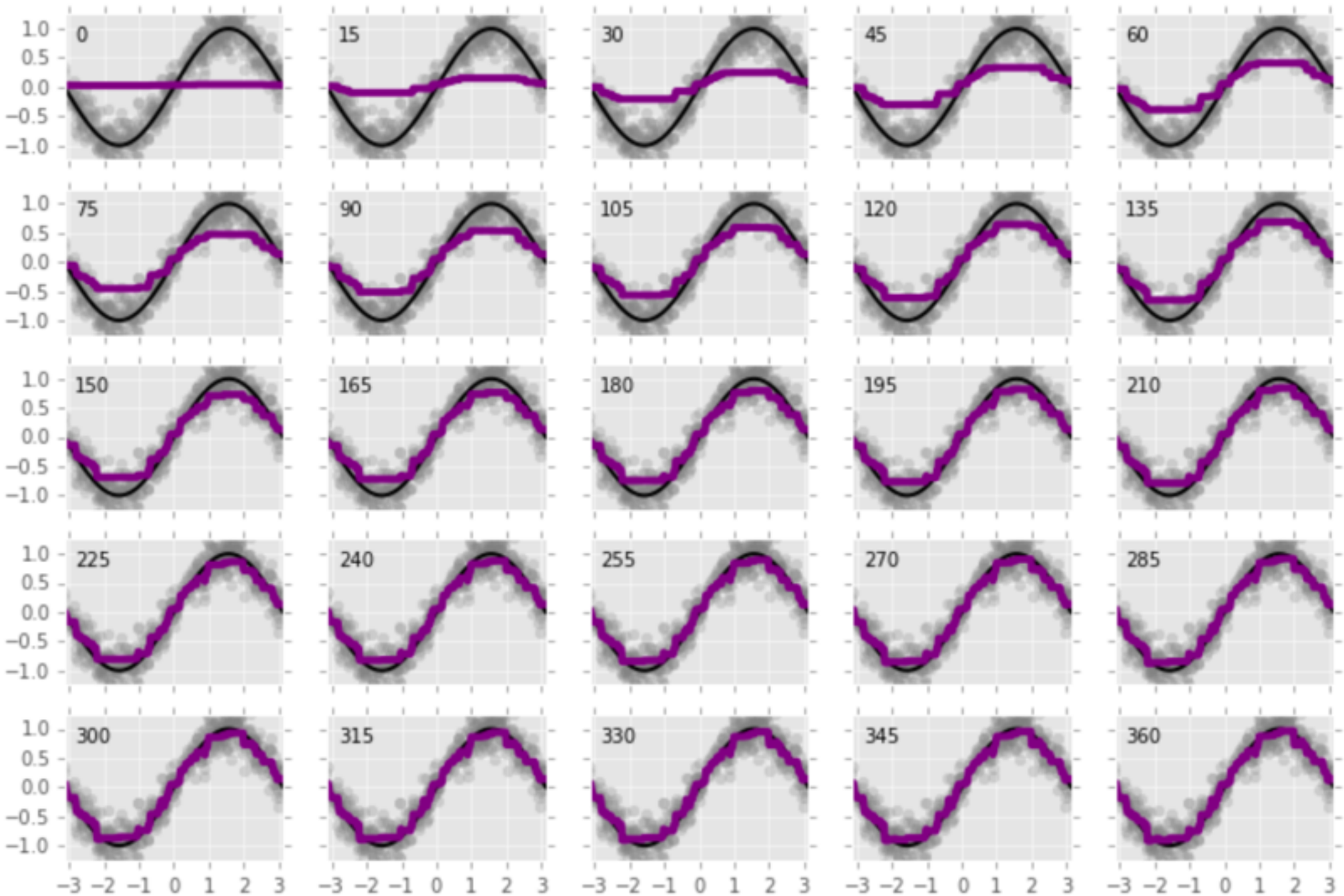
Boosting

Wisdom of the Crowd
aka
A Bunch of Bad Pizza

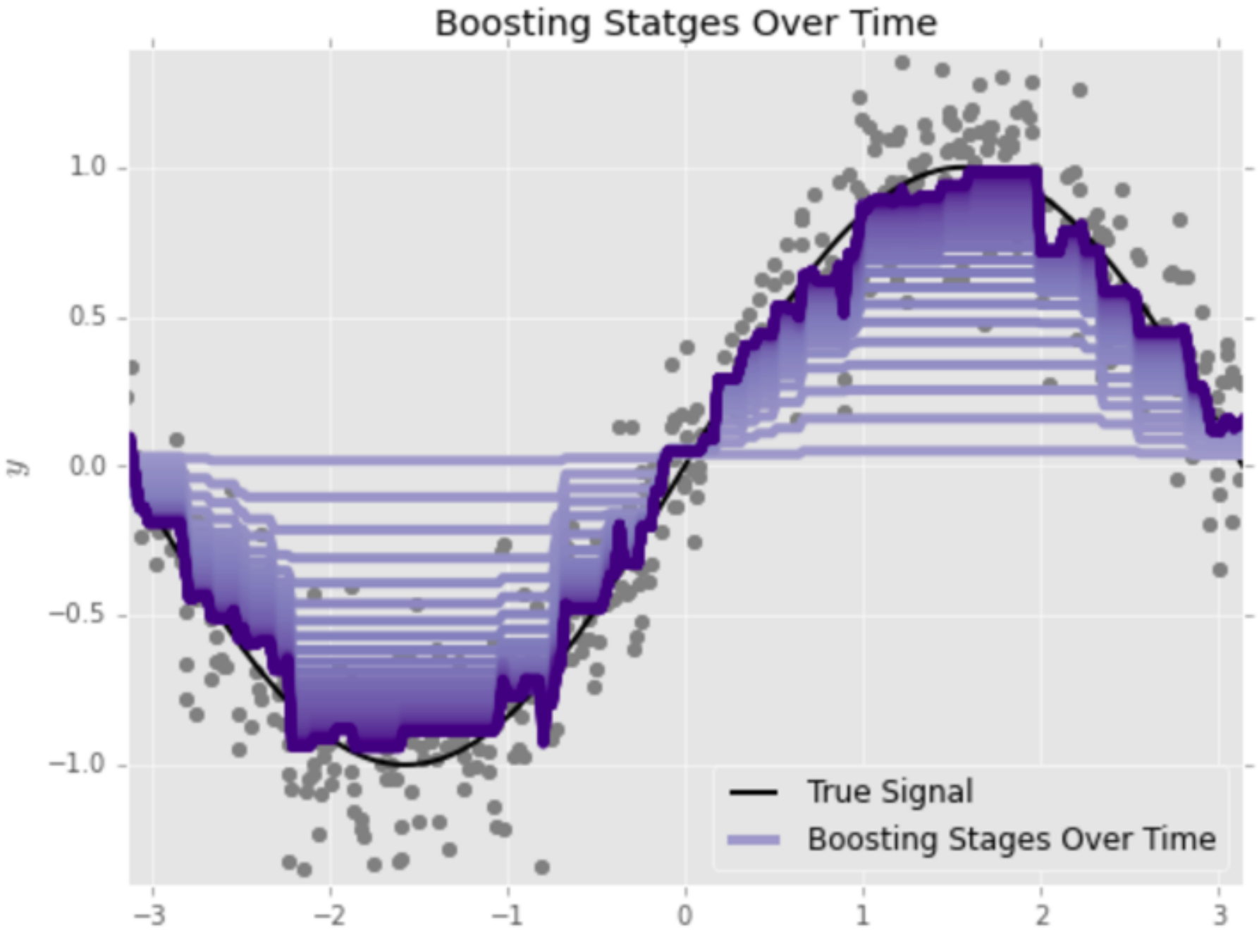
Darren Reger Lecture for Galvanize DSI

*Special thanks to Matthew Drury

Building Slowly Over Time



Stages


Boosting Statges Over Time
1.0
0.5
0.0
−0.5
−1.0
y
−3
−2
−1
0
1
2
3
True Signal
Boosting Stages Over Time

Brief Review:

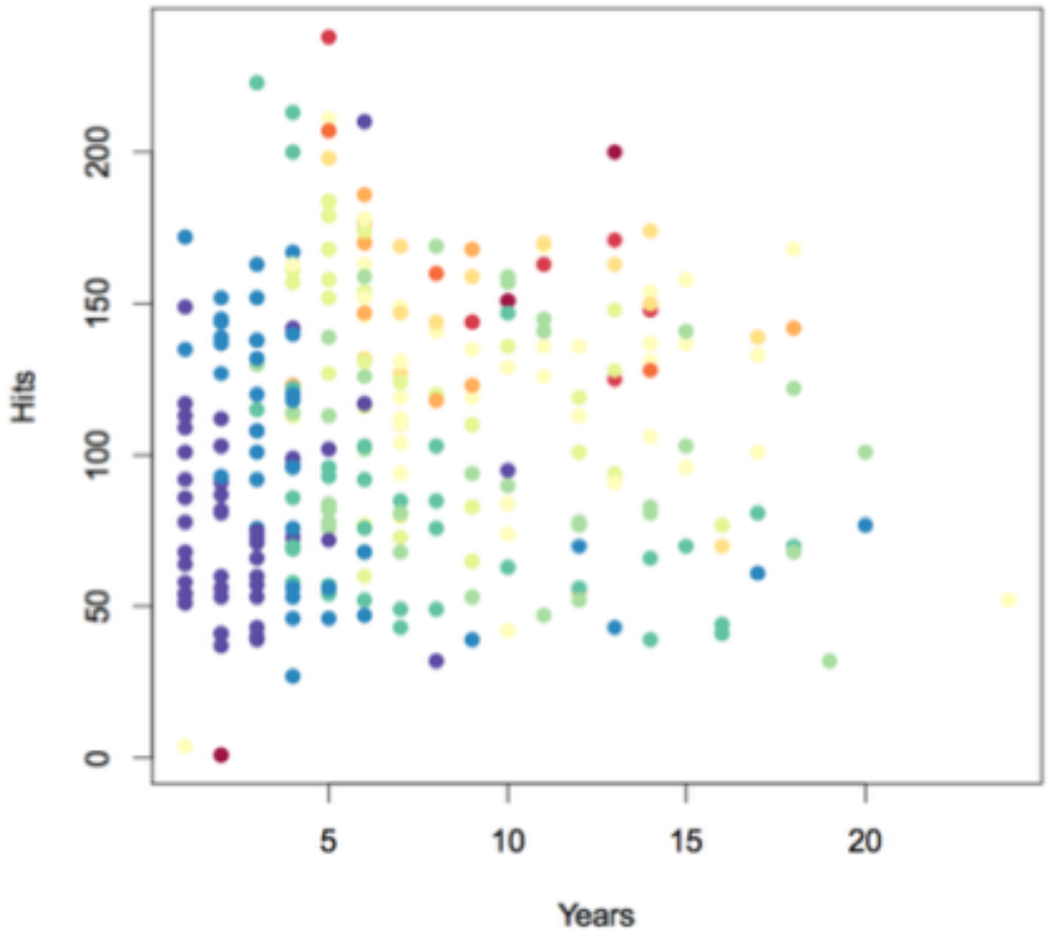
Decision Tree Regression

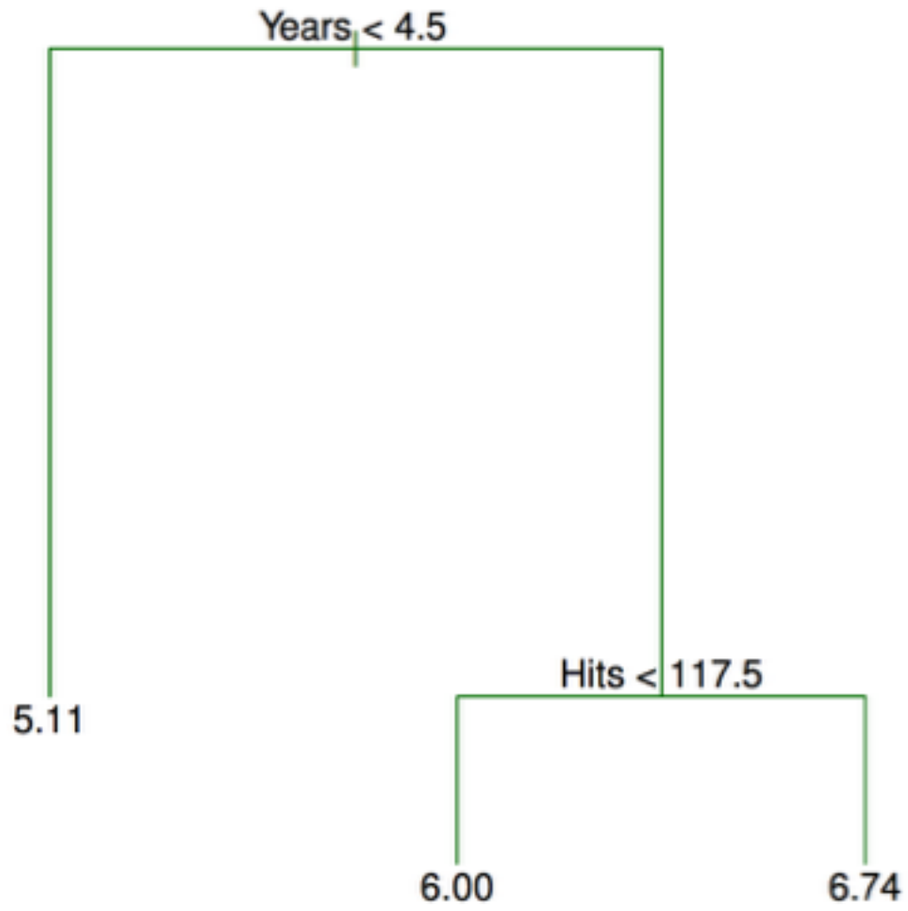
"Some of the figures in this presentation are taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013) with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani "

