Boosting

Matthew Drury

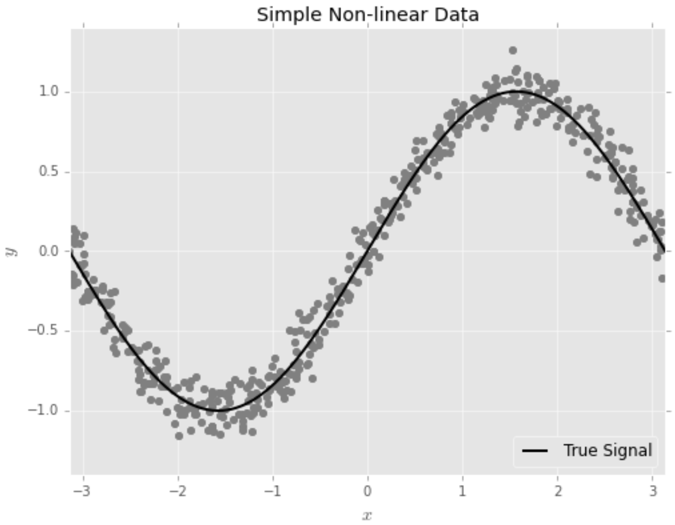
August 14, 2016

Introduction to Boosting

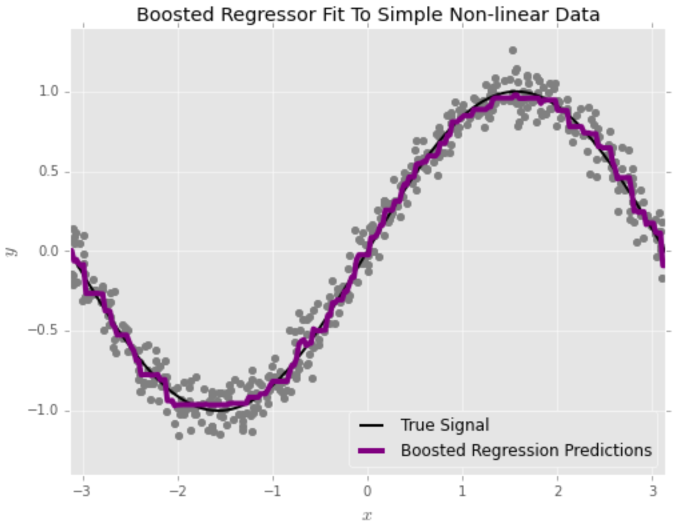
Boosting encompasses a highly successful set of learning algorithms.

- Allstate has held three Kaggle competitions. All three were won by algorithms incorporating gradient boosting as a fundamental component.
- In the 1990's the insurance industry discovered that incorporating consumers credit information into pricing greatly increased the accuracy of prices, this revolutionized the industry. Using a boosted model in place of a linear model when setting prices gives roughly the same increase in power.

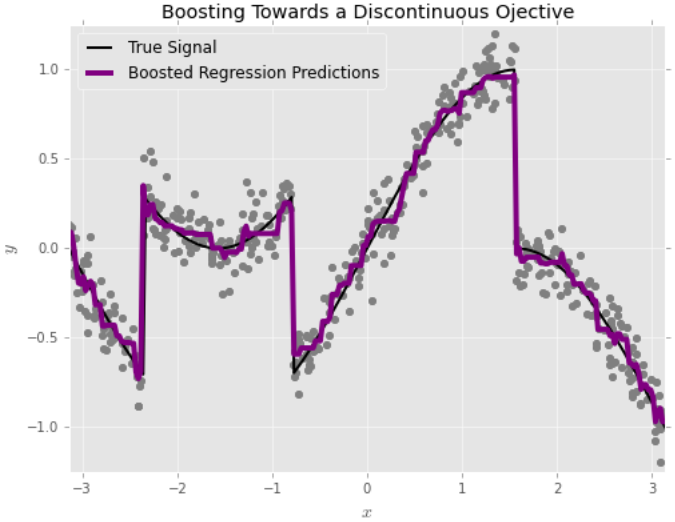
Boosting can adapt itself effortlessly to very non-linear objectives



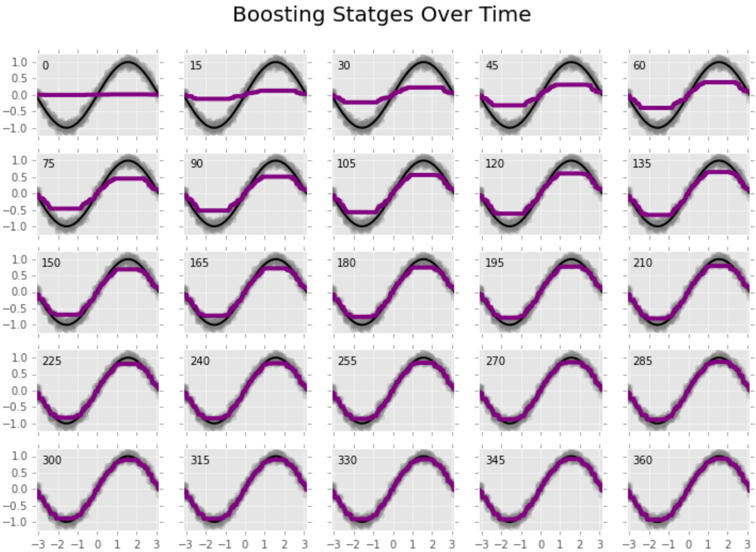
Boosting can adapt itself effortlessly to very non-linear objectives



