Boosting

Natalie Hunt
Lots of credit to Matt Drury

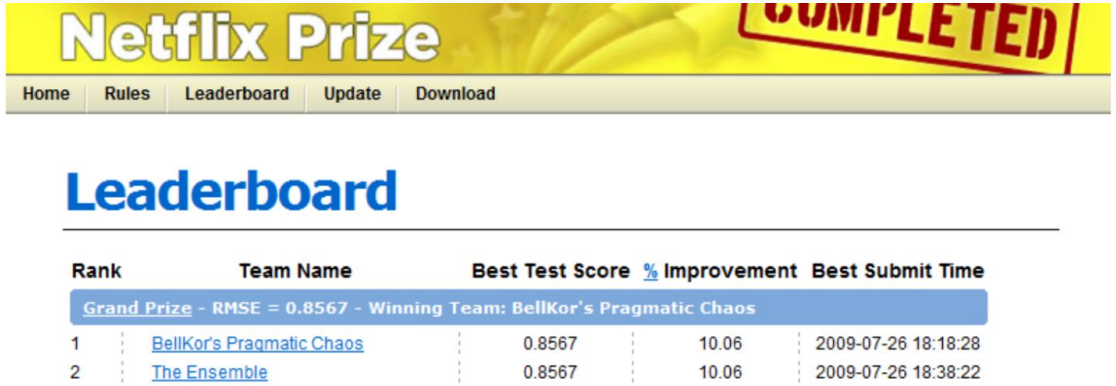
galvanize

Objectives

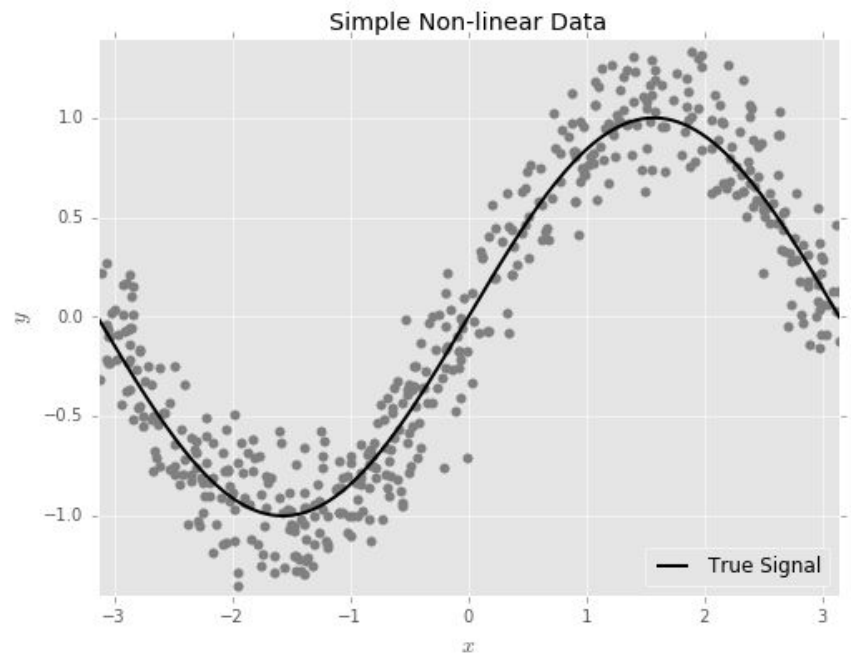
- Get an intuitive understanding of boosting
- Understand the algorithm's hyperparameters
- State the difference between boosting and bagging/RF
- Understand the bias/variance tradeoff of boosting and other ensemble algorithms
- State basic strategies for interpreting a booster

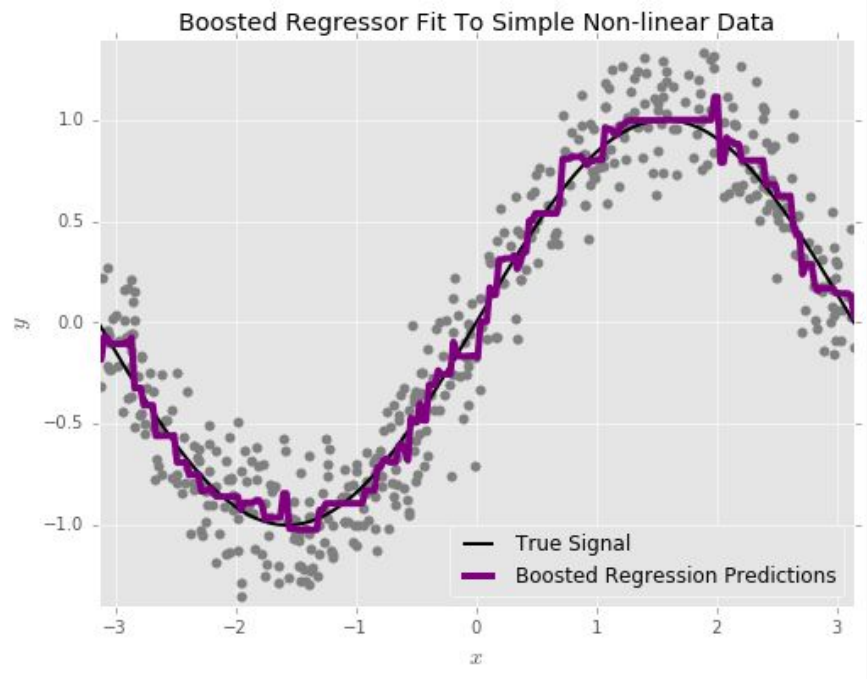
Boosting

- Boosting is a powerful ensemble algorithm
- Boosting can be applied to many machine learning algorithms but is mostly used in combination with decision trees
- Bagging goes after variance
- Boosting goes after bias
- One of the best “out of the box” algorithms
- Often the winning algorithm in ML competitions!

