

# Boosting

Slowly Learning from a Lot of  
Smart Mistakes

Mark Llorente, Much of the  
Content from Various  
Instructors



Morning: Intro to Boosting

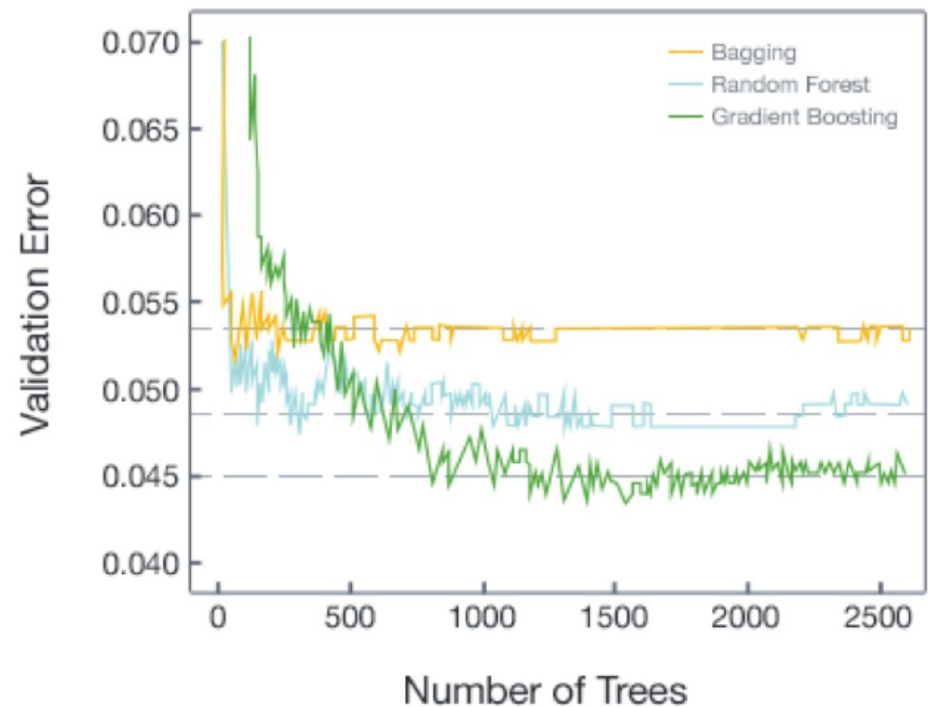
Afternoon: Adaboost vs  
Gradient Boost, Walking  
through the Adaboost  
Algorithm

# Morning Objectives

- Contrast Boosting with Random Forests
  - Bias vs Variance, Again!
- Be Familiar With the Principles of Boosting
  - Variations on Boosting: Gradient Boosting, Adaptive Boosting (Adaboost), XGboost
    - Why is it called Gradient Boosting?
  - Why do we use “shallow trees?”
- Know How to Tune Hyperparameters
  - Learning Rate, Number of Trees, Subsampling
    - Depth of Trees (Keep Them Simple!), and all other tree pruning hyperparams

# First Off: Why Boosting?

- Squeeze out the most predictive power from models
  - Boosting can be applied to non-tree models but is most often used with trees
- Resistant to overfitting
- “Smoother” predictions than RF
  - Ensemble of high bias/low-variance predictors
- You prefer to learn from previous mistakes rather than rely on the wisdom of the crowd
  - You absolute rebel~



# What *Isn't* Boosting?

## TOP DEFINITION

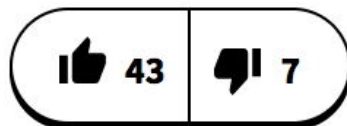
### Boosting

To steal retail items and resell them on the black market.

*Women/Men selling knockoff purses in the street are boosting.*

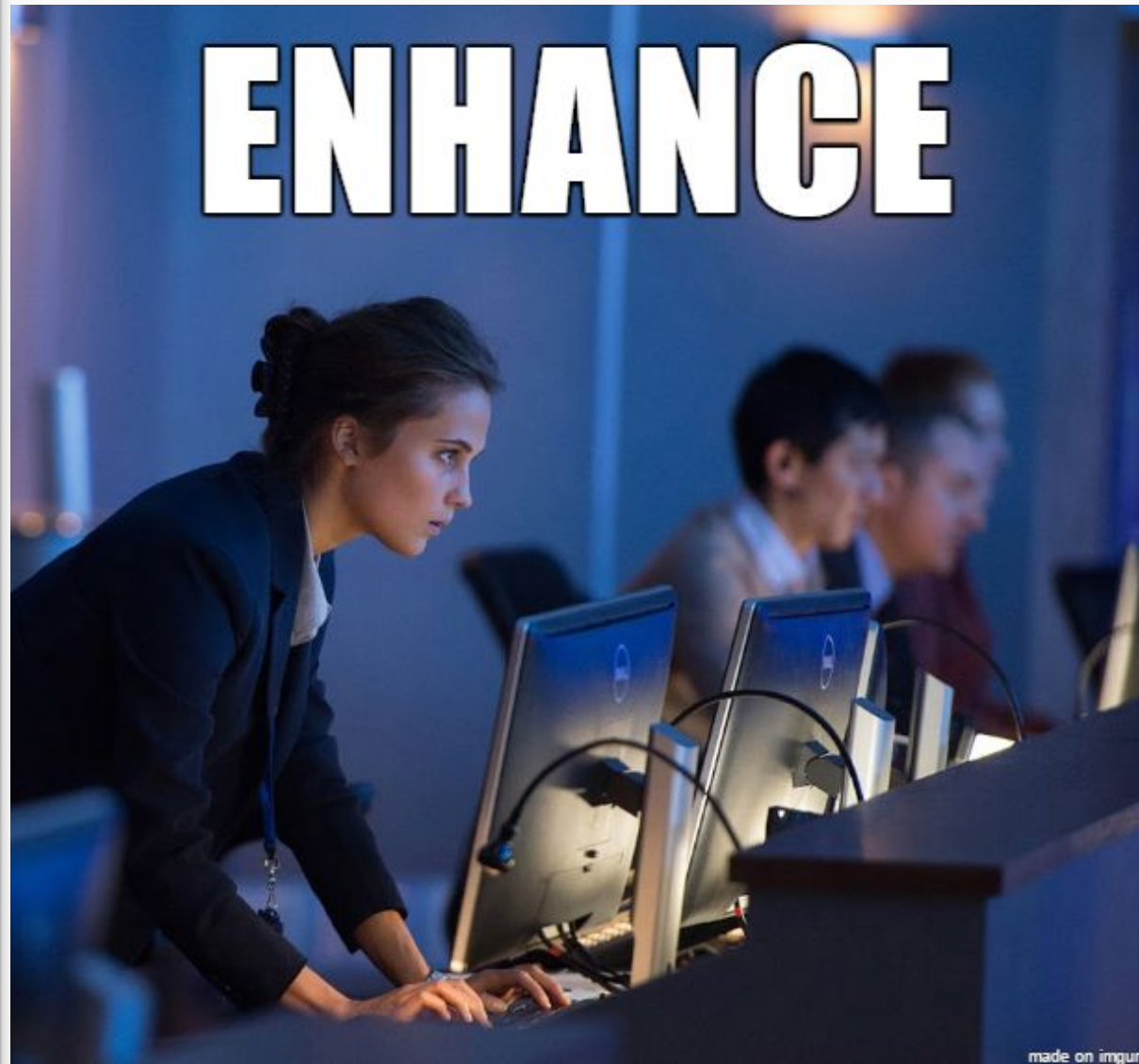
#thief #steal #stolen #kleptomaniac #knockoff