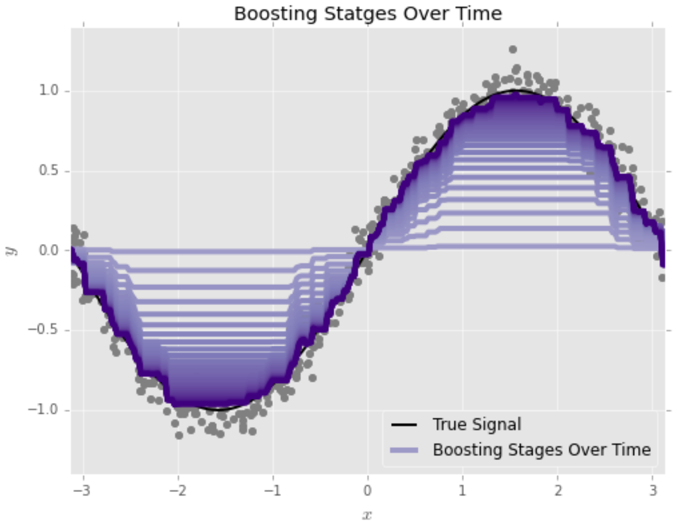
Boosting can adapt itself effortlessly to very non-linear objectives



Boosting accomplishes this by *growing the model gradually*



At each stage of the growth, the next model is built as an adjustment to the previous model



Outline of the Lesson

Agenda:

- Introduction
- You Could Have Invented Gradient Boosting
- Practical Gradient Boosted Regression
- Practical Gradient Boosted Classification
- Adaboost
- Drawbacks of Boosting

Objectives:

- Understand the conceptual foundation of Boosting
- Understand the algorithms hyperparameters, and how to tune them.
- Understand how to create a booster for your own loss function.
- Understand some basic strategies for interpreting a booster.
- Understand the drawbacks of boosting.

You Could Have Invented Gradient Boosting

Let's start with our basic setup.

$\{x_i, y_i\}$ is a data set, where i indexes the samples we have available for training our model.

Each x_i may be a vector, in which case I'll refer to it's components (if needed) as x_{ij}.

Our goal is to construct a function f so that, approximately

$$y_i \approx f(x_i) \text{ for all } i$$

Question: What should the domain of f be?

Degenerate Choice: $\text{Domain}(f) = \{x_i\}$.

That is, let's only attempt to define f on our training sample.

"But Matt. This is silly. The answer is obvious."

Define: $f(x_i) = y_i$

True.

But let's try to derive this in a creative way.

Recall: Gradient descent is a general purpose algorithm for optimizing any objective function $L(x)$.
Algorithm: Gradient Descent to Minimize a function L.

- Compute $\nabla L(x)$ somehow, on paper is good.
- Initialize $x_0 = 0$ (for example, there may be more principled choices).
- Until satisfied, iterate:
  - Set $x_{i+1} = x_i - \nabla L(x_i)$.

Let's focus on a single point in our domain, and stick with the classic squared error loss function

$$L(f, y) = \frac{1}{2}(y - f)^2$$

Here f is not a function yet, it is just a number.

Following the gradient descent recipe for optimizing this loss function, let's initialize f to the average value of y_i

$$f_0 = \frac{1}{N} \sum_i y_i$$

And compute the gradient with respect to f by hand

$$\nabla_f(f, y) = \frac{\partial}{\partial f}\frac{1}{2}(y - f)^2 = f - y$$

...and apply the update rule

$$f_1 = f_0 - \nabla_f(f_0, y) = f_0 - (f_0 - y) = y$$

So this (admittedly quite bizarre) application of gradient descent immediately recovers the correct solution

$$f(x_i) = y_i$$

for every data point.

What is stopping up from applying this scheme in the more realistic situation where we want to construct a function f with domain $\mathbb{R}^n$ so that

$$f(x_i) \approx y_i$$

The **first step works**, we can certainly define f_0 to be the constant function

$$f(x) = \frac{1}{N} \sum_i y_i \text{ for all } x \in \mathbb{R}^n$$

The **update step fails**, we cannot evaluate the gradient at any point where we have not observed a value of y.

$$\nabla_f(f, y) = f - y$$

Solution: Fit a simple model to the new dataset

$$\{x_i, \nabla_f L(f_0(x_i), y_i)\} = \{x_i, f_0(x_i) - y_i\}$$

The **predictions from this model** can be viewed as an extension of the gradient to all of $\mathbb{R}^n$.

Algorithm: Gradient Boosting to Minimize Sum of Squared Errors.

Inputs: A data set $\{x_i, y_i\}$.
Returns: A function f such that $f(x_i) \approx y_i$.

- Initialize $f_0(x) = \frac{1}{N} \sum_i y_i$.
- Iterate (parameter k) until satisfied:
  - Create the working data set $W_k = \{x_i, f_k(x_i) - y_i\}$.
  - Fit a decision tree to W_k, minimizing least squares (though most anything would work here). Call this tree T_k.
  - Set $f_{k+1}(x) = f_k(x) - T_k(x)$.
- Return $f_{\max}(x) = f_0(x) - T_1(x) - T_2(X) - \cdots - T_{\max}(x)$.

Comments:

- We didn't *have* to use decision trees, literally anything would work.
- Just like in other algorithms, we can introduce a *learning rate* to make the gradient descent more robust

$$x_{i+1} = x_i - \lambda \nabla L(x_i)$$

  This is particularly important in boosting, to prevent overfitting.
- We could have fit the tree to the negative gradient, which would have resulted in the more aesthetically appealing

$$f_{\max}(x) = f_0(x) + T_1(x) + T_2(X) + \cdots + T_{\max}(x)$$