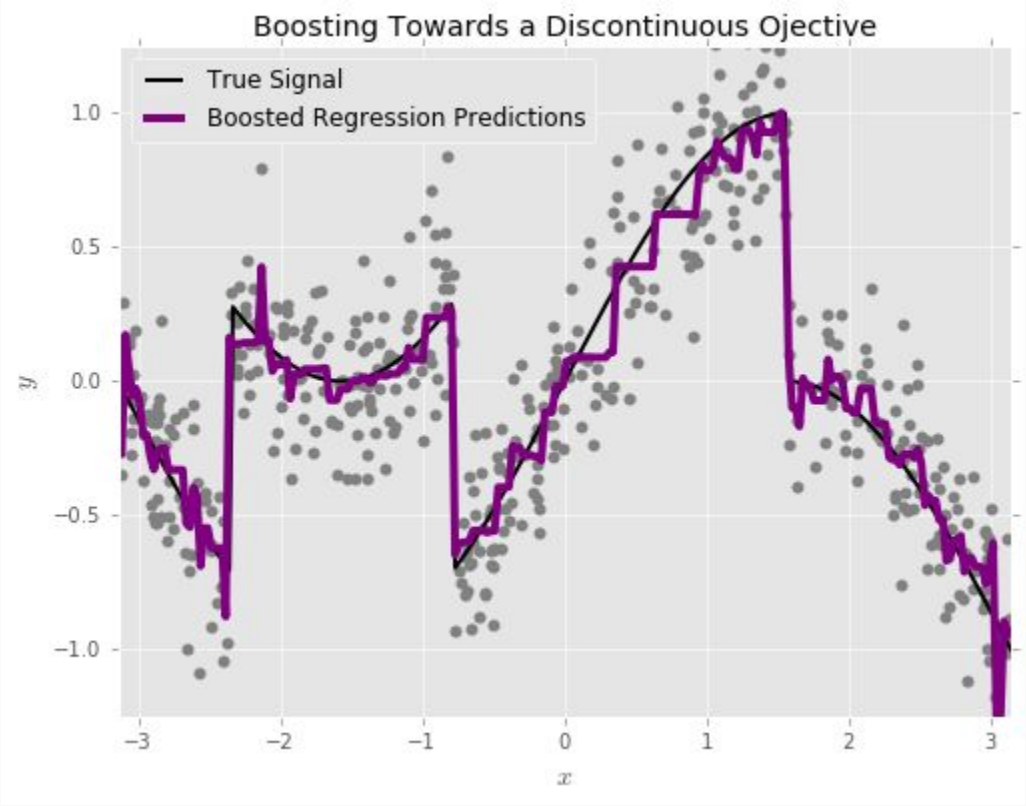


Boosting effortlessly adapts to very non-linear objectives

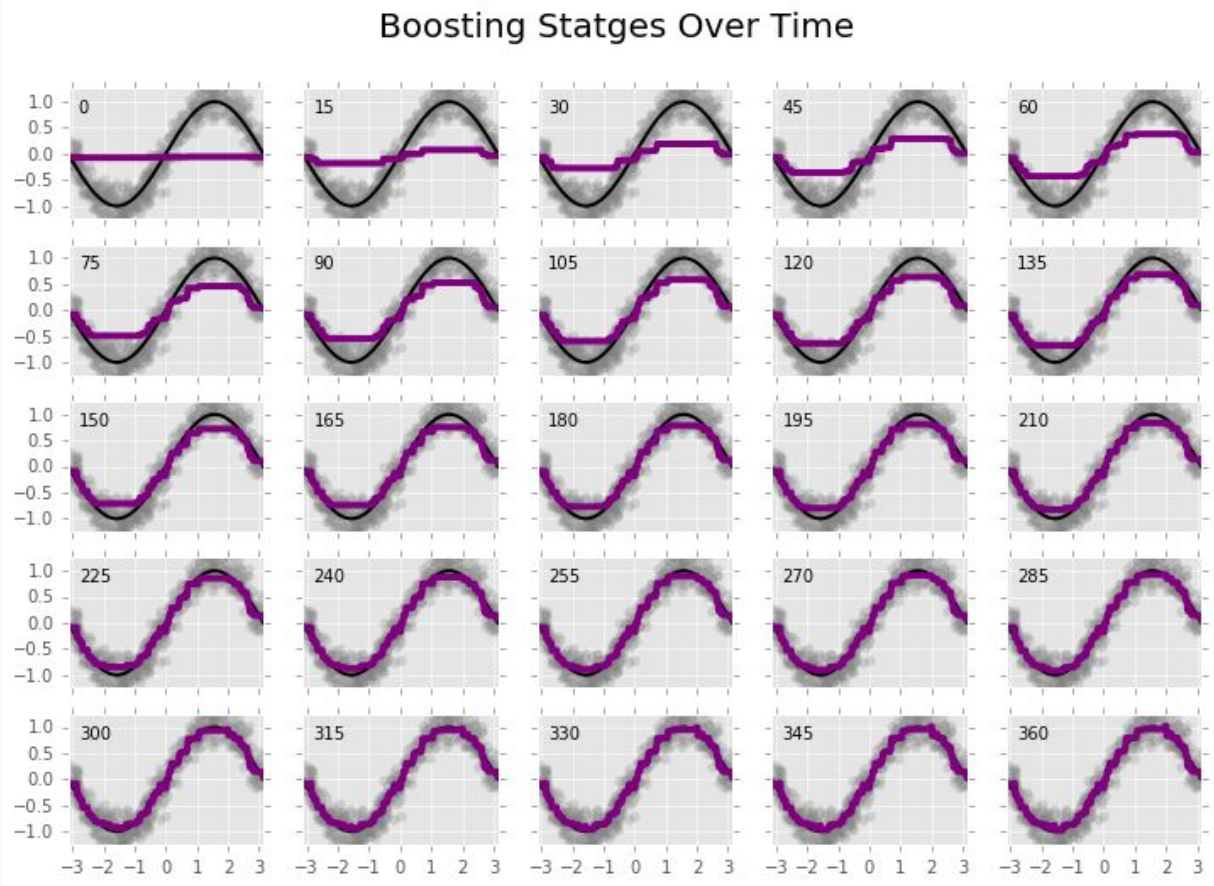




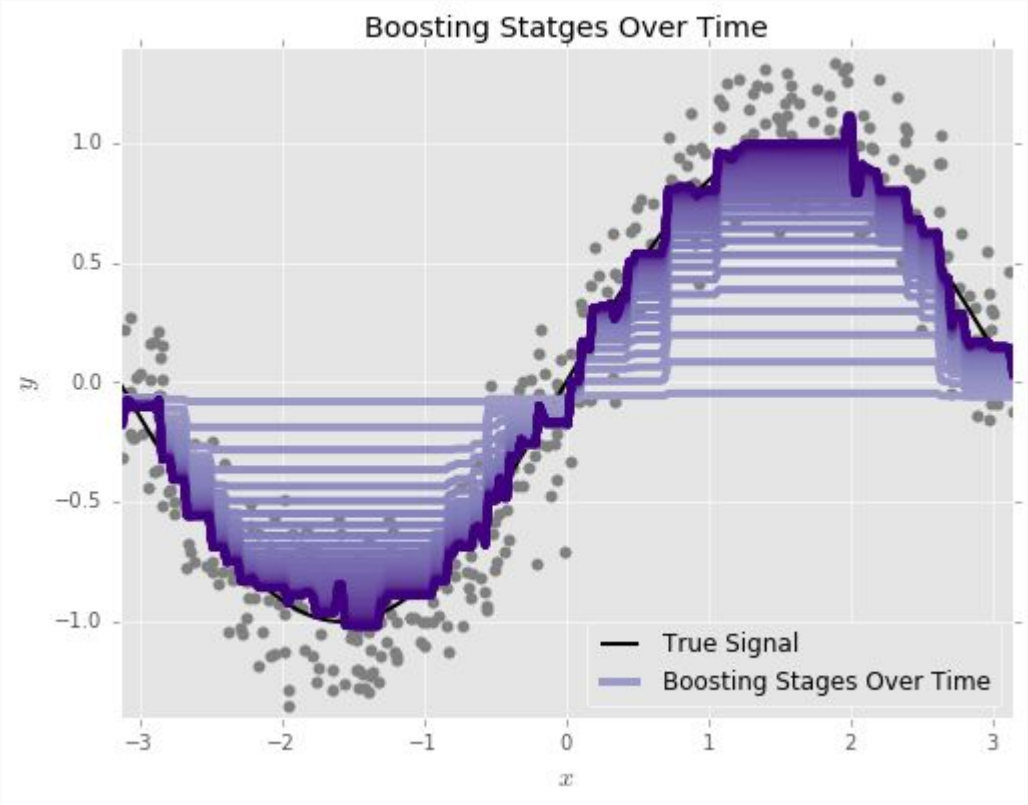
Boosting effortlessly adapts to very non-linear objectives



Boosting grows model gradually



Each stage makes small improvements to previous model



Boosting: Bias and variance

galvanize

- Boosting goes after variance by growing very slowly
- Boosting goes after bias by combining many weak models into a “super model”

- How does a linear model go after variance? **Regularization (Ridge/Lasso)**
- How does a linear model go after bias? **More features/interactions**

- How does Random Forest go after variance? **Averaging many trees**
- How does RF go after bias? **Grow larger trees (= interactions)**

Boosting: The setup

- We have a training set $\{x_i, y_i\}$
- N sample size
- M number of features
- Goal: Construct f(x) so that $f(x_i) \approx y_i$ for all i
- In other words, choose f(x) so that $\sum(f(x_i) - y_i)^2$ is small

f(x) will be the sum of many smaller models (“weak learners”)

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

Grown sequentially:

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

Where to start?

Simplest choice: $f_0(x)$ = constant

What constant?