by **Goe** September 13, 2015



What /s Boosting?

More like this

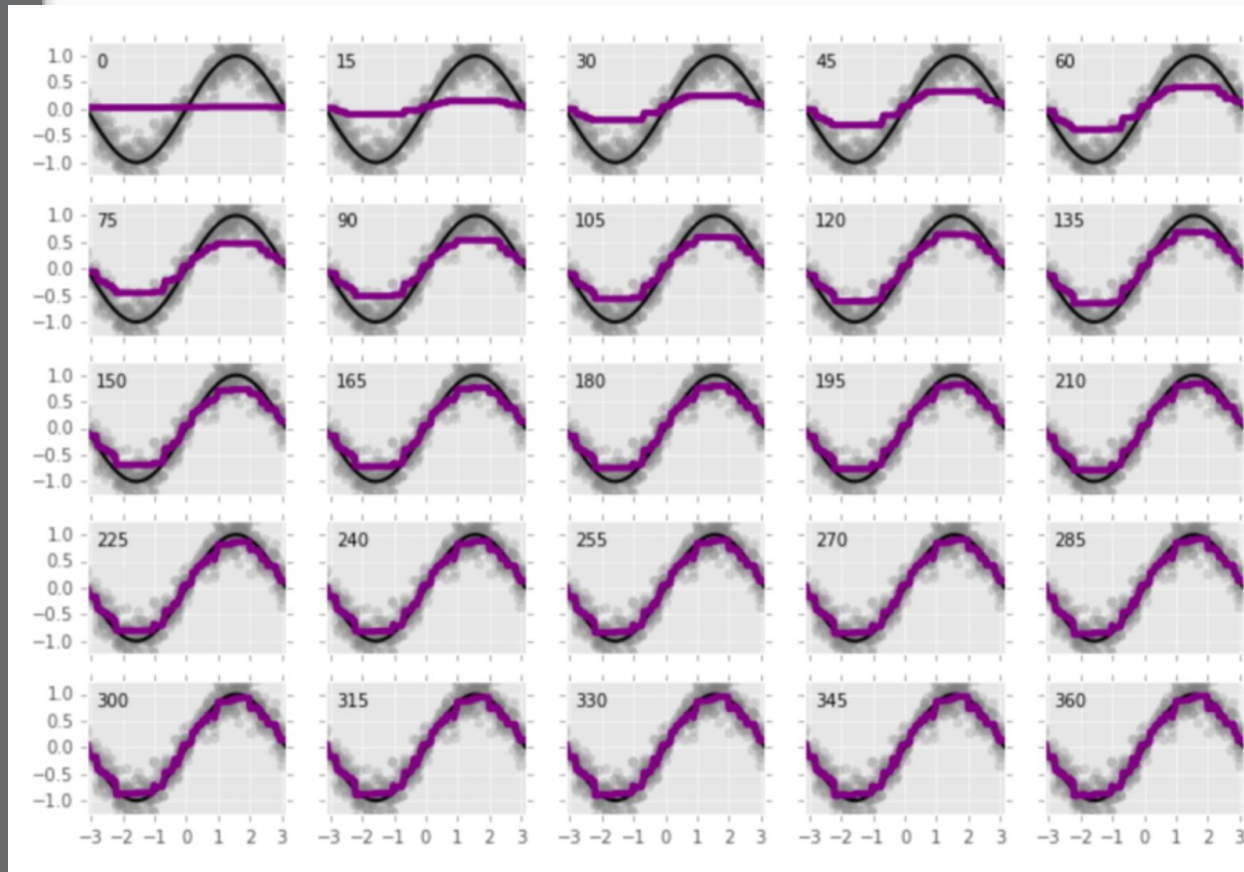


galvanize

## What Is Boosting?

Many weak approximations added in stages, each stage focusing on what was missed by all previous stages.

