j^2$$

$$= \sum_{\text{leaf node } j=1}^{T}\left[\left(\underbrace{\sum_{i\in I_j} g_i}_{G_j}\right) w_j + \frac{1}{2}\left(\underbrace{\sum_{i\in I_j} h_i}_{H_j} + \lambda\right) w_j^2\right] + \gamma T,$$

where $I_j = \{i \mid q(x_i) = j\}$ is set of training example indices landing in leaf j.

XGBoost: Simple Expression for Tree Penalty/Loss

- Simplifies to

$$\sum_{j=1}^{T}\left[G_j w_j + \frac{1}{2}\left(H_j + \lambda\right) w_j^2\right] + \gamma T$$

- For fixed $q(x)$ (i.e. fixed tree partitioning), objective minimized when leaf node values are

$$w_j^* = -G_j / \left(H_j + \lambda\right).$$

- Plugging w_j^* back in, this objective reduces to

$$-\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T,$$

  which we can think of as the loss for tree partitioning function $q(x)$.

- If time were no issue, we could search over all trees to mininize this objective.

XGBoost: Building Tree Using Objective Function

- Expression to evaluate a tree's node assignment function $q(x)$:

$$-\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda}+\gamma T,$$

  where $G_j=\sum_{i\in I_j} g_i$ for examples i assigned to leaf node j. And $H_j=\sum_{i\in I_j} h_i$.

- Suppose we're considering splitting some data into two nodes: L and R.
- Loss of tree with this one split is

$$-\frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}\right]+2\gamma.$$

- Without the split – i.e. a tree with a single leaf node, loss is

$$-\frac{1}{2}\left[\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right]+\gamma.$$

XGBoost: Node Splitting Criterion

- We can define the **gain** of a split to be the reduction in objective between tree with and without split:

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right]-\gamma.$$

- Tree building method:
  - recursively choose split that maximizes the gain.