Decision Trees - Regression

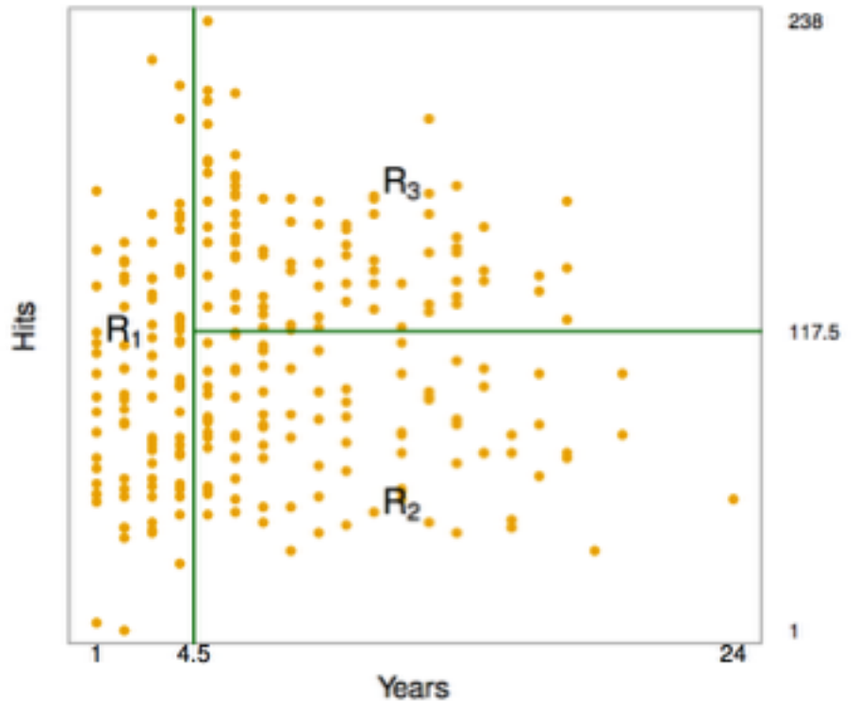
Salaries color-coded from low to high

- Low salaries in blue, green
- High salaries in orange, red





Decision Trees – Regression



Consider sequence of trees indexed by tuning parameter α.

For each α, there is a corresponding subtree, **T**, such that the following is minimized:

$$\sum_{m=1}^{|T|} \sum_{i:\, x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha|T|$$

where |T| is the number of terminal nodes

Boosting Time

Weak Learners

The motivation for boosting was to build an algorithm that combined the outputs of many **weak learners** to produce a powerful committee

- A **weak learner** is defined as an algorithm (model) whose error rate is only slightly better than we would achieve through intelligent random guessing (think **high bias, low variance**)

Starting Our Problem

$\{x_i, y_i\}$ 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}.

If needed, N will be the number of training samples, and M the number of features.

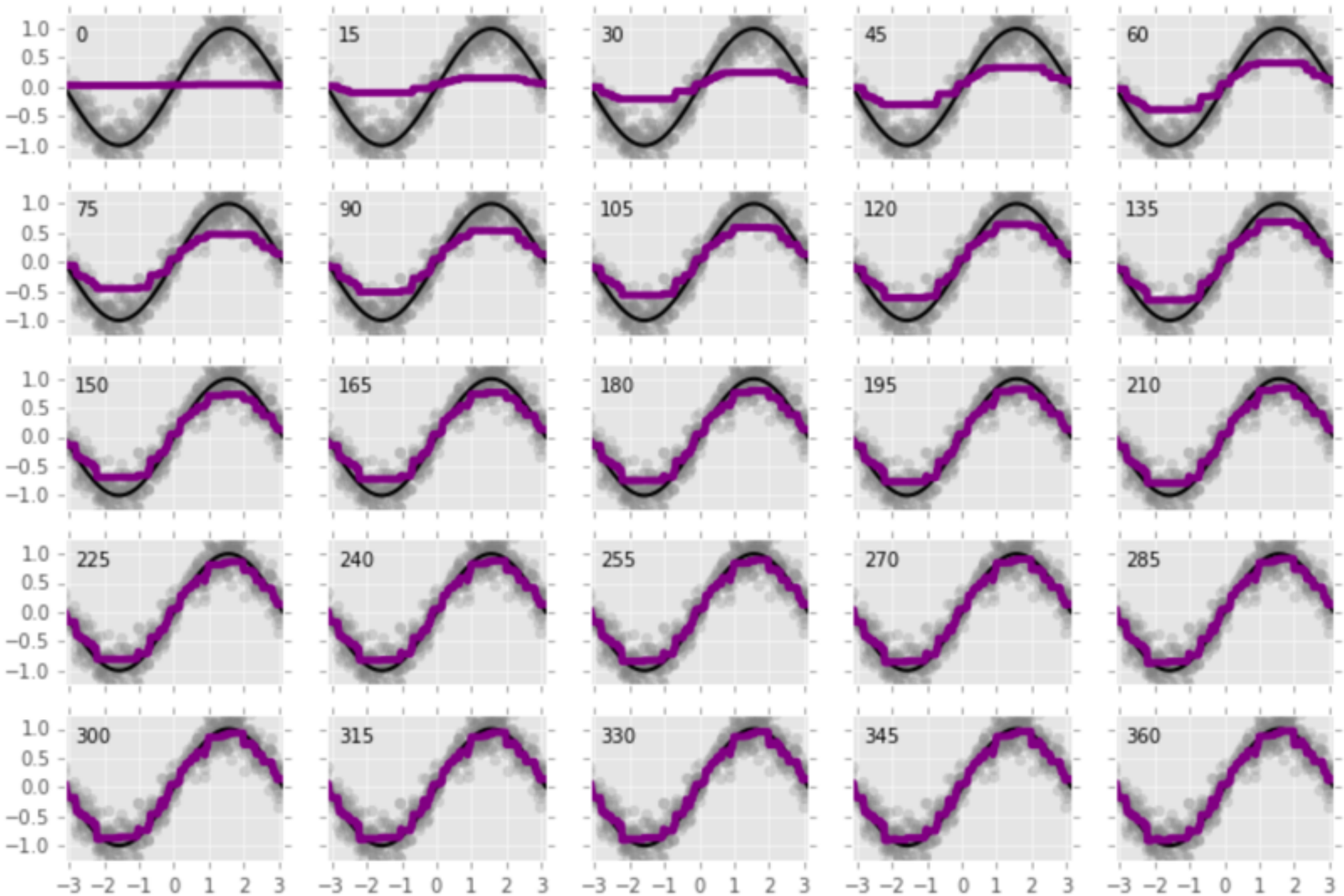
Our goal is to construct a function f so that

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

To be more precise, let's look for an f so that

$$\sum_i (y_i - f(x))^2 \quad \text{is small.}$$

Building Slowly Over Time



Building Slowly

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

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

The process of building up the model looks like

$$\begin{aligned}
S_0(x) &= f_0(x) \\
S_1(x) &= f_0(x) + f_1(x) \\
S_2(x) &= f_0(x) + f_1(x) + f_2(x) \\
&\vdots \\
S_{\text{max}}(x) &= f_0(x) + f_1(x) + f_2(x) + \cdots + f_{\text{max}}(x)
\end{aligned}$$

Starting Out Slow

We are trying to minimize the sum of squared errors, so a good choice would be to find the constant minimizing

$$\sum_i (y_i - \text{constant})^2$$

Starting Out Slow

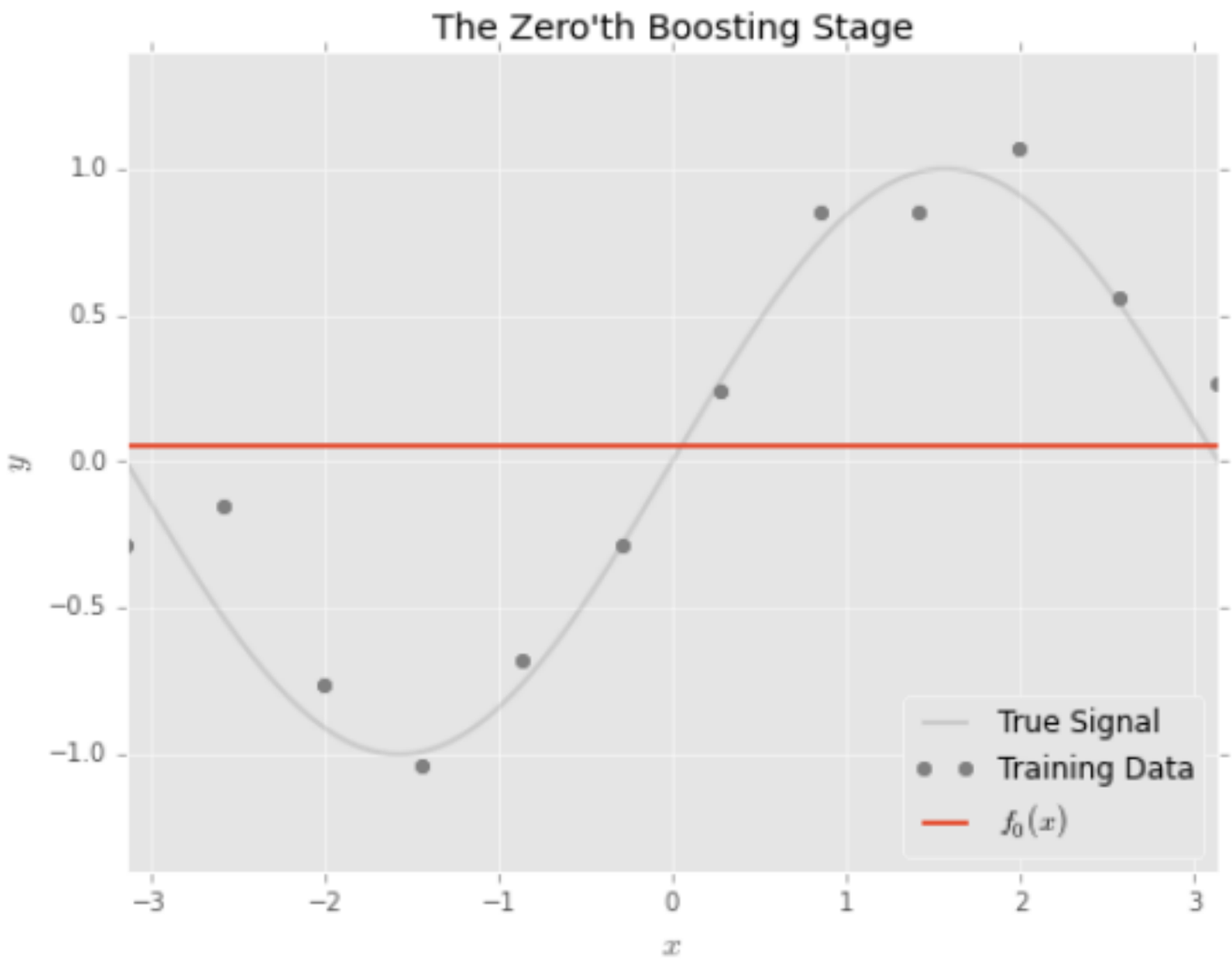
We are trying to minimize the sum of squared errors, so a good choice would be to find the constant minimizing

$$\sum_i (y_i - \text{constant})^2$$

Use the mean!