“Learning off residuals”

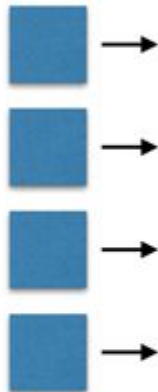


# What Is Boosting?

**Random Forest** (a lot of clever trees)  
Can be trained in parallel (multi-core).

**Boosting** (a lot of not clever trees)  
*Cannot* be trained in parallel!

Bagging/RF



$\text{pred} = \text{consensus}(f_1(x), f_2(x), f_3(x), f_4(x))$

\*consensus() is via majority vote, mean, etc

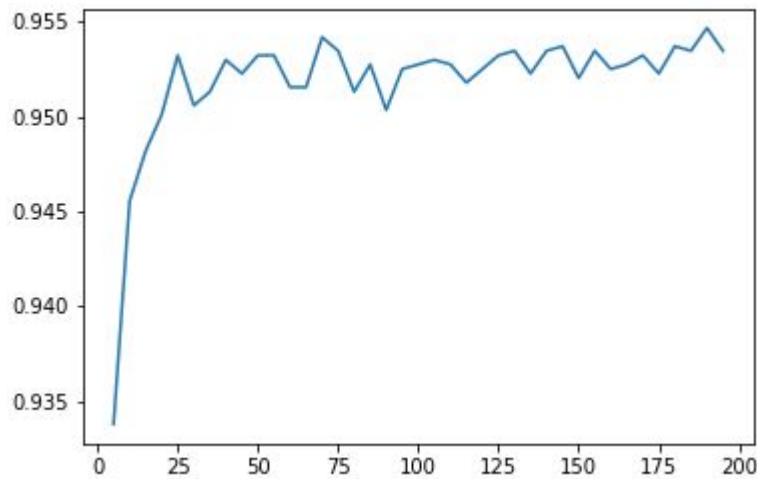
Boosting



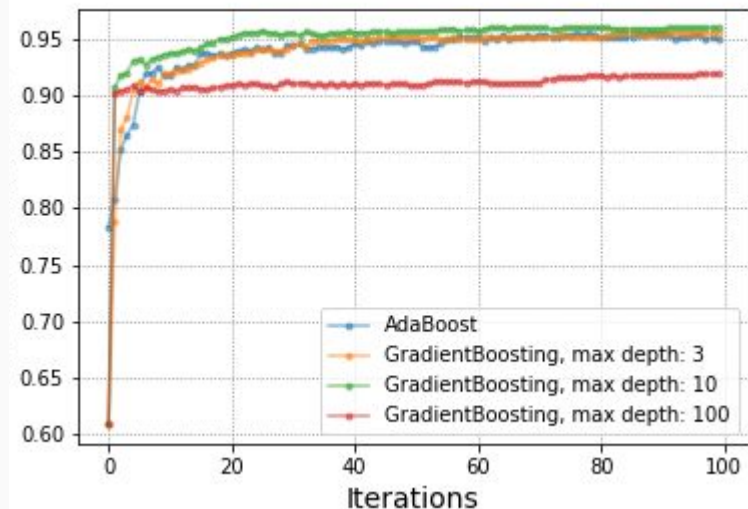
$\text{pred} = f_1(x) + f_2(x) + f_3(x) + f_4(x)$

# RF and GB Classification Scores with Increasing Number of Trees

## Random Forest Test Accuracy

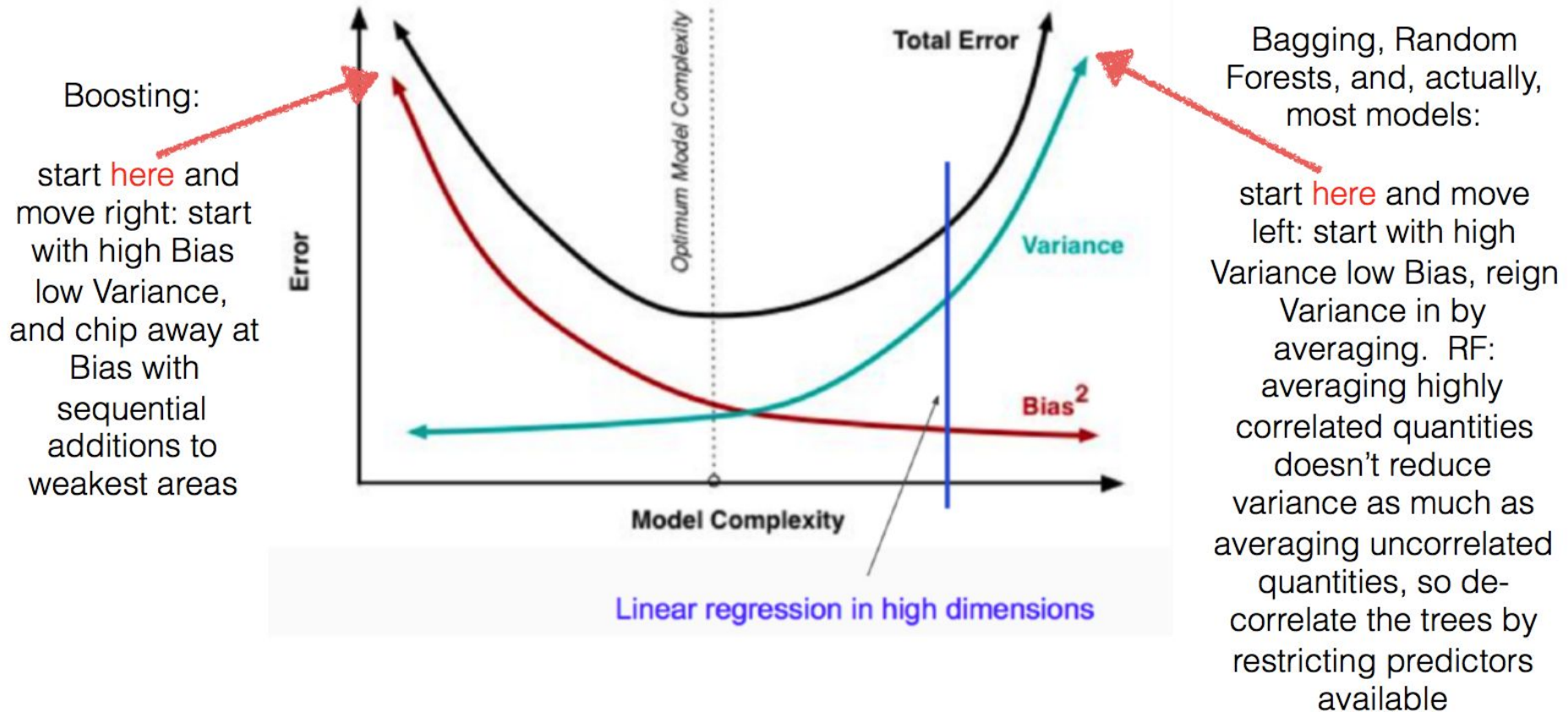


## Gradient Boost Test Accuracy





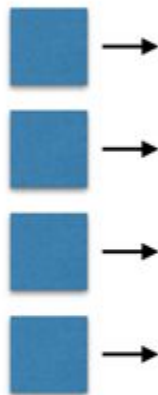
# RF vs GB “Starting Points”



# Random Forest vs Boosting

Random Forest (and Bagging) trees can be trained in parallel. Boosted trees must be trained in series.