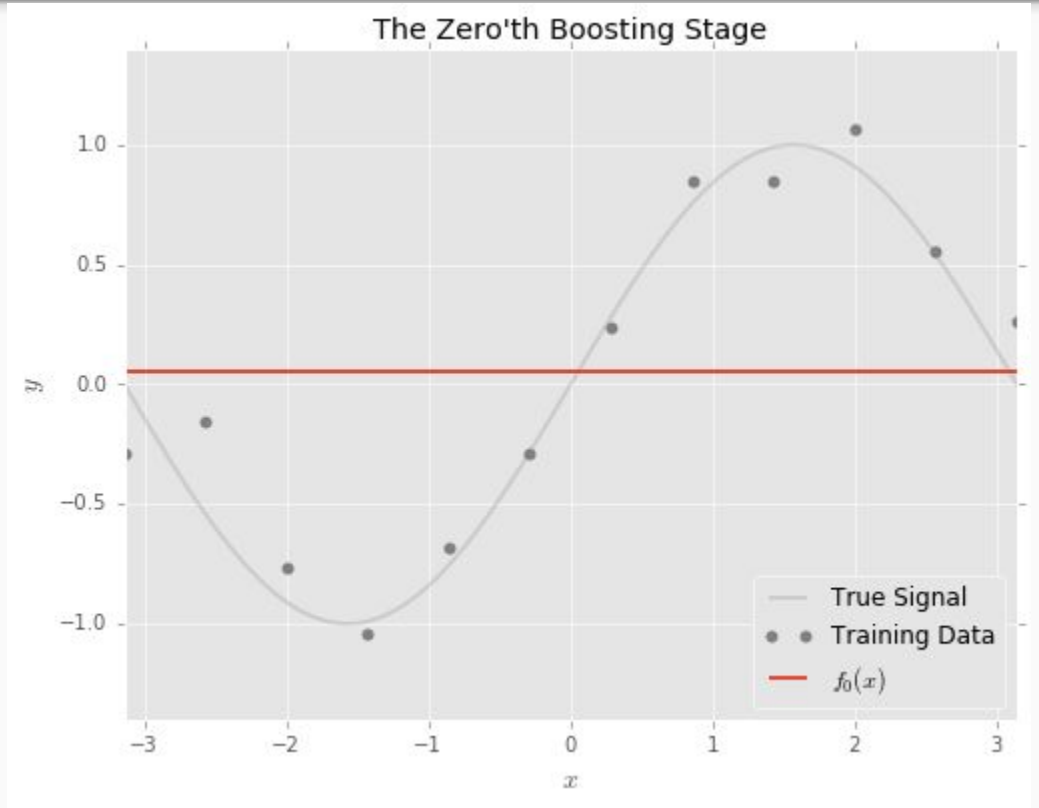
We want to minimize $\Sigma (y_i - \text{constant})^2$

Which constant minimizes this?

$f_0(x) = 1/N * \Sigma y_i$ (the mean)

Next step: finding $f_1(x)$

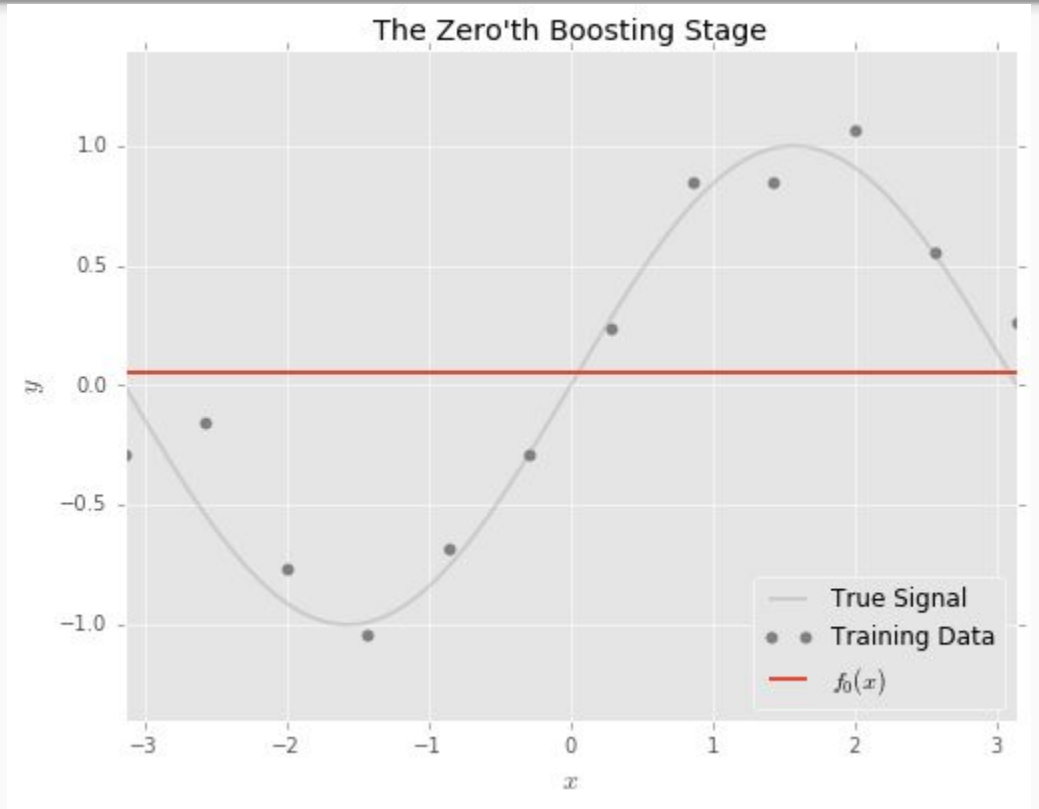
We are now in the following situation



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

Next step: finding $f_1(x)$

Intuitively, what do we have to do to the red line to fit the data better?