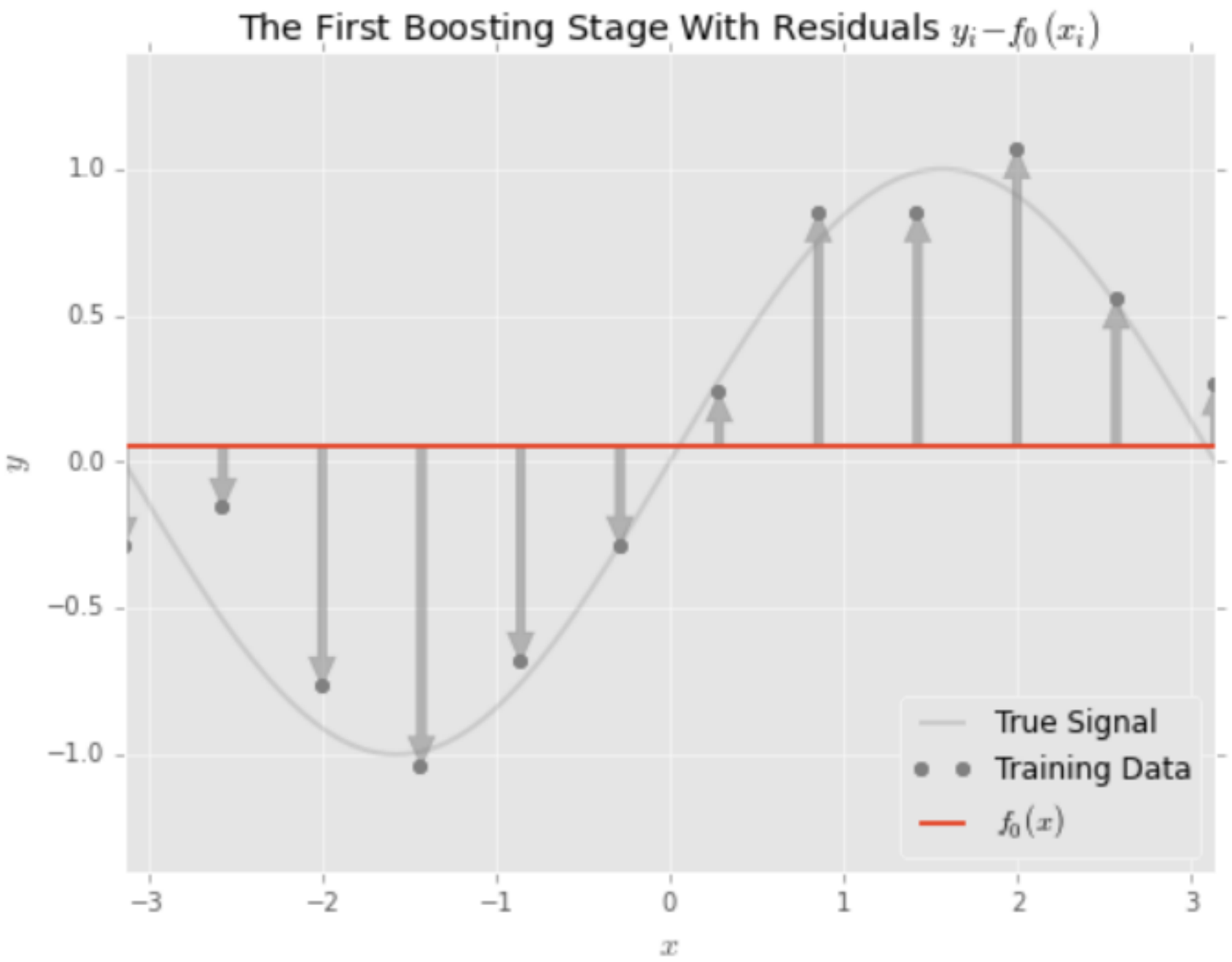
$$f_0(x) = \frac{1}{N} \sum_i y_i$$

How to Update



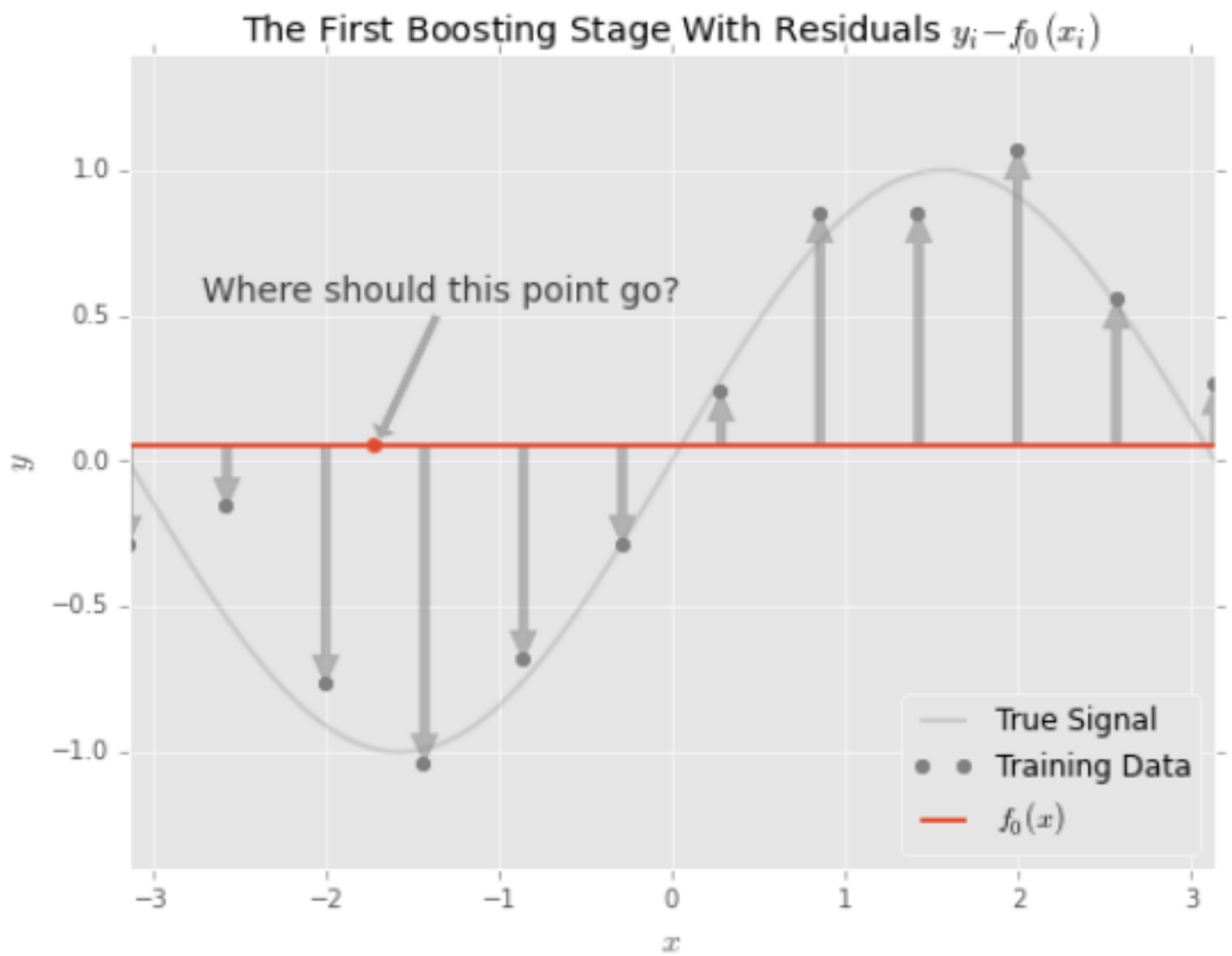
Our next task is to update our simple f_0 to $S_1(x) = f_0(x) + f_1(x)$.

How to Update Some More



Problems Arise

We can only calculate residuals *at the training data points*!



We need some way of **extending** the values of the residuals to places we *do not have data*.

Fitting to the Residuals

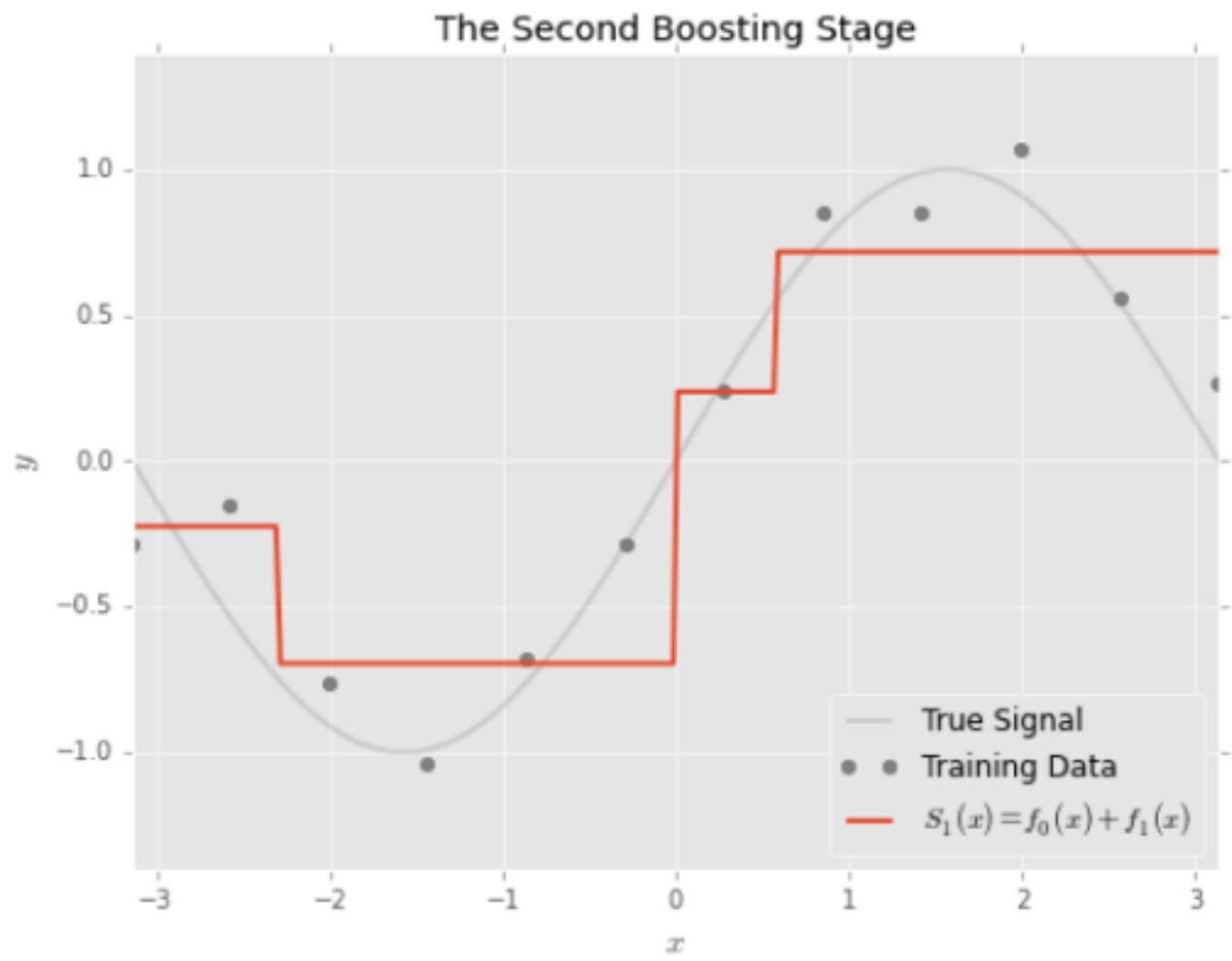
Here is a *new* dataset.

- The values of x are the same as before, taken directly from the training data.
- The response values are the *residuals*: $y_{\text{new}} = y_i - f_0(x_i)$.

Keep On Fitting

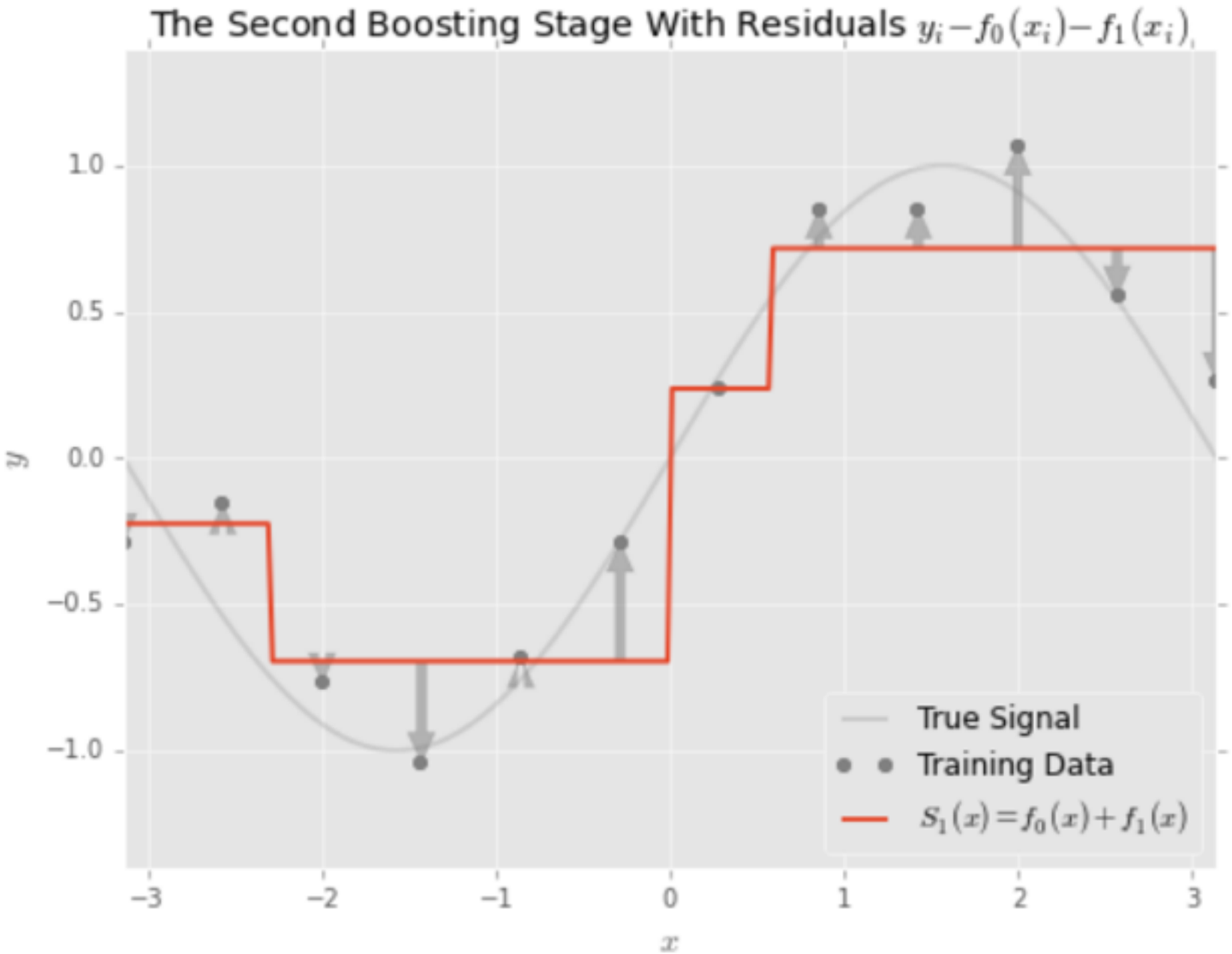
Now we can update the model.