Bagging/RF



$\text{pred} = \text{consensus}(f_1(x), f_2(x), f_3(x), f_4(x))$

\*consensus() is via majority vote, mean, etc

Boosting



$\text{pred} = f_1(x) + f_2(x) + f_3(x) + f_4(x)$

In the end,  $f$  will be a sum of smaller (often called *weak*) learners

$$f(x) = f_0(x) + f_1(x) + f_2(x) + \cdots + f_{\max}(x)$$

The process of building up the model looks like

$$S_0(x) = f_0(x)$$

$$S_1(x) = f_0(x) + f_1(x)$$

$$S_2(x) = f_0(x) + f_1(x) + f_2(x)$$

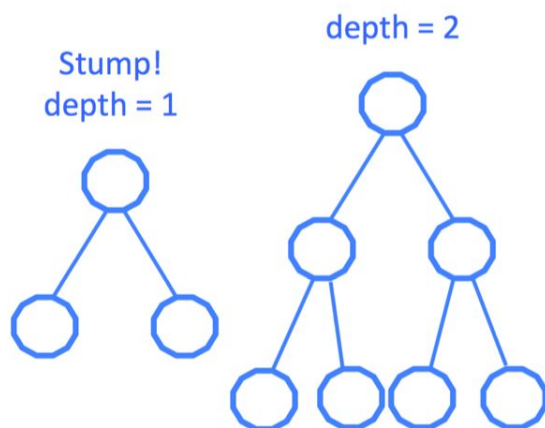
$$\vdots$$

$$S_{\max}(x) = f_0(x) + f_1(x) + f_2(x) + \cdots + f_{\max}(x)$$

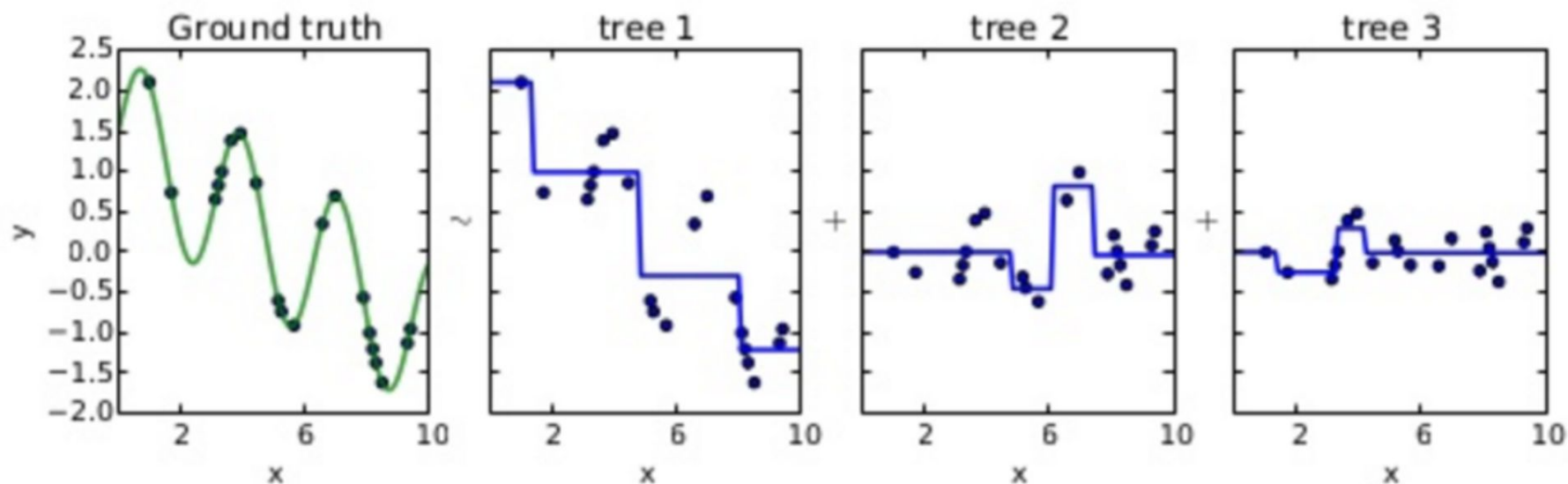
To the Poll(s)!