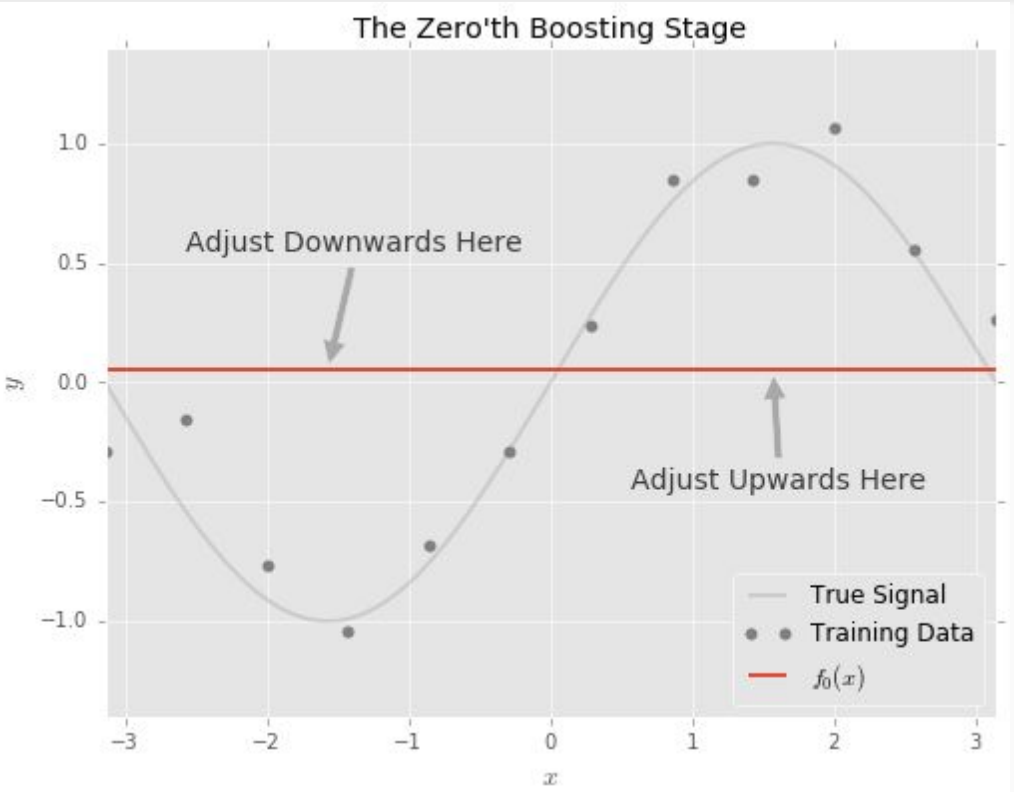


Next step: finding $f_1(x)$

Intuitively, what do we have to do to the red line to fit the data better?

Let's think about this for a moment

How is the data telling us to do that?



Look at the residuals!

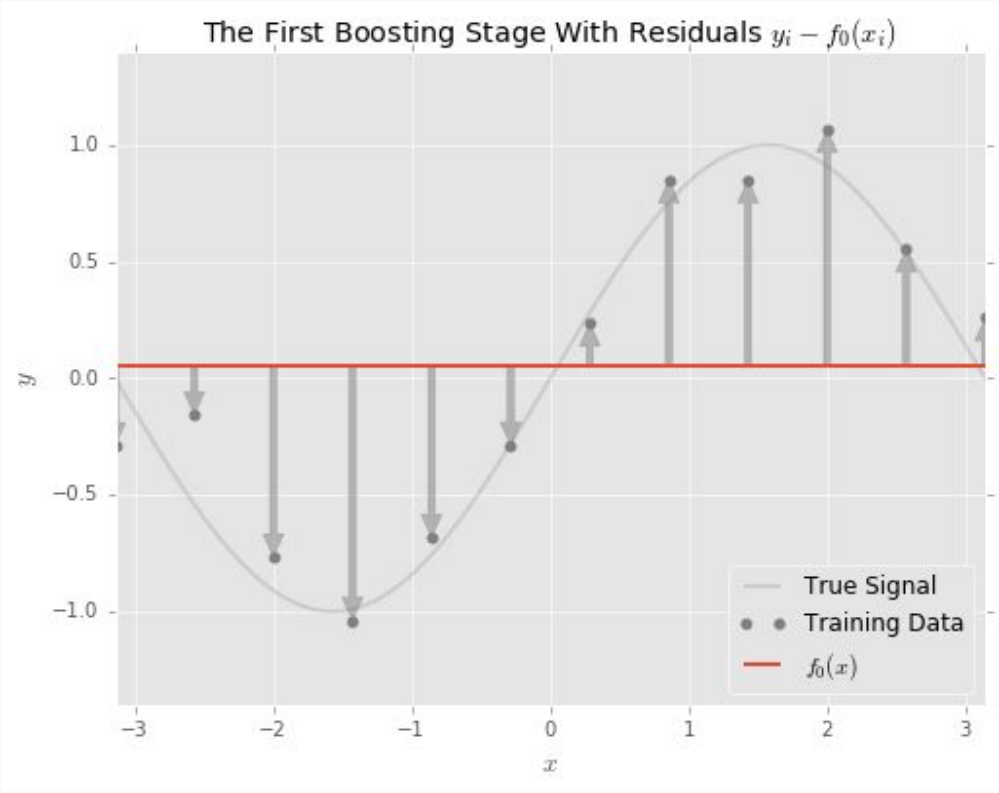
The residuals tell us what to do!

$e_i = f_0(x_i) - y_i$

We should adjust the f(x) towards the residuals

How?

Fit a model to the residuals!