$$S_1(x) = f_0(x) + f_1(x) \leftarrow \text{Model fit to residuals!}$$



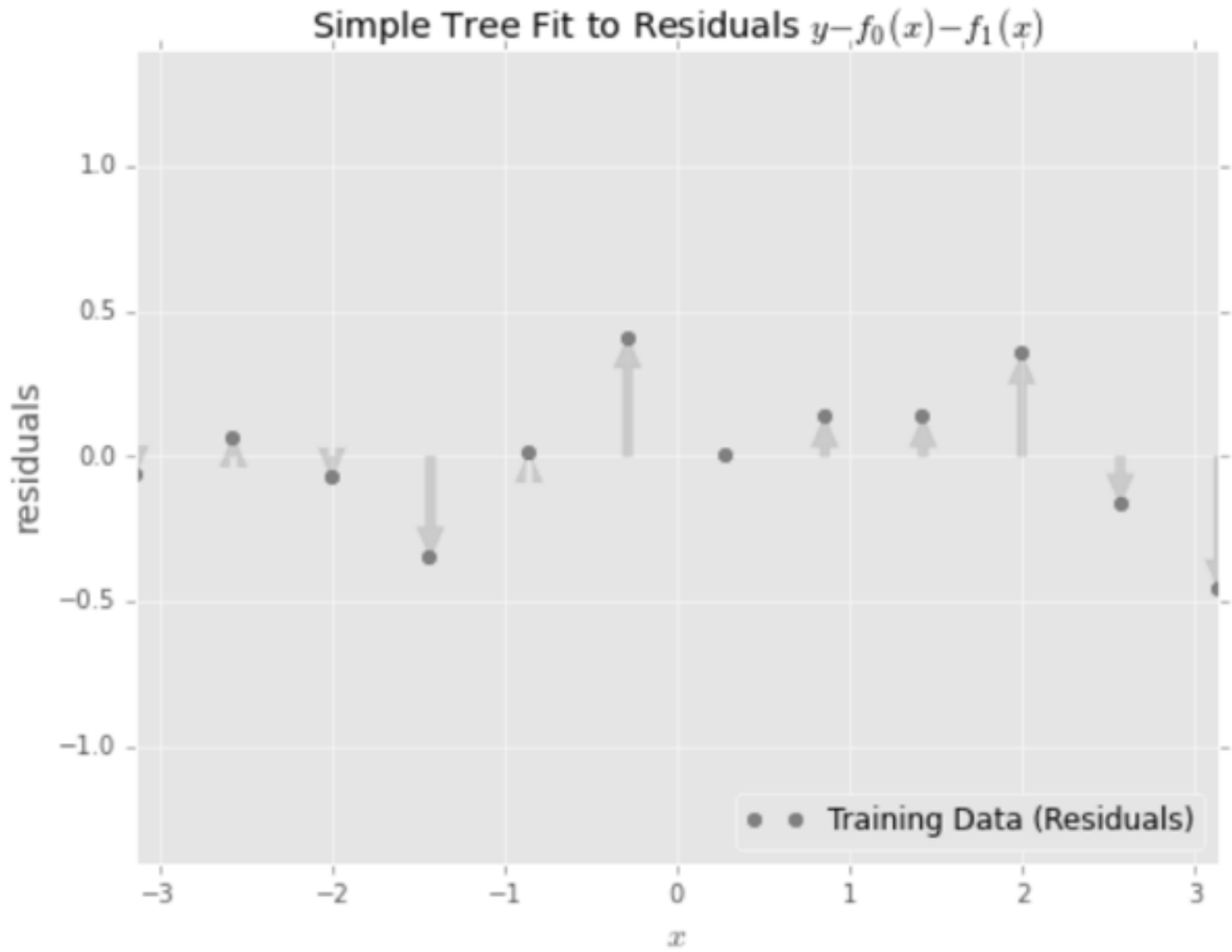
Keep On Fitting

Calculate the residuals of the current model...



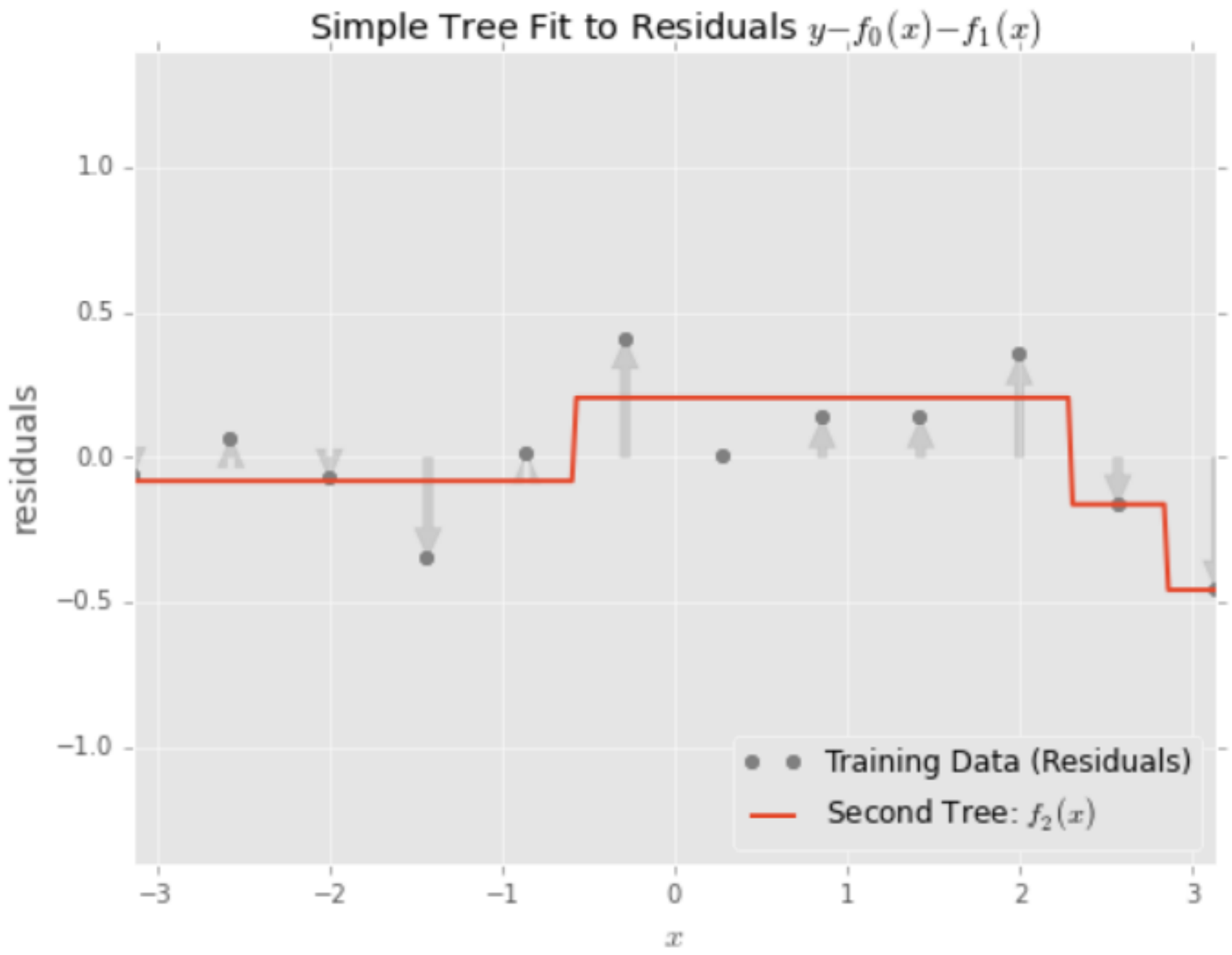
Keep On Fitting

Create a training data set with the residuals as the response...



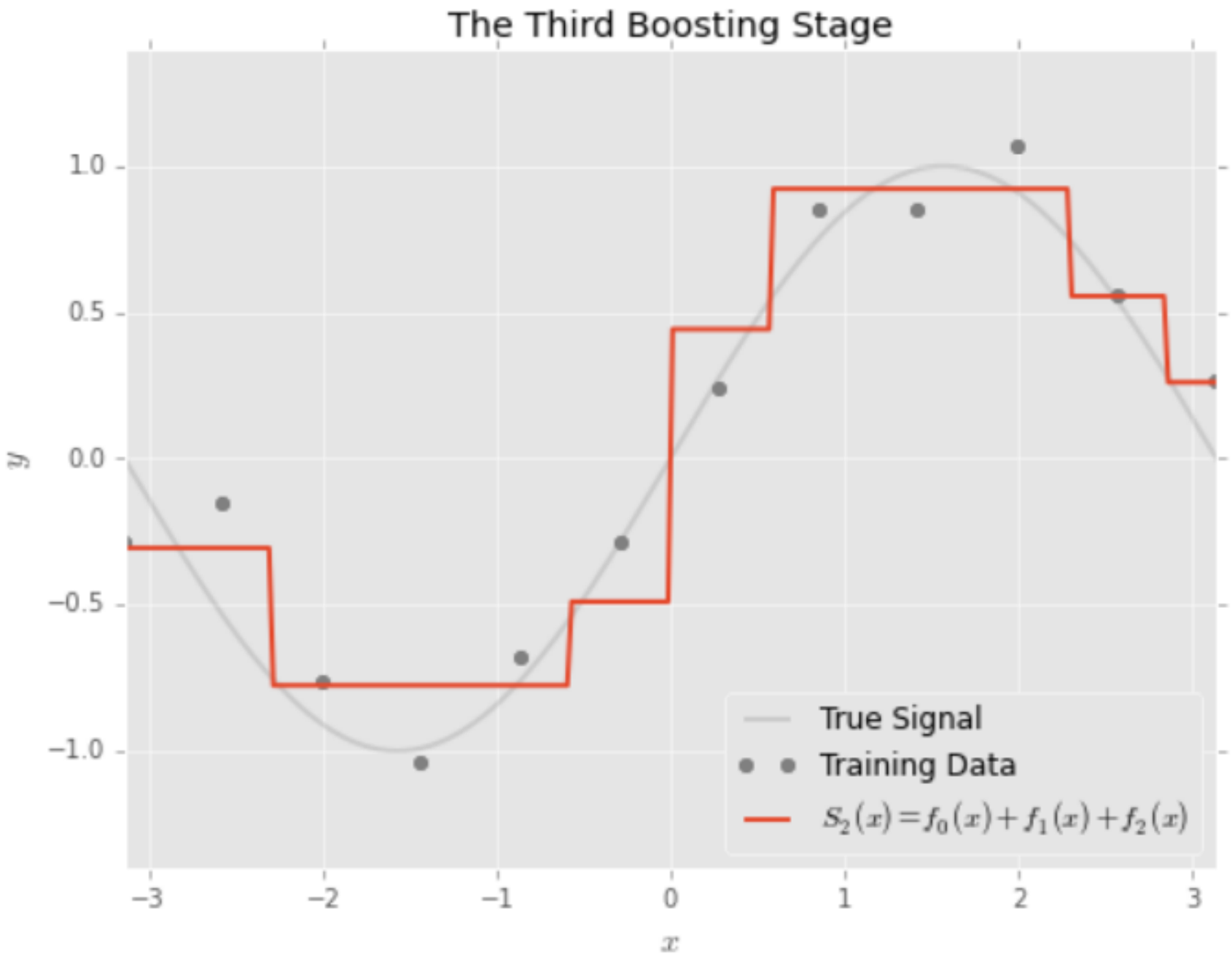
Keep On Fitting

Create a training data set with the residuals as the response...



Keep On Fitting

Update the model!