galvanize

# Fitting regressions on sequential residuals

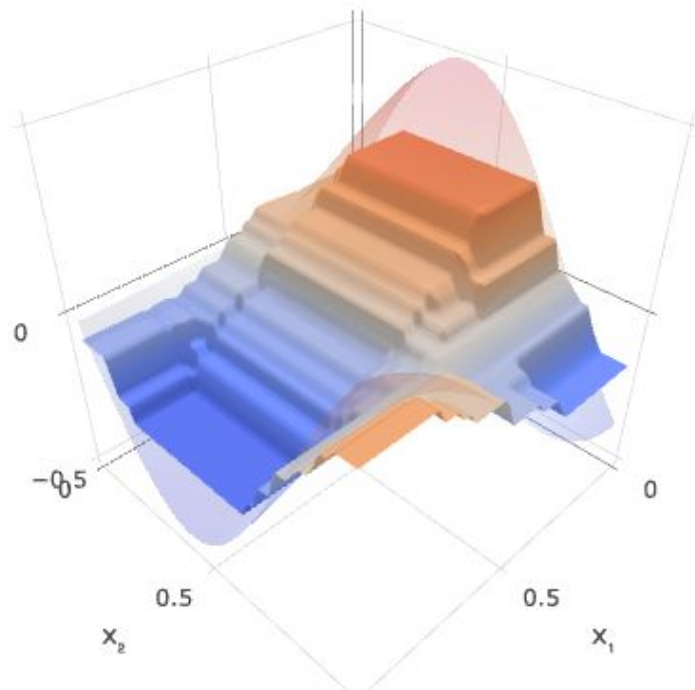


1. Set  $\hat{f}^{(0)}(\mathbf{x}_i) = 0$  and  $\hat{\epsilon}_i^{(1)} = Y_i$
2. For  $k = 1, \dots, m$ 
  - 2.1 Fit a tree  $\hat{f}^{(k)}$  to  $\hat{\epsilon}^{(k)}$  using features  $\mathbf{x}$
  - 2.2 Update the estimator  $\hat{f}^{(k+1)} = \hat{f}^{(k)} + \alpha_k \hat{f}^{(k)}$
  - 2.3 Update the residuals  $\hat{\epsilon}_i^{(k+1)} = \hat{\epsilon}_i^{(k)} - \alpha_k \hat{f}^{(k)}(\mathbf{x}_i)$
3. Return the boosted model  $\hat{f}(\mathbf{x}_i) = \sum_{k=1}^m \alpha_k \hat{f}^{(k)}(\mathbf{x}_i)$

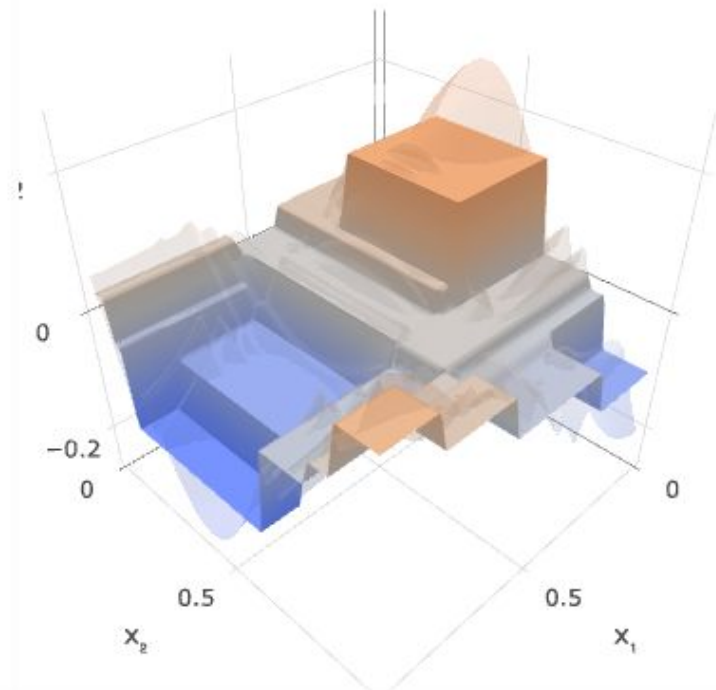


[https://arogozhnikov.github.io/2016/06/24/gradient\\_boosting\\_explained.html](https://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html)

target function  $f(\mathbf{x})$  and prediction of previous trees  $D(\mathbf{x})$