Fitting to residuals

- Residuals treated as new dataset
- New response values are:

$$y_{i,new} = y_i - f_0(x_i)$$

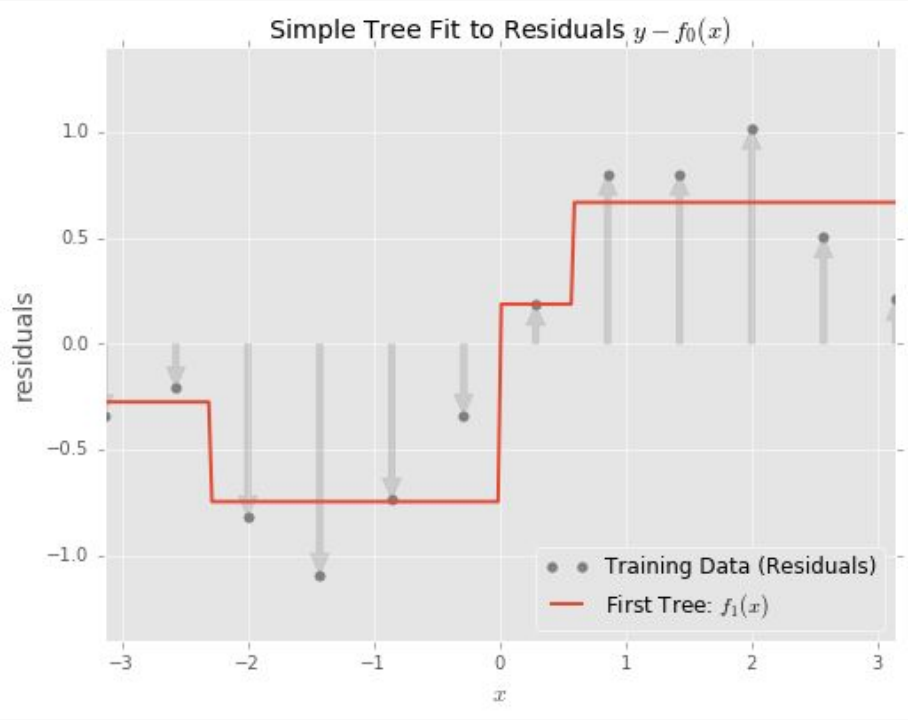
Fitting to residuals

Here a simple model fit to this new working dataset (the residuals after step 0)

We can update our main model:

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

Let's do one more step!



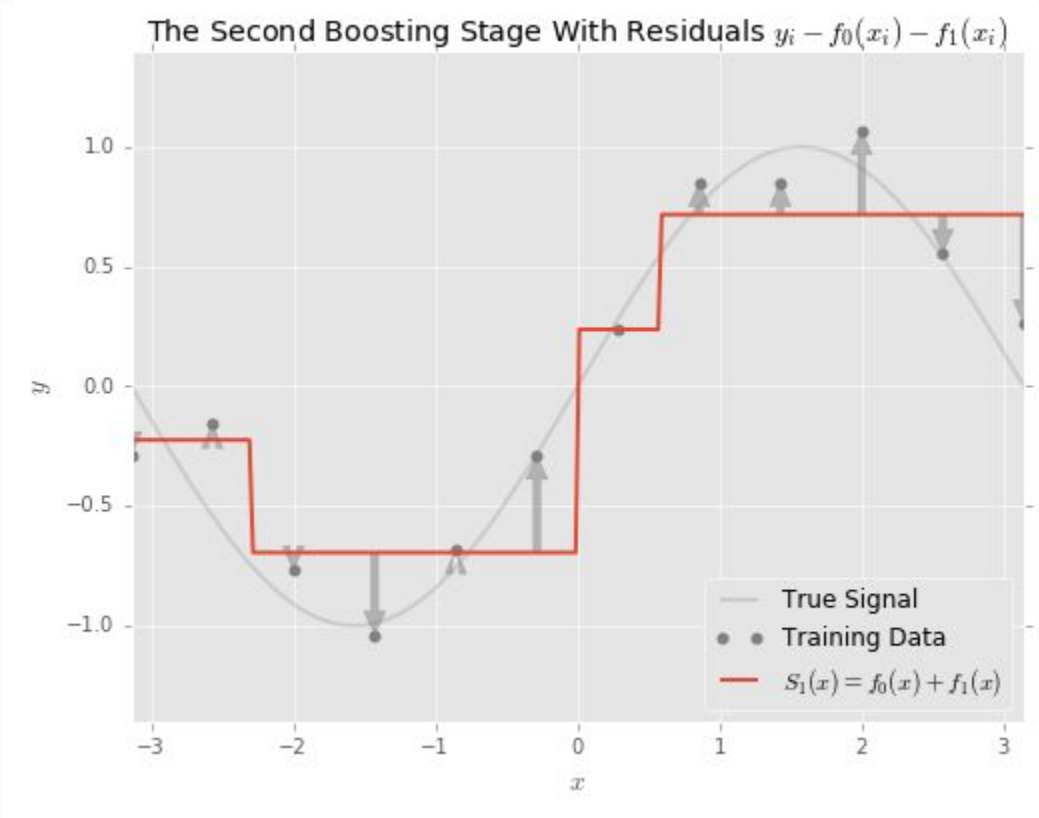
Fitting to residuals (again!)

Get residuals after second step:

$e_i = y_i - S_1(x_i) = yi - (f_0(x_i) + f_1(x_i))$

Set $y_{i,new} = e_i$

Fit new model $f_2(x)$



Fitting to residuals (again!)