Take It Slow

Instead of adding in the entirety of the residual fitted tree during an update

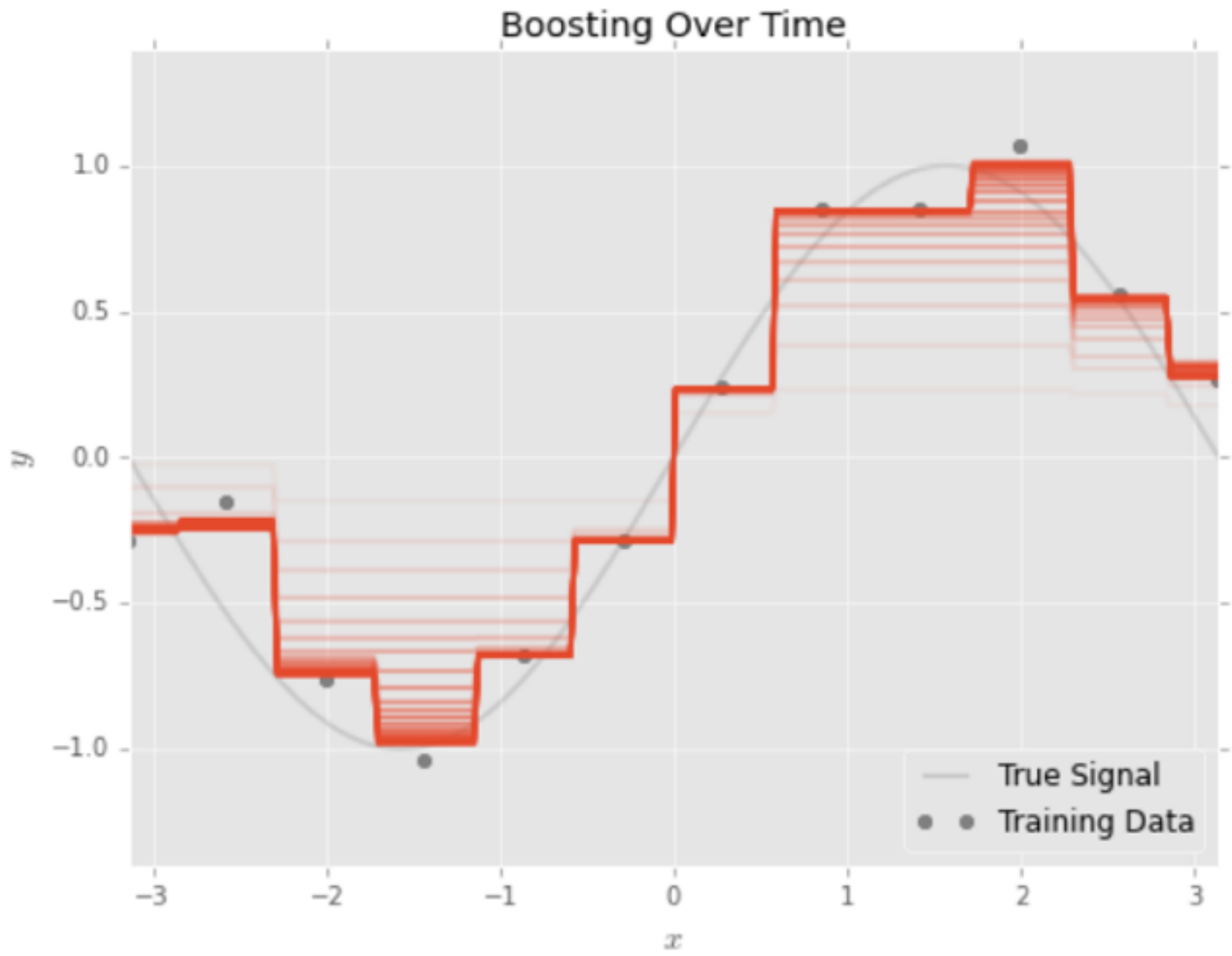
$$S_{k+1}(x) = S_k(x) + f_{k+1}(x)$$

Instead we added some *small fraction* of f_{k+1}

$$S_{k+1}(x) = S_k(x) + \lambda f_{k+1}(x)$$

Be a Tortoise, Not a Hare

$$\lambda = 0.01$$



Mathy Version

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 $F(x_1), F(x_2), ..., F(x_n)$.

Notice that $F(x_1), F(x_2), ..., F(x_n)$ 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.

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

*Description from Cheng Li, Professor at Northeastern University

Mathy Version in Words

How is this related to gradient descent?

For regression with **square loss**,

$$\textit{residual} \Leftrightarrow \textit{negative gradient}$$

$$\textit{fit } h \textit{ to residual} \Leftrightarrow \textit{fit } h \textit{ to negative gradient}$$

$$\textit{update } F \textit{ based on residual} \Leftrightarrow \textit{update } F \textit{ based on negative gradient}$$

So we are actually updating our model using **gradient descent**!
It turns out that the concept of **gradients** is more general and useful than the concept of **residuals**. So from now on, let's stick with gradients. The reason will be explained later.

Mathy Version Clarification

You wish to improve the model such that

$$F(x_1) + h(x_1) = y_1$$
$$F(x_2) + h(x_2) = y_2$$
$$\dots$$
$$F(x_n) + h(x_n) = y_n$$

Or, equivalently, you wish

$$h(x_1) = y_1 - F(x_1)$$
$$h(x_2) = y_2 - F(x_2)$$
$$\dots$$
$$h(x_n) = y_n - F(x_n)$$

Just fit a regression tree h to data
$(x_1, y_1 - F(x_1)), (x_2, y_2 - F(x_2)), ..., (x_n, y_n - F(x_n))$

Relating It Back to GD

Gradient Descent

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

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

$$F(x_i) := F(x_i) + h(x_i)$$

$$F(x_i) := F(x_i) + y_i - F(x_i)$$

$$F(x_i) := F(x_i) - 1 \frac{\partial J}{\partial F(x_i)}$$

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

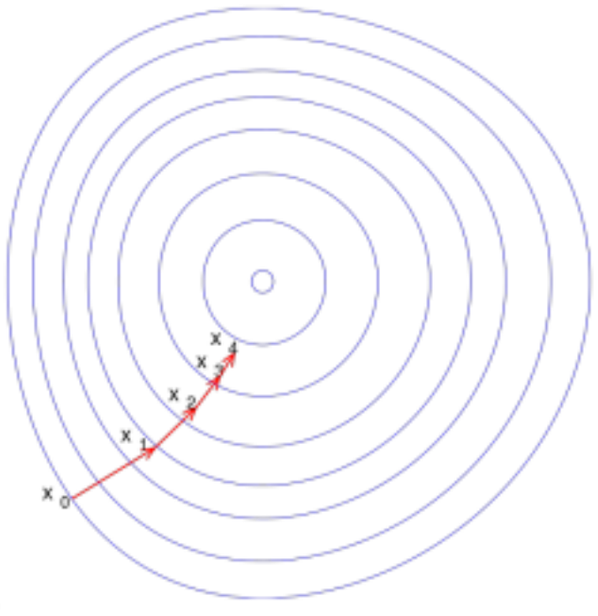


Figure : Gradient Descent. Source: `http://en.wikipedia.org/wiki/Gradient_descent`

General Form of Gradient Boosting

Give any differentiable loss function L

start with an initial model, say $F(x) = \frac{\sum_{i=1}^{n} y_i}{n}$
iterate until converge:
calculate negative gradients $-g(x_i) = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$
fit a regression tree h to negative gradients $-g(x_i)$
$F := F + \rho h$

In general,

$$\textit{negative gradients} \not\Leftrightarrow \textit{residuals}$$

We should follow negative gradients rather than residuals. Why?

Other Loss Functions

Know that they exist. You might use them some day.

Absolute loss (more robust to outliers)

$$L(y, F) = |y - F|$$

Huber loss (more robust to outliers)

$$L(y, F) = \begin{cases} \frac{1}{2}(y - F)^2 & |y - F| \leq \delta \\ \delta(|y - F| - \delta/2) & |y - F| > \delta \end{cases}$$

Why not just fit to the residual?

Huber loss

$$L(y, F) = \begin{cases} \frac{1}{2}(y - F)^2 & |y - F| \leq \delta \\ \delta(|y - F| - \delta/2) & |y - F| > \delta \end{cases}$$

Update by Negative Gradient:

$$h(x_i) = -g(x_i) = \begin{cases} y_i - F(x_i) & |y_i - F(x_i)| \leq \delta \\ \delta sign(y_i - F(x_i)) & |y_i - F(x_i)| > \delta \end{cases}$$

Update by Residual:

$$h(x_i) = y_i - F(x_i)$$

Difference: negative gradient pays less attention to outliers.

Summary

Fit an additive model $F = \sum_t \rho_t h_t$ in a forward stage-wise manner.

In each stage, introduce a new regression tree h to compensate the shortcomings of existing model.

The "shortcomings" are identified by negative gradients.

For any loss function, we can derive a gradient boosting algorithm.

Absolute loss and Huber loss are more robust to outliers than square loss.

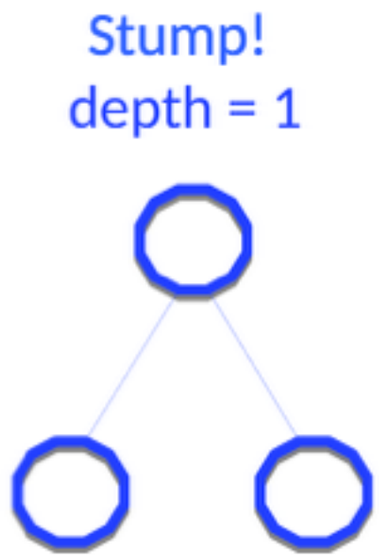
Parameter tuning

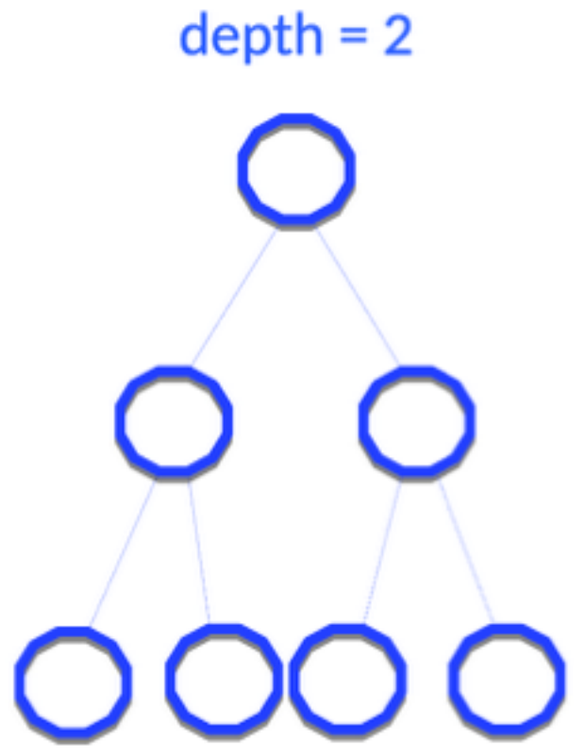
- Must set tuning parameters correctly to obtain optimal performance
- Parameters control:
  - Tree structure
  - Shrinkage
  - Stochastic gradient boosting

Tree structure

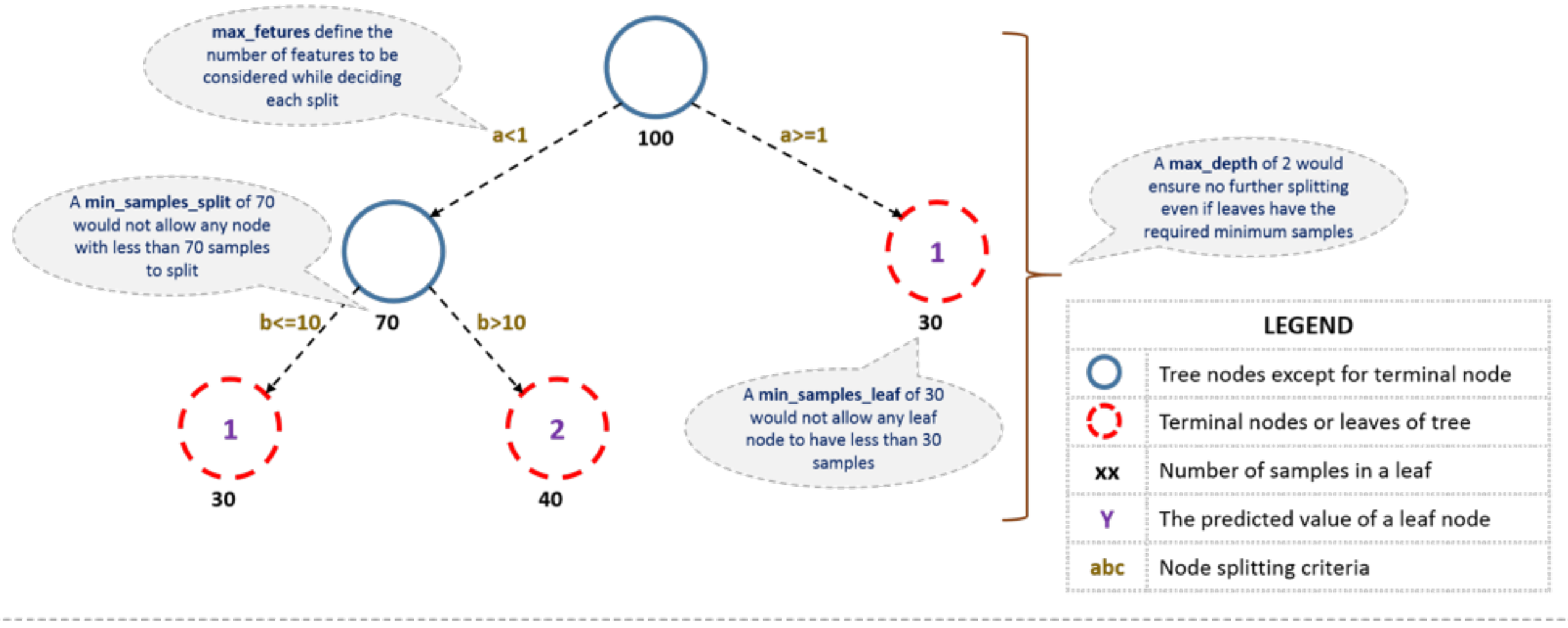
- `max_depth`
  - controls degree of interactions
  - Ex. Latitude and Longitude
  - not often larger than 4 or 6