residual  $R(\mathbf{x})$  and prediction of next tree  $d_n(\mathbf{x})$



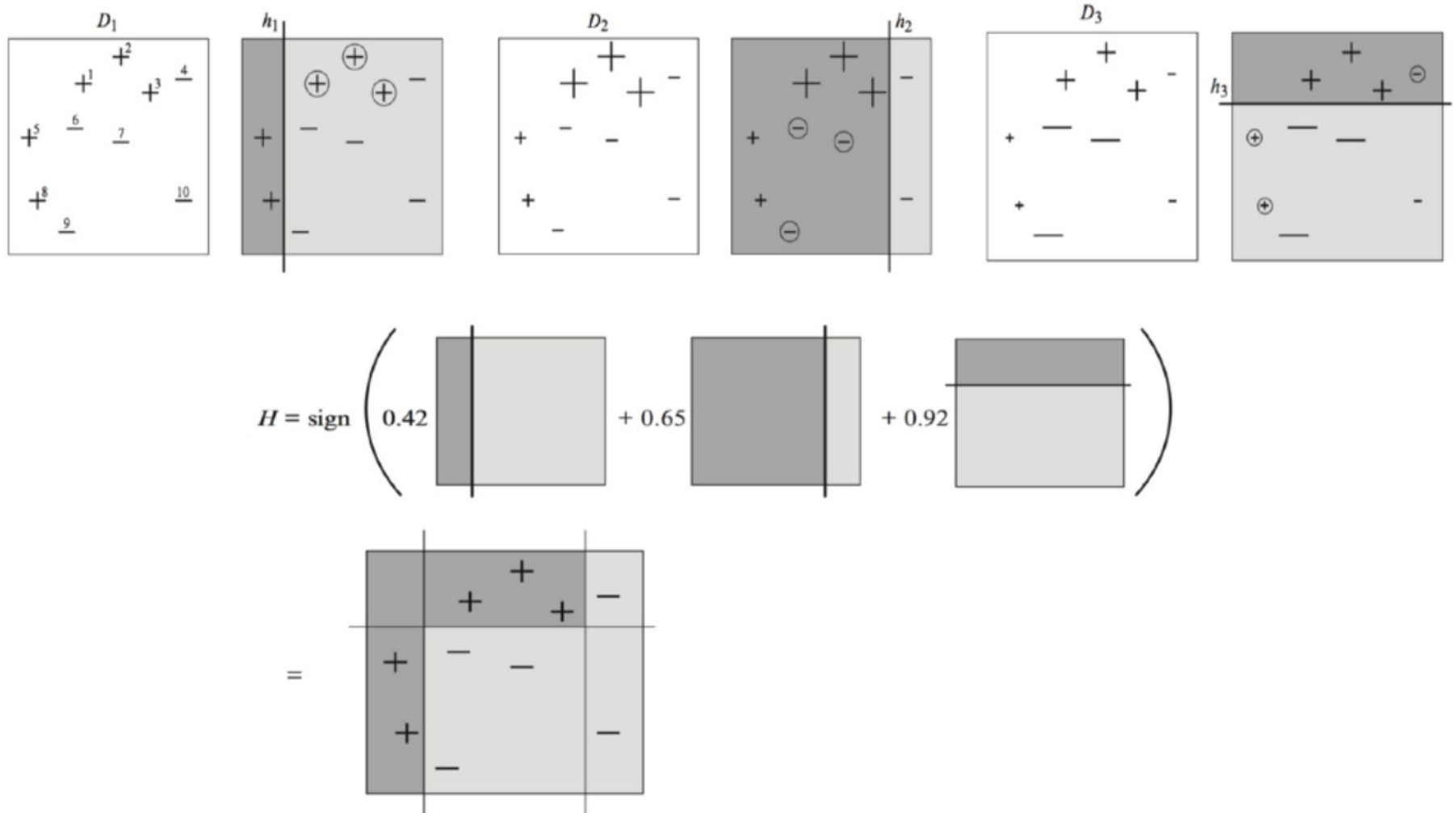
Tree depth: 4



Number of built trees: 3



# Classification with “Stumps” using Adaboost






# • Boosting Algorithm

---

**Algorithm 8.2** *Boosting for Regression Trees*

---

1. Set  $\hat{f}(x) = 0$  and  $r_i = y_i$  for all  $i$  in the training set.
2. For  $b = 1, 2, \dots, B$ , repeat:
  - (a) Fit a tree  $\hat{f}^b$  with  $d$  splits ( $d + 1$  terminal nodes) to the training data  $(X, r)$ .  The (view of the) 'training data' changes below
  - (b) Update  $\hat{f}$  by adding in a shrunk version of the new tree:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \quad (8.10)$$

- (c) Update the residuals,

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \quad \text{←} \quad (8.11)$$

3. Output the boosted model,

$$\hat{f}(x) = \sum_{b=1}^B \lambda \hat{f}^b(x). \quad (8.12)$$

---



## Explanation Adapted from Dan Wiesen

- Start with a model predicting only 0, whether for classification or regression.
  - That means no trees yet!  $f(x) = 0$
- Set our first residuals to be our original  $y$  values:
  - Why? If we predicted all zeros, we get  $y$  back.  $y - f(x) = y - 0 = y$
  - We still haven't updated  $f(x)$ .
- For each of the  $B$  trees we want to train:
  - Fit a *new* tree,  $f^b(x)$ , to the *current* residuals
    - Residuals =  $y$  only at the beginning. Residuals should continue to decrease as we fit more trees to them!
    - Note: We limit each tree's depth so we don't overfit
  - Combine previous  $f(x)$  with  $f^b(x)$  to update it but with a small weight applied to  $f^b$  so our "step size" isn't too big:
    - $f(x) += \lambda f^b(x) = 0 + \lambda f^1(x) + \lambda f^2(x) + \dots + \lambda f^b(x)$
    - Update the residuals by using the new 'ensemble' **and repeat!**
      - $r_{\text{current}} = y - f_{\text{current}}(x)$     or     $r -= \lambda f^b(x)$
- Return  $f(x)$ , the weighted sum of all  $B$  trees

# Common Types of Boosting

- Adaptive Boosting, Adaboosting
  - First boosting algorithm circa 1997
  - Binary Classification Only
  - *Adaptively* updates new tree weights based on size of total errors were AND *Adaptively* updates data point weights to focus on where errors were made
  - *Not* the strongest boosting algorithm but still used
- Gradient Boosting
  - A generalized form of Adaboost that can use different loss functions
  - Called gradient boosting because we're implicitly using gradient descent to update our model with trees that help us minimize our loss!
    - Not just a metaphor. More info at end of presentation.
- XGBoost
  - Regularly used in Kaggle Competitions for Big Data
  - Cleverly only selects data to look at if they represent a percentile for a given feature.
  - Avoids needing to evaluate every data point and adds bonus "regularization"
  - Still gradient boosting

More polls!

galvanize

# Tuning Hyperparameters

Learning rate: Just like step-size in Gradient Descent. The smaller, the smoother the final forest will predict and the more accurate it will be BUT at the cost of needing way more trees.

Number of trees: Use a lot of trees and you can stop when you're happy with the plateau or you start seeing a trend toward overfitting (happens easily with high tree depth or high learning rates)

Depth: Keep it simple! Increasing depth may tease out feature interactions but too deep and you lose the benefit of a clumsy stubby tree. Stumps are more than okay!

Subsample: Somewhere between 0.5-0.9 is fine. Not necessary but may help depending on your dataset.

# Tuning Hyperparameters

In Practice:

Start with relatively high learning rates ( $\sim 0.1$ ) and grid search over other parameters, primarily to find a nice happy tree depth and a good value for subsampling.

Change to a MUCH smaller learning rate, 0.001 for example. Train with so MANY trees. Your computer may need to run overnight to finish.

Wake up to a very powerful model (and hopefully not an error warning sign).

# Pros and Cons

Pros: Powerful and relatively easy to interpret. Only a few hyperparameters, primarily learning rate, which can dramatically improve your model output. Adaboost easy to use right out of the box. Variations of this are used in Kaggle competitions.

Cons: Can't be trained in parallel like Random Forest models. Can eventually cause overfitting if you use too many trees, but you can always choose to get rid of those offending trees.



# Individual Sprint

Compare performances for a few different forest models and explore what happens as you adjust hyperparameters!