Get residuals after second step:

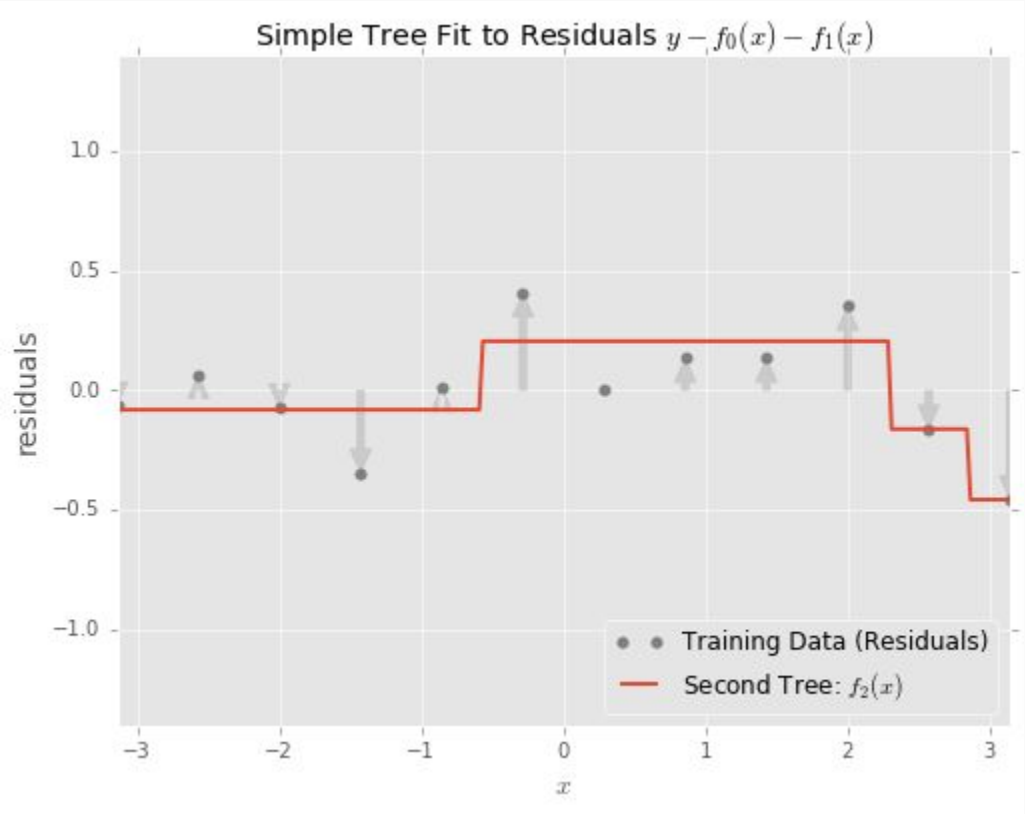
$e_i = y_i - S_1(x_i) = yi - f_0(x_i) - f_1(x_i)$

Set $y_{i,new} = e_i$

Fit new model $f_2(x)$

Update the overall model to:

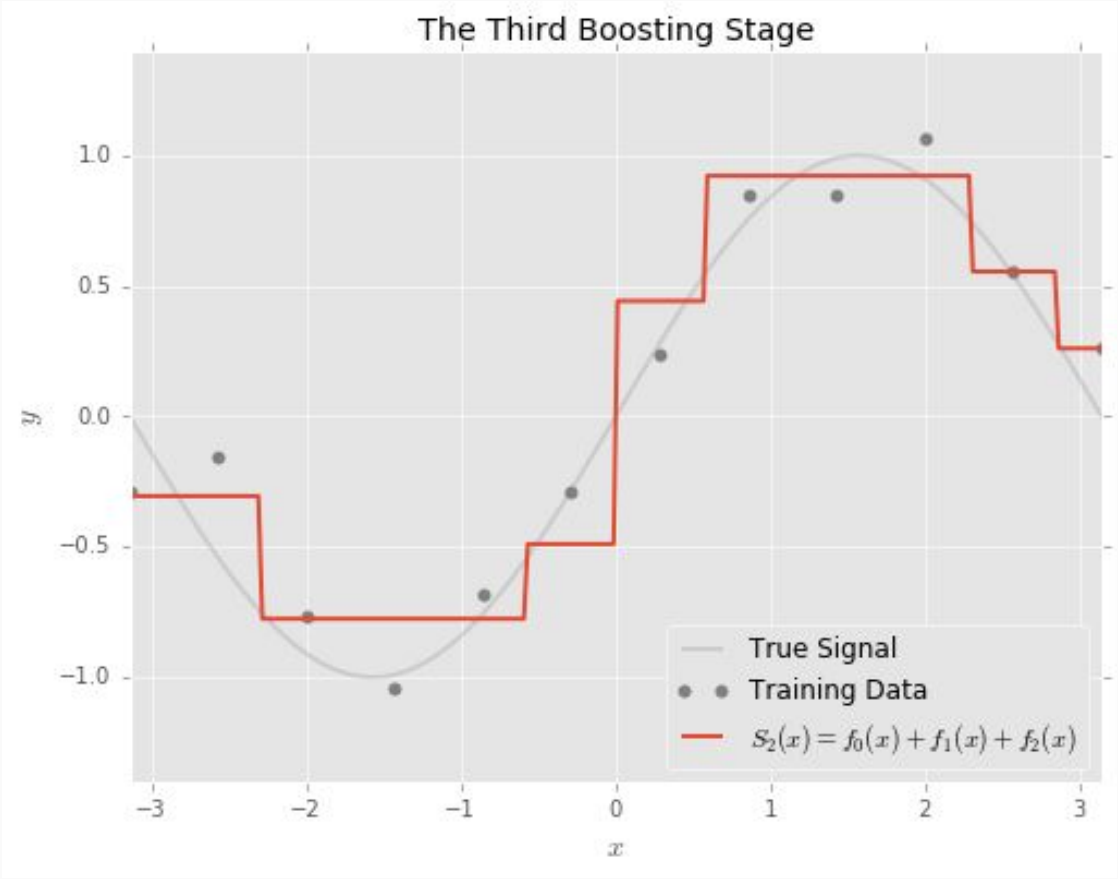
$S_2(x) = f_0(x) + f_1(x) + f_2(x)$



Update the overall model (again!)