- `min_samples_per_leaf`
  - may not want terminal nodes with too few leaves





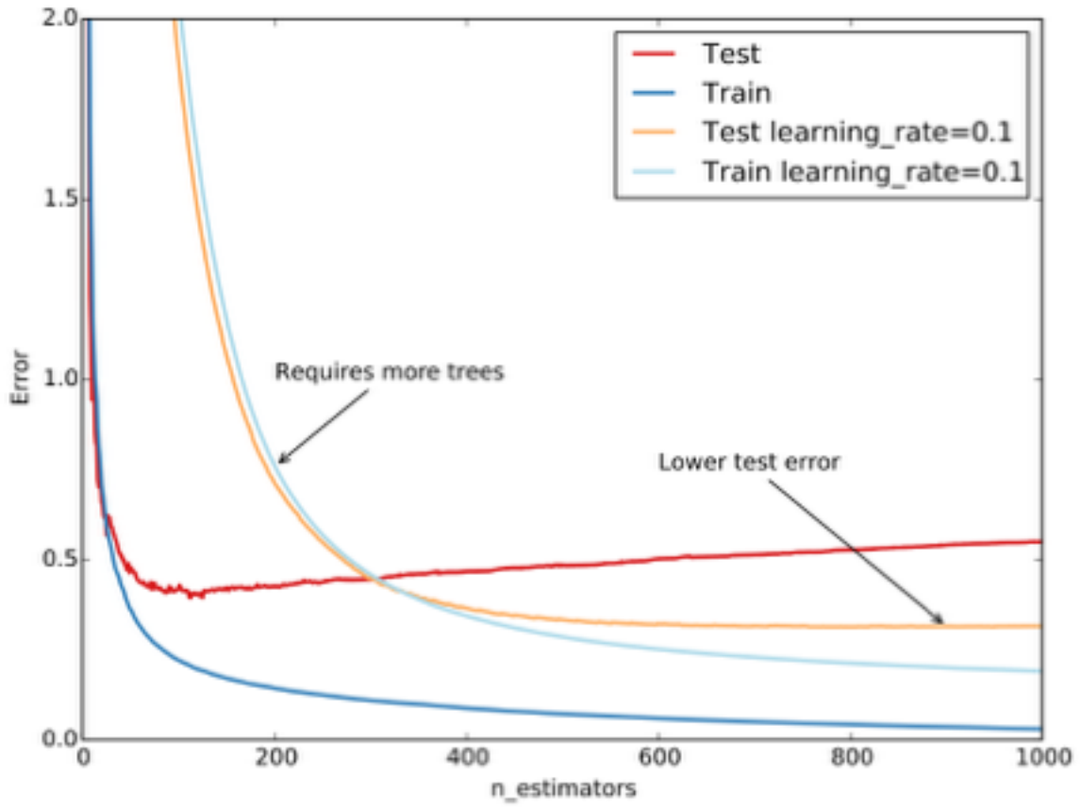
Things we can change



Shrinkage

- `n_estimators`
  – number of trees grown

- `learning_rate`
  – lower learning rate requires higher n_estimators



As the learning rate goes down, the number of trees needed goes up!

Learning rate is a very important tuning parameter.
Number of trees also needs to be tuned.

Stochastic gradient boosting

- `max_features`
  - random subsample of features
  - Especially good when you have lots of features

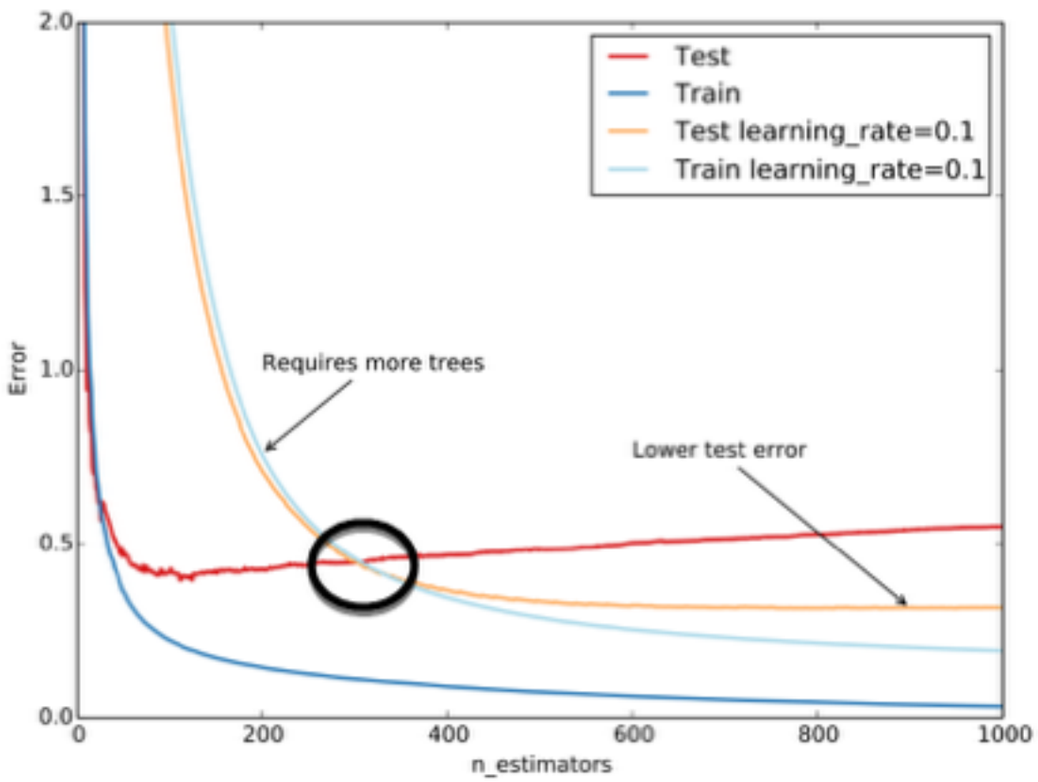
- `sub_sample`
  - random subset of the training set

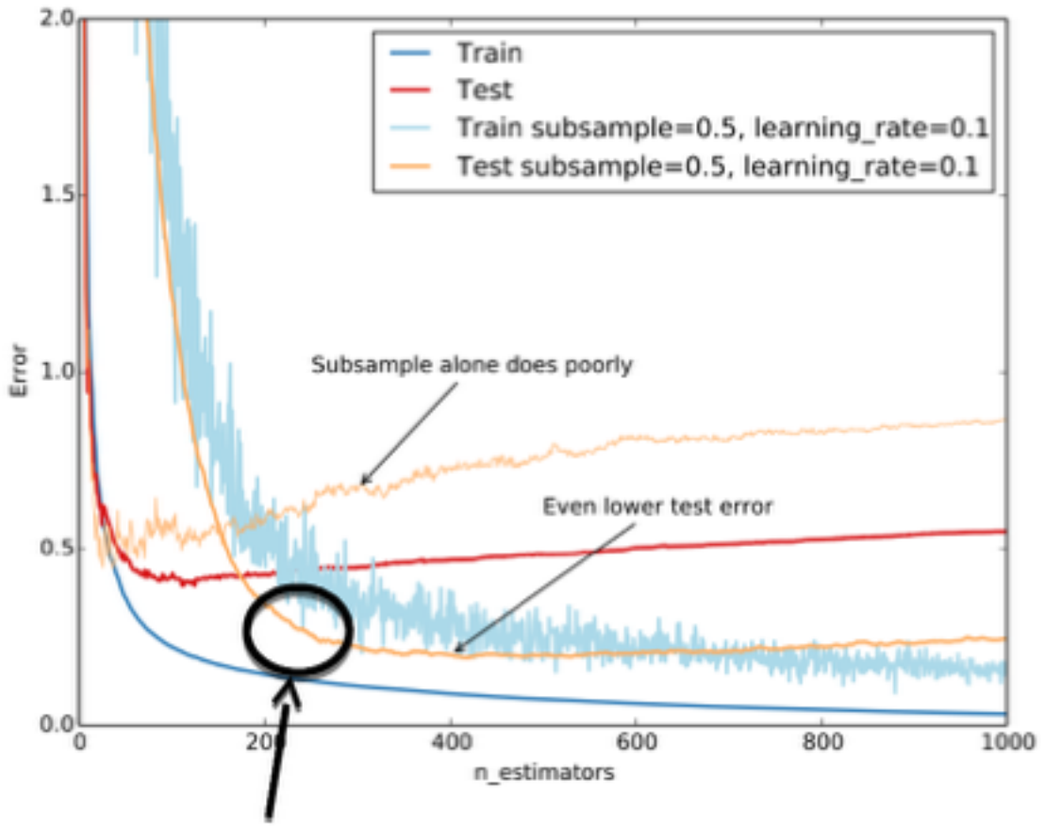
Both randomly sampling the features and randomly subsetting the training set can lead to improved accuracy and reduced run-time

Pretty good deal!

Example: stochastic gradient boosting

Can **improve accuracy** and **reduce runtime!**





Lower test error!
Fewer trees to get there!

Set Those Things

```
from sklearn.grid_search import GridSearchCV
param_grid = {'learning_rate': [0.1, 0.05, 0.02, 0.01]
              'max_depth': [4, 6],
              'min_samples_leaf': [3, 5, 9, 17],
              'max_features': [1.0, 0.3, 0.1]}
est = GradientBoostingRegressor(n_estimators=3000)
gs_cv = GridSearchCV(est, param_grid).fit(X, y)
# best hyperparameter setting
gs_cv.best_params_
```

Best practice

Scikit-learn recommends:

1.Set `n_estimators` high as possible (3000, 5000, etc.)
2.Tune all hyper-parameters via grid search
3. Further optimize `n_estimators` and `learning_rate`

- Scikit-learn uses canonical implementation from R's gbm package.
- gbm's documentation provides additional advice on tuning.

→ You must tune the model to get good performance!

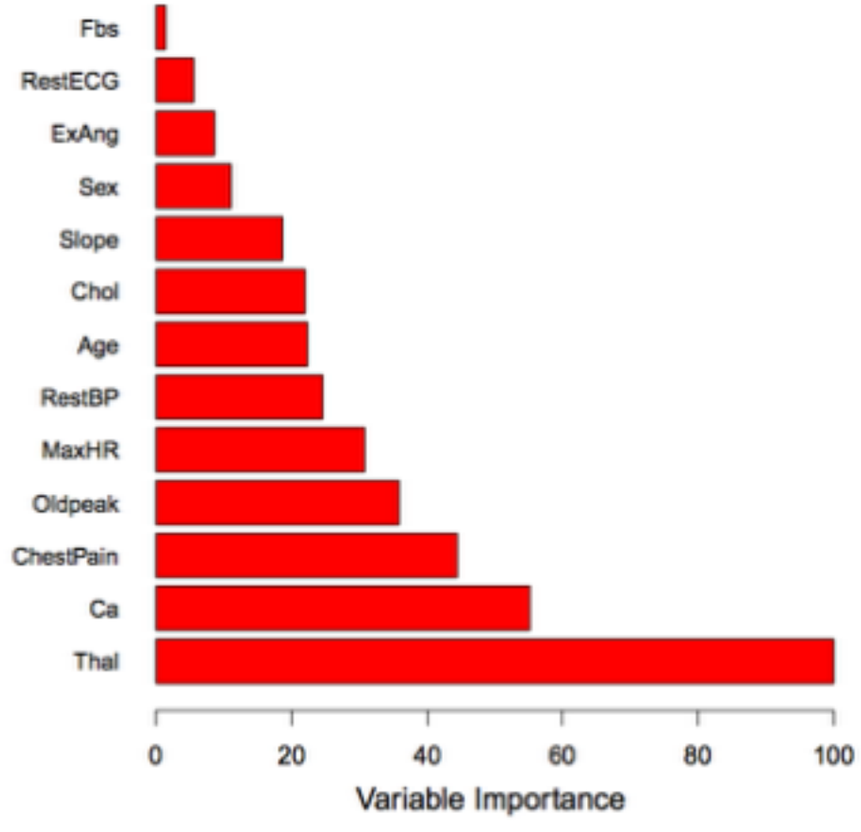
Morning Sprint Break

Diagnostics

- Need to understand strengths & weaknesses of model
- Which features matter?
- Which interactions matter?
- Does this make sense based on your domain knowledge?
- Diagnostic tools:
  - Variable importance
  - Partial dependence plots

Variable Importance

- For bagged/RF regression trees, we record the total amount that the RSS is decreased due to splits over a given predictor, averaged over all B trees. A large value indicates an important predictor.
- Similarly, for bagged/RF classification trees, we add up the total amount that the Gini index is decreased by splits over a given predictor, averaged over all B trees.