# Afternoon Objectives

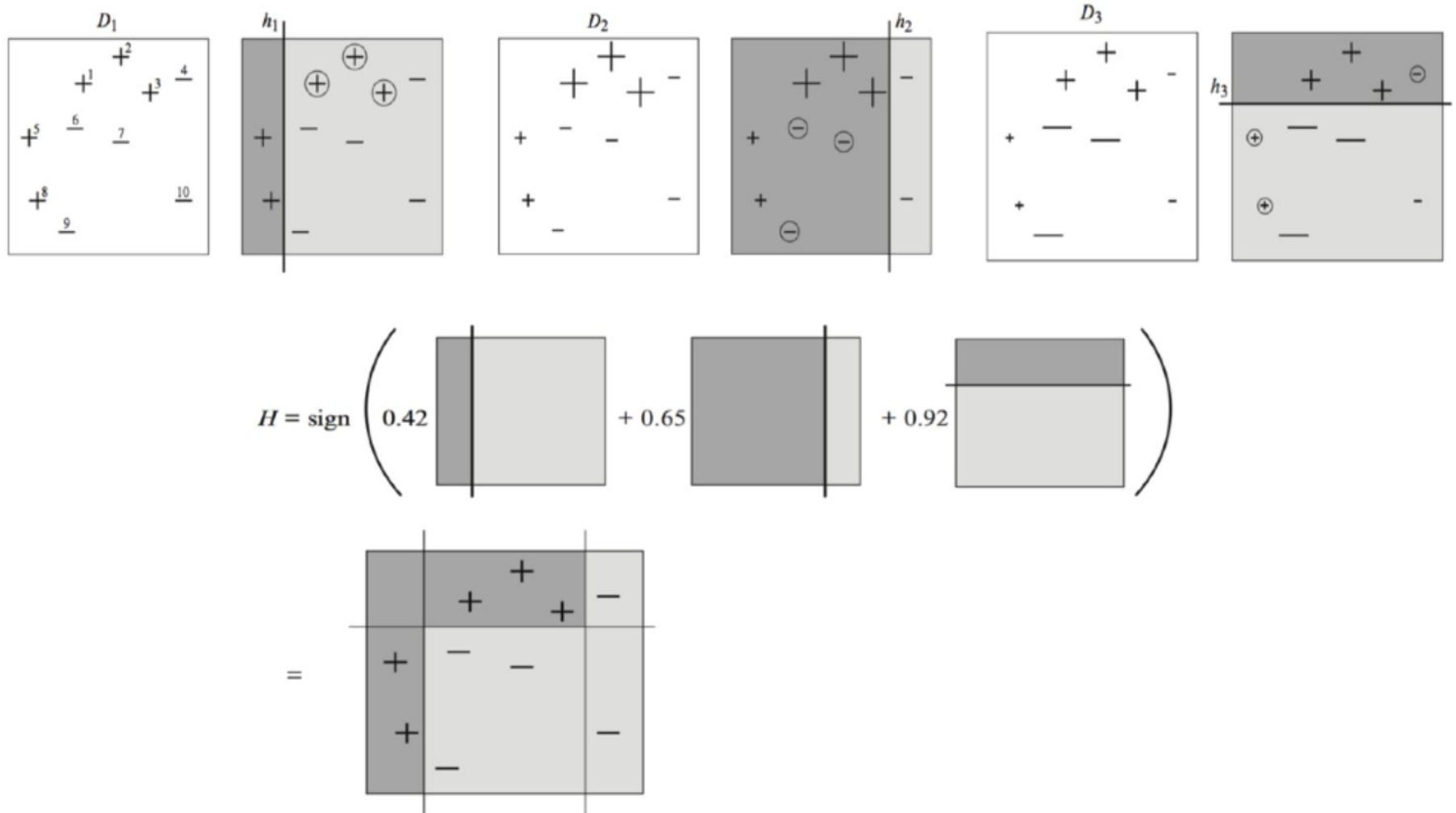
- Interpret our new friend for the sprint, the Adaboost (Adaptive Boosting) algorithm
- That's it. It looks complicated but it's just a few tweaks from a normal classifier problem with some math in the way.



First: More Polls!

galvanize

# Classification with “Stumps” using Adaboost



# AdaBoost algorithm

- for  $k = 1, \dots, m$ :
  - train a weak learner  $g_k(x)$  on  $\{x_i, y_i\}$  using sample weights  $\{w_i\}$
  - calculate tree error  $\epsilon_k = \sum_i^n w_i \mathbf{I}(g_k(x_i) \neq y_i)$ 
    - $\mathbf{I}(\theta)$  is the indicator function:  $\mathbf{I}(True) = 1, \mathbf{I}(False) = 0$
    - note that this is simply adding up all the weights of the incorrectly labeled data points
    - since the weights are normalized (and non-negative),  $\epsilon_k$  can range between 0 and 1
  - set the tree weight  $\alpha_k = \log\left(\frac{1-\epsilon_k}{\epsilon_k}\right)$
  - update sample weights  $w_i \rightarrow w_i \exp(-\alpha_k y_i g_k(x_i))$ 
    - note that  $\exp(-\alpha_k y_i g_k(x_i))$  is  $e^{-\alpha_k}$  when the model is correct and  $e^{\alpha_k}$  when it is wrong
    - note again (for fun), that  $e^{\alpha_k} = \frac{1-\epsilon_k}{\epsilon_k}$
  - normalize sample weights  $w_i \rightarrow \frac{w_i}{\sum_i^n w_i}$
- final model  $G(x) = \text{sign}\left(\sum_k^m \alpha_k g_k(x)\right)$

# Parts of the Algorithm

$g_k(x)$  = Prediction from Decision Tree  $m$

**Important note: This algorithm uses -1 and +1 instead of 0 and +1 for the two classes! Your sklearn decision tree doesn't do that as a default.**

$w_i$  = weight for datapoint  $i$

Start with uniform weight  $i/N$  and update weight values using formula and use in next tree.

You can pass newly updated sample weights to a DecisionTreeClassifier using the `sample_weight` keyword argument of the `.fit()` method.

$I(g_k(x_i) \neq y_i)$  Indicator function. This one indicates when an error is found! Returns 1 when inside is **True** (i.e. when label and prediction *don't* match!) and 0 when inside is **False** (i.e. *correct* prediction).