Variable importance plot for the `Heart` data

Partial Dependency Plots

With **partial dependency plots**, we have a useful tool for teasing out and quantifying the effects of an individual variable on our response

Effectively, **after fitting the model**, we'll cycle over some pre-determined values of the individual variable of interest, predicting on those values and observing how our responses changes

How the response changes across different values of our variable of interest is the **partial dependency** of the response on that variable

Partial Dependency Plots

To calculate the partial dependence by hand, a common way of doing it for a single variable is the following:

1. Fit our machine learning model/algorithm - this should be able to basically tease out all of the average effects of each individual variable
2. Pick a variable that you would like to calculate the partial dependency of
3. Pick a range of values that you want to calculate the partial dependency for
4. Loop over those values, one at a time doing the following for **each value**:
    1. Replace the entire column corresponding to the variable of interest with the current value that is being cycled over (we'll do this with our training set)
    2. Use the model to `predict` (again with the training data)
    3. Average all of the reponses, and calculate the difference of this average to the average calculated in the last iteration of the loop
    4. This (`value`, `difference`) becomes an (`x`, `y`) pair for your partial dependency plot

Partial Dependency Plots

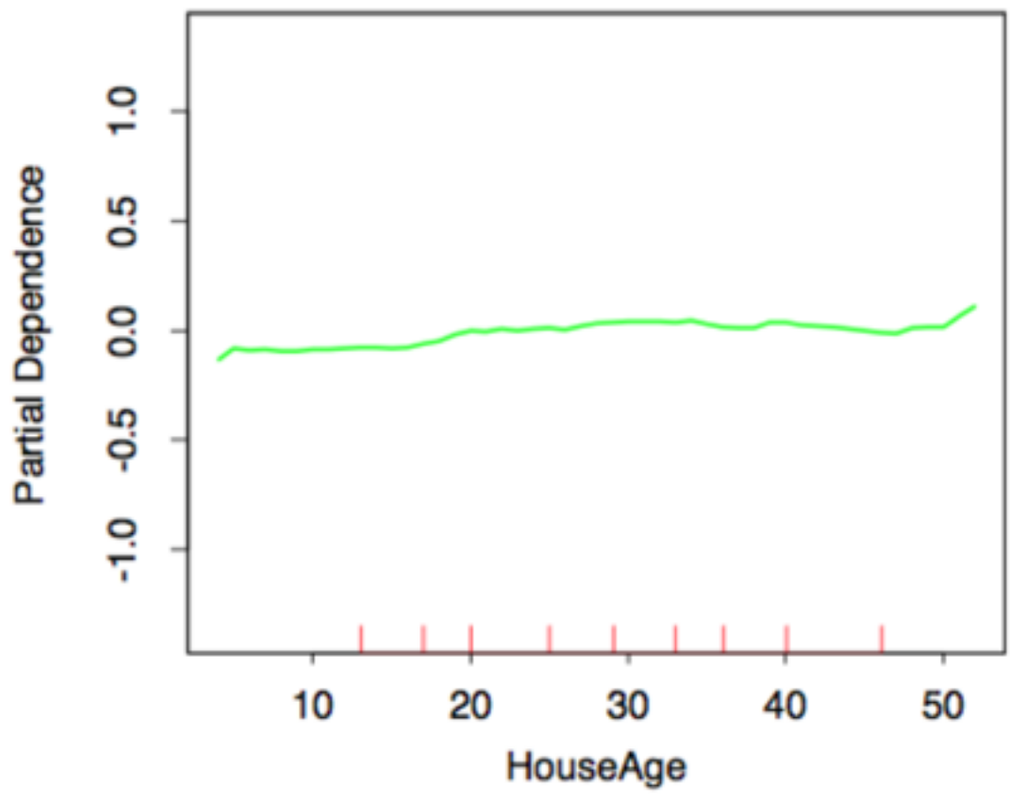


Figure 16:Partial dependence of median house value on median age of houses in the neighborhood

- Here, we can see that once we control for the average effects of all other variables, median house value has a small partial dependence on median age of the house

Partial Dependency Plots

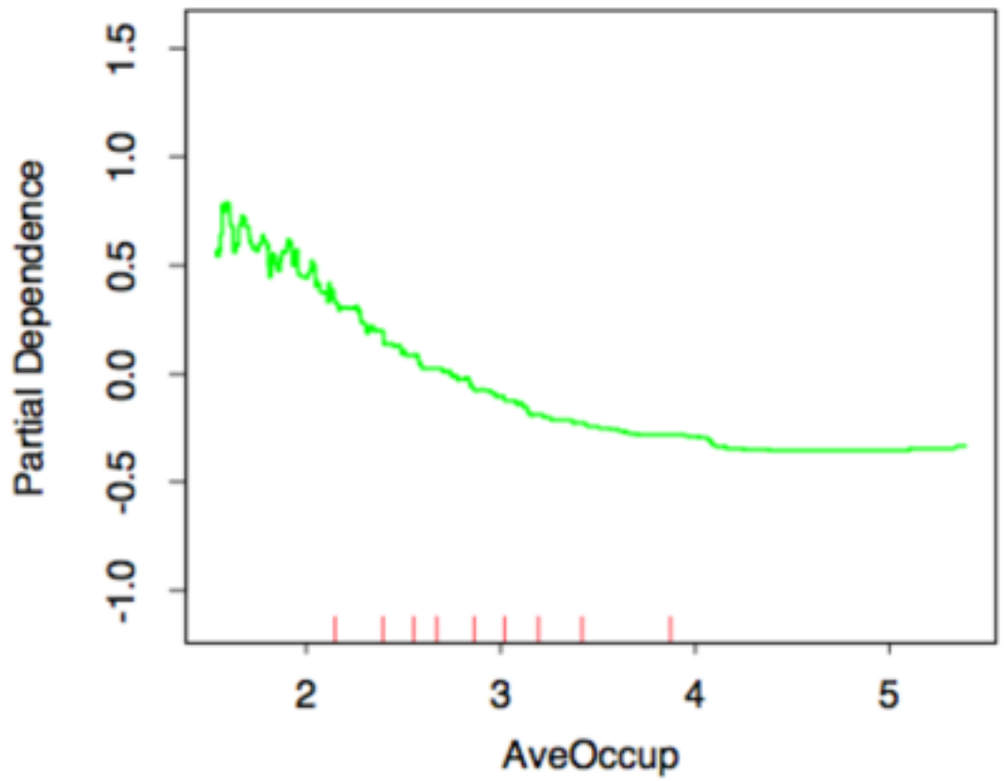


Figure 17:Partial dependence of median house value on average occupancy of houses in the neighborhood

- Here, we can see that once we control for the average effects of all other variables, median house value has a noticeable partial dependence on the average occupancy of houses in the neighborhood.

Now in 3D!

- We can even plot the partial dependency of two variables relative to the response (more than two gets tough):

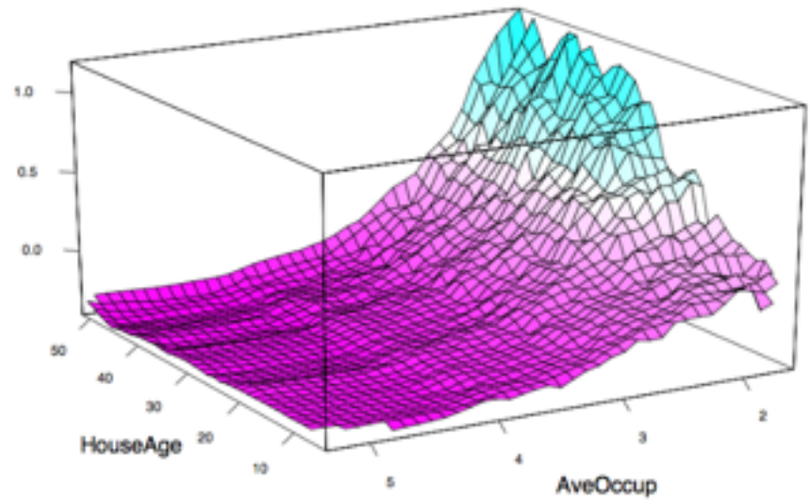


Figure 18:Partial dependence of median house value on median house age and average occupancy

- Here, we see that there is a strong interaction between `HouseAge` and `AveOccup`, which we weren't able to see in looking at either the feature importances or partial dependency plots of a single variable

Adaboost

Adaboost

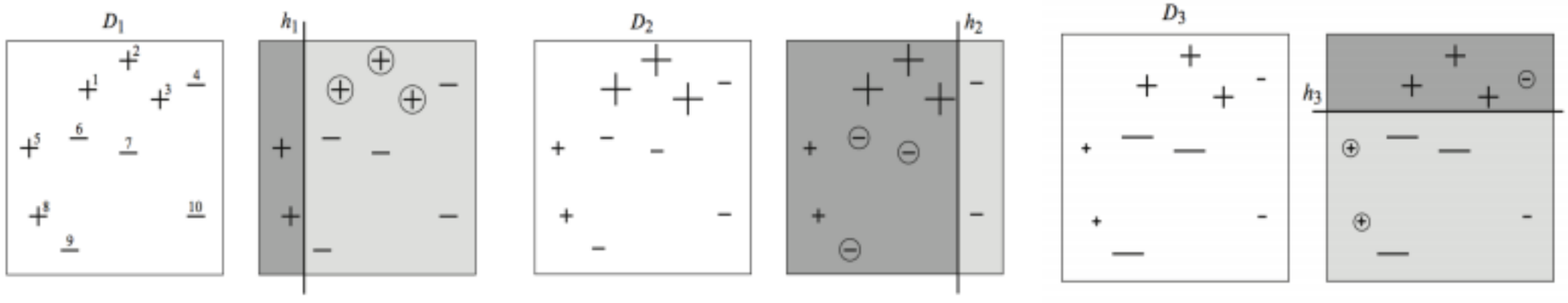


Figure : AdaBoost. Source: Figure 1.1 of [Schapire and Freund, 2012]

- Fit an additive model (ensemble) $\sum_t \rho_t h_t(x)$ in a forward stage-wise manner.
- In each stage, introduce a weak learner to compensate the shortcomings of existing weak learners.
- In Adaboost, "shortcomings" are identified by high-weight data points.

Adaboost

Adaboost

$$H(x) = \sum_t \rho_t h_t(x)$$

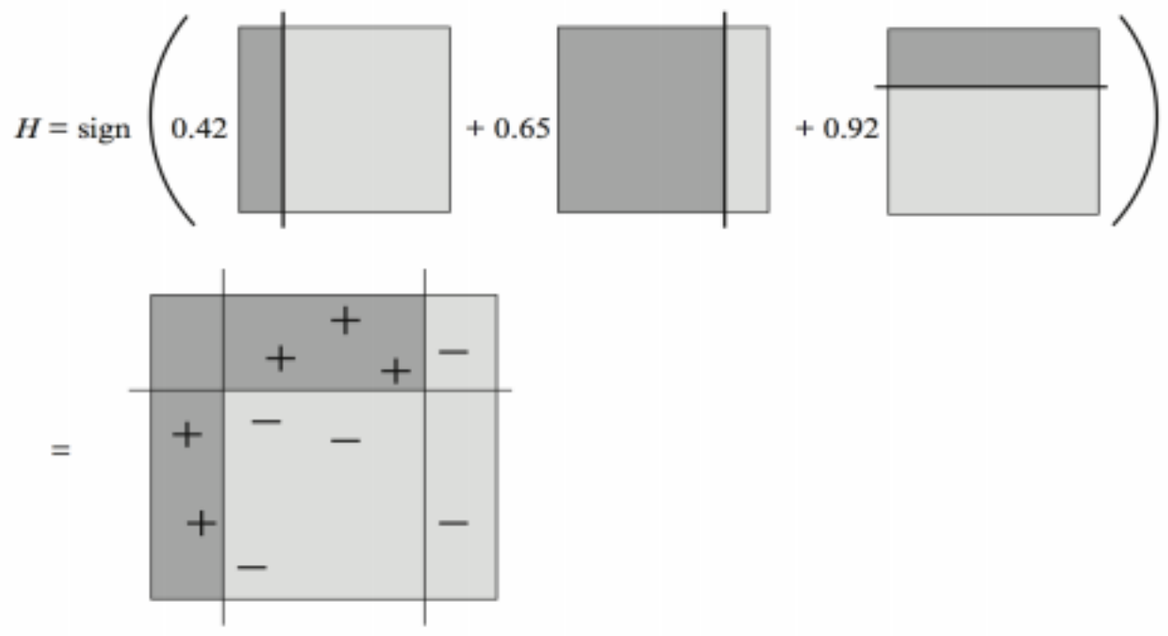


Figure : AdaBoost. Source: Figure 1.2 of [Schapire and Freund, 2012]

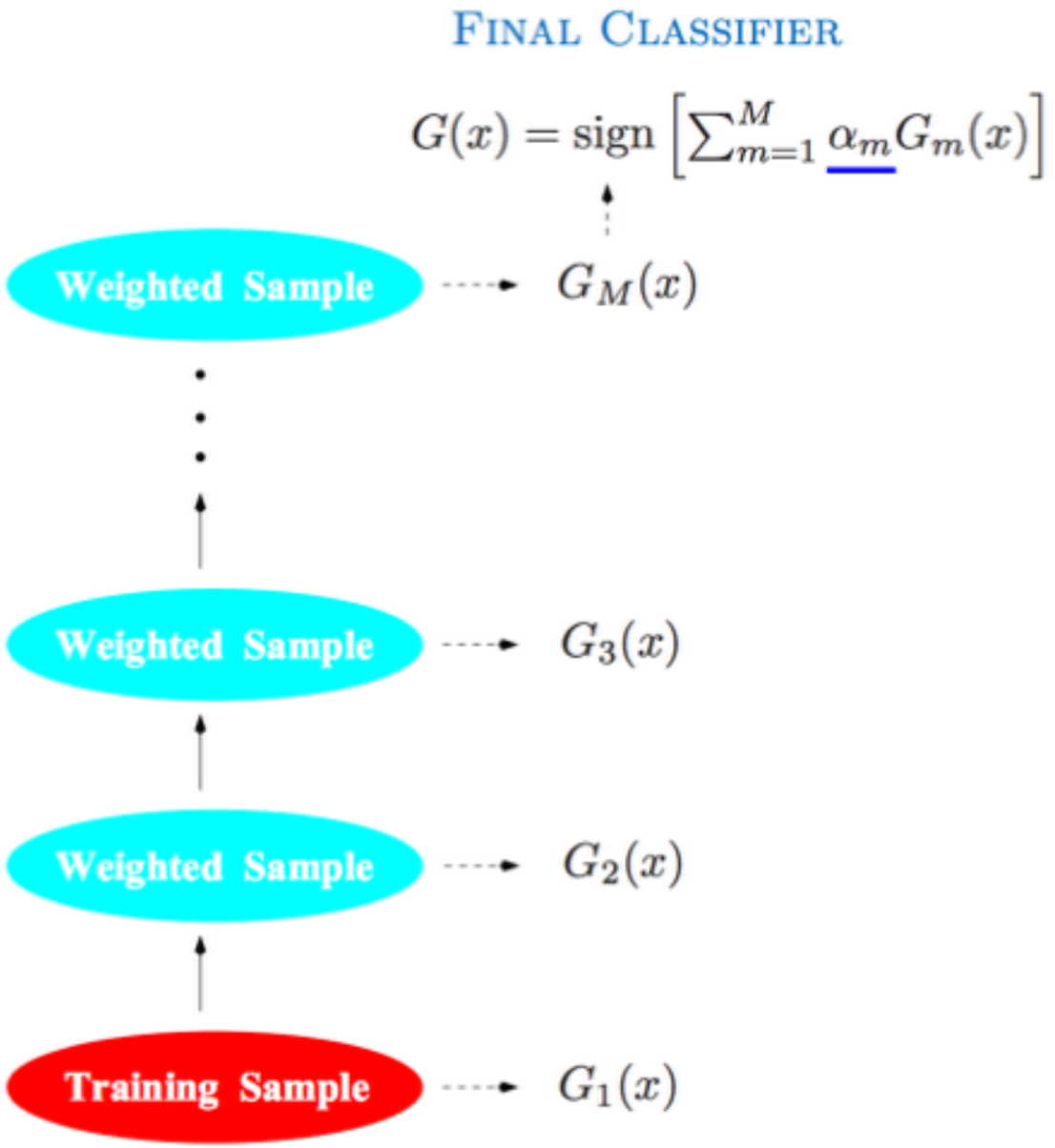
Same thing as on previous page, shown in a different way

<u>Discrete AdaBoost</u>

$G_i(x)$ weak classifiers

$G(x)$ strong learner

Note only $G_1(x)$ fit on training



Discrete AdaBoost

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

XGBoost

XGBoost (short for extreme gradient boosting) is gradient boosting with some niceties built in that make it much faster and more efficient than standard gradient boosting

The primary change from standard gradient boosting is that they bin observation's column values into percentiles when performing splitting

1. For some j_{th}, column, we split all observations up into percentiles based off their values for that j_{th} column
2. Then, only consider some average value (mean, median, etc.) across each percentile for splitting (this narrows the search space quite a bit)

Other changes that XGBoost includes are:

- Handles sparsity in the data (e.g. missing values)
- Handles mixed data types (e.g. categoricals, continuous)
- Allows for out-of-core computation
- Builds in smart memory management

How You Can XGBoost

- It can be installed using `pip`, and it does have an `sklearn` interface (e.g. `fit`, `predict` methods, etc.)

```
import xgboost as xgb
gbm = xgb.XGBClassifier(max_depth=3, n_estimators=300,
                        learning_rate=0.05)
                       .fit(train_X, train_y)
predictions = gbm.predict(test_X)
```

Whiteboard Example If There Is Time

All Slides After This Are Arbitrary Appendix Slides

We have at this point been mainly talking about trees as our **weak learner**, but note that we could use any model as a weak learner, subject to the degree with which we can train that model to have an error rate only **slightly better** than intelligent random guessing

- It turns out that in practice, boosting anything but trees isn't terribly worthwhile (e.g. it doesn't improve performance that much)
- As such, we pretty much only boost trees

- **AdaBoost** (short for Adaptive Boosting) is gradient boosting when we plug in an **exponential loss** function
- It has a nice interpretation of assigning weights to individual observations, and then iteratively adjusting those weights based on how well we are predicting for each individual observation (sounds familiar, right?)

```
gs_cv = GridSearchCV(model, param_grid).fit(X, y)
```

- When this is called, the GridSearchCV is going to iterate over every possible combination of parameters that could be created given the parameters in our `param_grid`, create folds for the data, and for each fold:

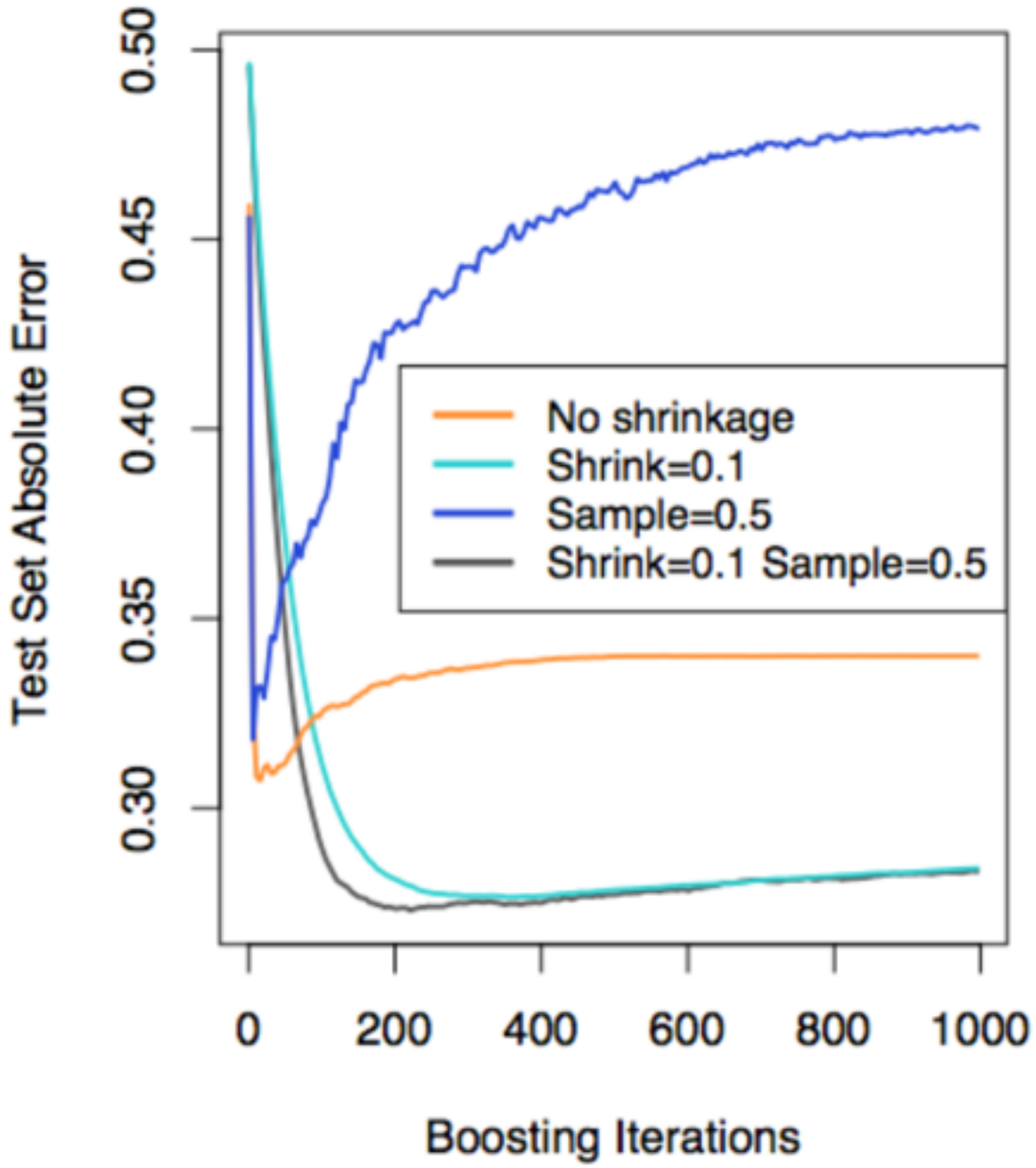
1. call the `fit` step on the `model`
2. call the `predict` step on the `model`
3. record the score (which you'll be able to access later)

Note that in the last slides we didn't tune `n_estimators`...

This is because `n_estimators` and the `learning_rate` work opposite each other

- As the `learning_rate` decreases, all else being equal, `n_estimators` has to increase (a smaller `learning_rate` means you're descending more slowly along the error curve)
- Typically, it's a good idea to fix one or the other and only tune one (it's common to try to optimize the `learning_rate` to be low and simply use lots of trees)
    - ★ Be careful to try to avoid overfitting - too large of a `learning_rate` can lead to overfitting with even a small number of trees

- **Takeaway**: Trying out combinations of hyper-parameters is critical in terms of finding a good set



Regression with Square Loss

Let us summarize the algorithm we just derived using the concept of gradients. Negative gradient:

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

start with an initial model, say, $F(x) = \frac{\sum_{i=1}^{n} y_i}{n}$
iterate until converge:

- calculate negative gradients $-g(x_i)$
- fit a regression tree h to negative gradients $-g(x_i)$
- $F := F + \rho h$, where $\rho = 1$

The benefit of formulating this algorithm using gradients is that it allows us to consider other loss functions and derive the corresponding algorithms in the same way.

Boosting Algorithm for Regression Trees

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

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

   - 2.3 Update the residuals,

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

3. Output the boosted model,

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

Simplified version. Easier to understand for building intuition.

Details on GBRT, see page 361
http://web.stanford.edu/~hastie/local.ftp/Springer/OLD/ESLII_print4.pdf

AdaBoost is not black magic

- But there is quite a bit of math and underpinning concepts to go through to really understand it.
  http://web.stanford.edu/~hastie/local.ftp/Springer/OLD/ESLII_print4.pdf - Page 341-346

Very very roughly,

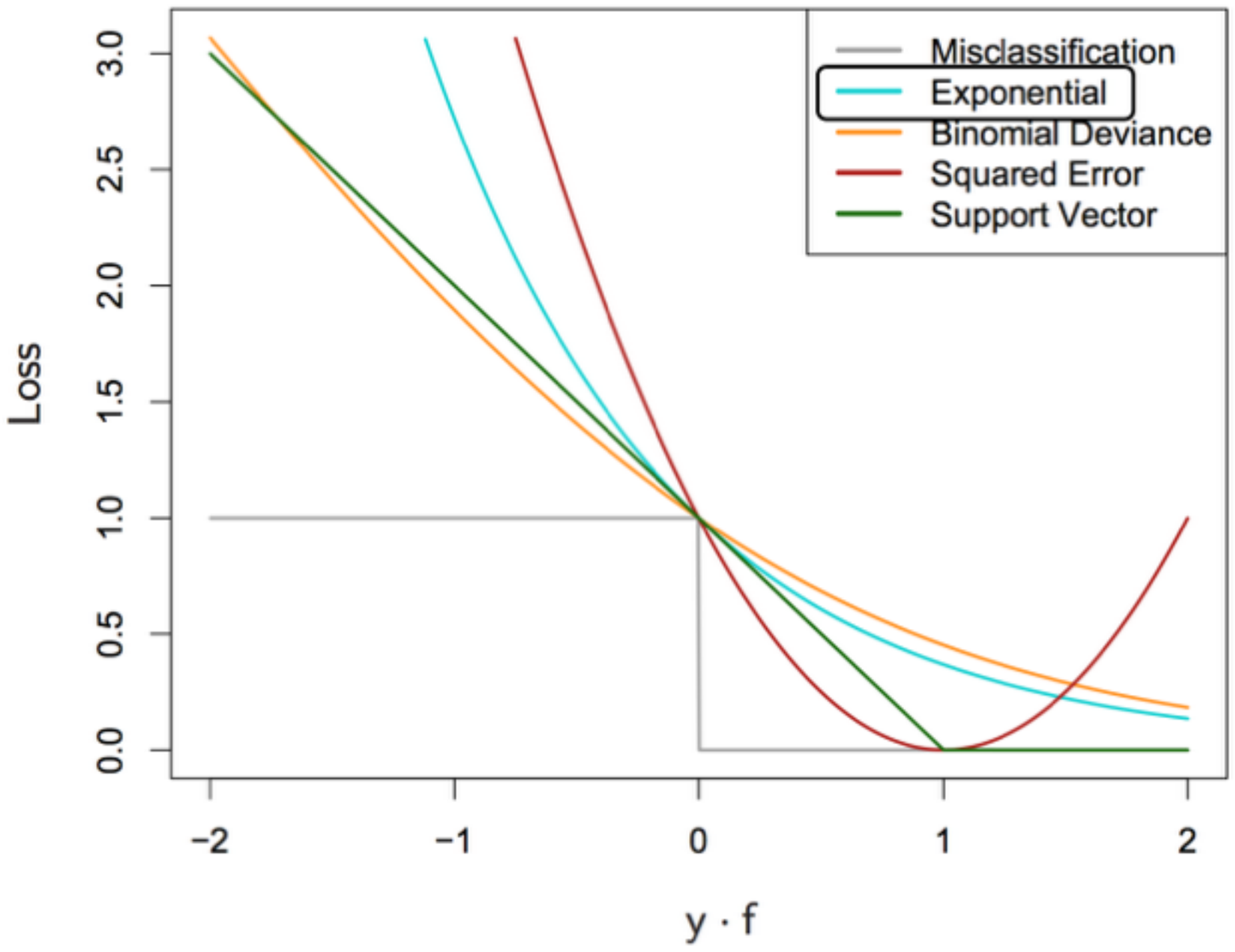
- It is a version of Forward Stagewise Additive Modeling, which adds new basis functions without adjusting previous parameters and coefficients.
  - This contrasts with gradient boosting, see pg 342 vs. pg 361 in link above
  - AdaBoost uses Exponential Loss $L(y, f(x)) = \exp(-y\, f(x))$.
    - It can be shown that to minimize this loss, at each iteration, we can reweight our observations $w_i^{(m+1)} = w_i^{(m)} \cdot e^{\alpha_m I(y_i \neq G_m(x_i))} \cdot e^{-\beta_m}$
  - Use exponential loss because of computational advantage; could consider others.
- Can be shown that the additive expansion in AdaBoost is estimating

$$f^*(x) = \arg\min_{f(x)} \mathrm{E}_{Y|x}(e^{-Yf(x)}) = \frac{1}{2}\log\frac{\Pr(Y=1|x)}{\Pr(Y=-1|x)}$$

which justifies taking the sign as classification rule for final classifier

$$G(x) = \operatorname{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$$

Comparison of Loss Functions for Classification



Comparison of Loss Functions for Regression