# Parts of the Algorithm

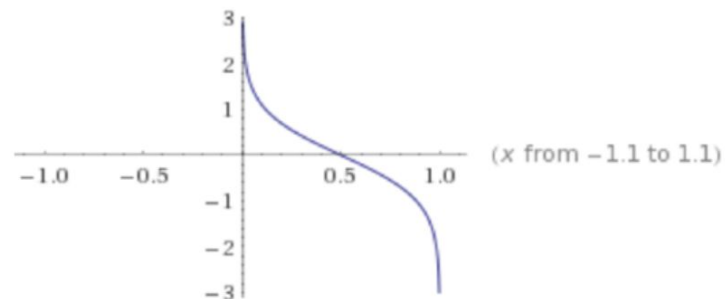
$\alpha_k$  = the tree weight, how much weight we give the *prediction* from a particular tree to our total prediction. Each tree gets one to say how “important” it is to the prediction.

$$\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)$$

Input:

$$0.5 \log \left( \frac{1 - x}{x} \right)$$

Plots:



# Tips for Afternoon Sprint

- We're using sklearn's standard decisions tree classifiers as the base Adaboost classifier. It uses 0 and 1 for the two classes but the algorithm needs -1 and 1 to calculate and update the weights.
  - If  $x = \text{either } 0 \text{ or } 1$ ,  $(2*x-1)$  will give -1 when  $x$  is 0 and +1 when  $x$  is 1.
  - You can also just set
- The final model *also* assumes we're using -1 and 1 to describe the two classes, so you need to do the same conversion for the outputs from each of the individual decision trees you've trained.
- Using data point weights, repeated from previous slide:
  - Start with uniform weight  $1/N$  and update weight values using formula and use in next tree.
  - You can pass newly updated sample weights to a DecisionTreeClassifier using the `sample_weight` keyword argument of the `.fit()` method.

# Additional Info: Loss

## Gradient Boosting

From: A Gentle Introduction to Gradient Boosting, Cheng Li

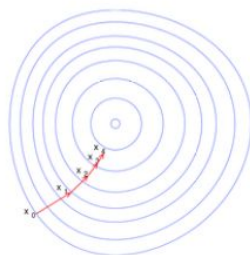
### Gradient Boosting for Regression

### Gradient Boosting for Regression

#### Gradient Descent

Minimize a function by moving in the opposite direction of the gradient.

$$\theta_i := \theta_i - \rho \frac{\partial J}{\partial \theta_i}$$



#### How is this related to gradient descent?

Loss function  $L(y, F(x)) = (y - F(x))^2/2$

We want to minimize  $J = \sum_i L(y_i, F(x_i))$  by adjusting  $F(x_1), F(x_2), \dots, F(x_n)$ .

Notice that  $F(x_1), F(x_2), \dots, F(x_n)$  are just some numbers. We can treat  $F(x_i)$  as parameters and take derivatives

$$\frac{\partial J}{\partial F(x_i)} = \frac{\partial \sum_i L(y_i, F(x_i))}{\partial F(x_i)} = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = F(x_i) - y_i$$

So we can interpret residuals as negative gradients.

$$y_i - F(x_i) = -\frac{\partial J}{\partial F(x_i)}$$

Figure : Gradient Descent. Source:

[http://en.wikipedia.org/wiki/Gradient\\_descent](http://en.wikipedia.org/wiki/Gradient_descent)

# Additional Info: Loss

## Some loss functions

### Regression

Squared Loss

$$\frac{1}{2}(Y_i - \mathbf{x}_i^T \boldsymbol{\beta})^2$$

Absolute Loss

$$|Y_i - \mathbf{x}_i^T \boldsymbol{\beta}|$$

### Classification $\{-1, 1\}$

Log Loss

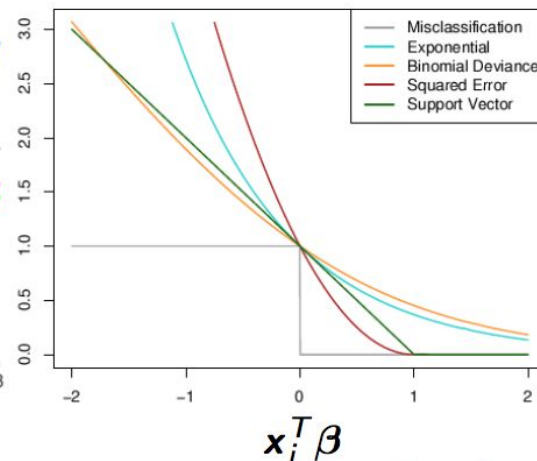
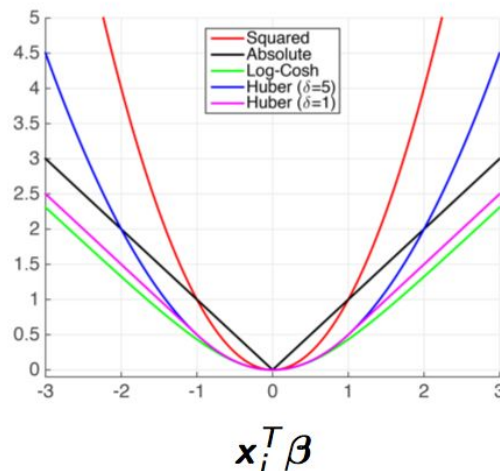
$$\frac{1}{\ln 2} \ln(1 + e^{-Y_i / (1 + e^{-\mathbf{x}_i^T \boldsymbol{\beta}})})$$

(Logistic Reg.)

Exponential Loss

$$\exp(-Y_i \cdot \mathbf{x}_i^T \boldsymbol{\beta})$$

(Ada Boost)





Last poll questions!

galvanize

# Partial Dependence Plots

We can always pull up feature importances from tree based models to give us an intuition as to which features were split on that produced the most information gain.

Does feature importance say anything about what values the splits were made on? Does it say which splits, left or right, correspond to higher output values?

In linear and logistic regressions we used  $\beta$  to tell us about our trends. We don't have a clear linear relationship between our features and our output!

For forest regression models, we want some kind of tool to evaluate how our output varies as a function of a given feature (or two).

# Partial Dependence Plots

One of the best tools we have, though it's not a great tool to be quite honest, is the PDP.

**Premise:** Try to plot the regression output as a function of a continuous predictor,  $X_k$ .

**Problem:** What does “regression output” mean here? We need data points to make predictions.

**Solution:** Hold  $X_k$ , i.e. treat it as a continuous variable instead of a data column filled with values for each data point. For a given value of  $X_k$  that we wish to plot, fill in that column with that value and **average** all the prediction outputs.

**You won't actually need to do this yourself. sklearn has a PDP module.**

# Partial Dependence Plots