Update the overall model to:

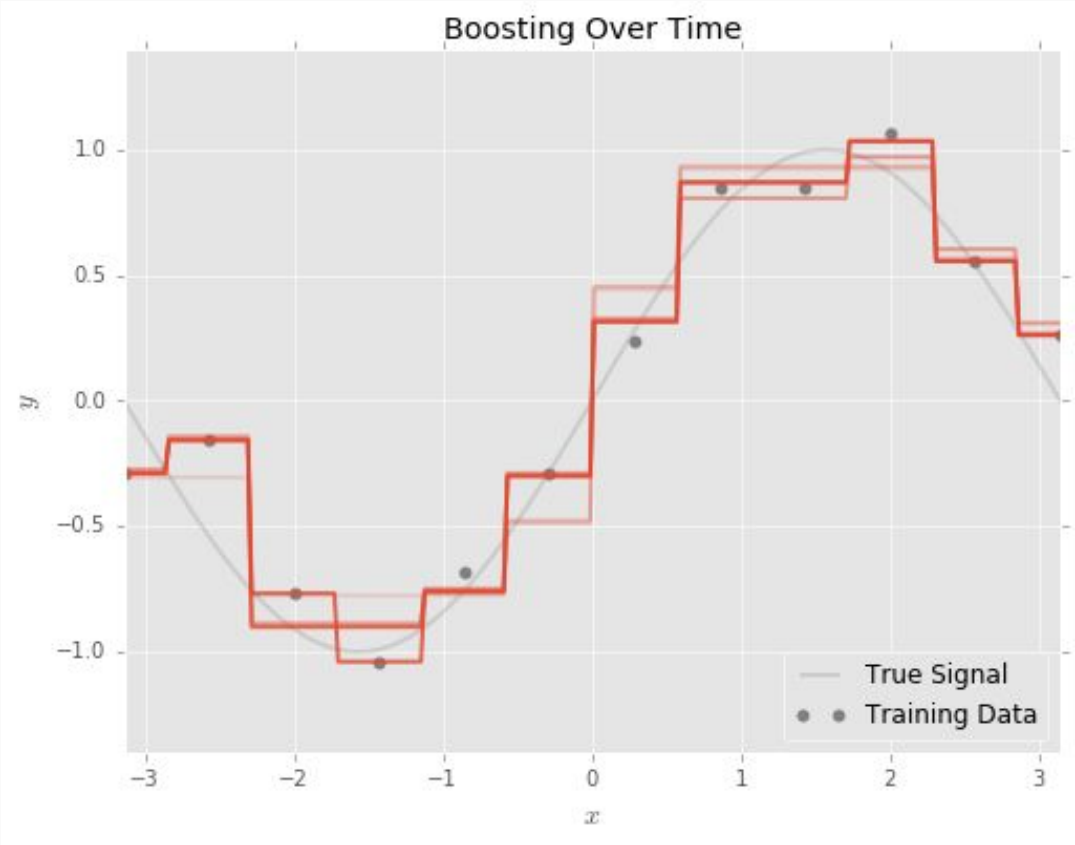
$S_2(x) = f_0(x) + f_1(x) + f_2(x)$



Overall model over time

galvanize

Overall model improves over time



Growing slowly

What happened to growing slowly?

Instead of fitting “hard” to the residuals every time, we can add a softening factor λ

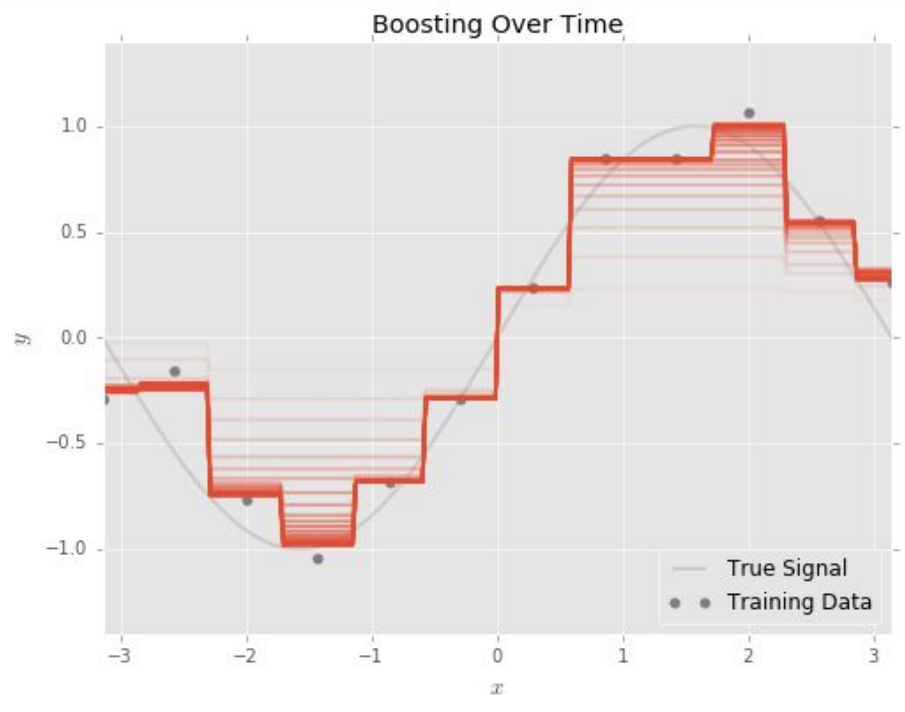
$$S_{k+1}(x) = S_k(x) + f_{k+1}(x)$$

becomes

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

with λ << 1

λ = 0.01



Smaller λ make model train more slowly but reduce overfitting!

Boosting algorithm overview

Input: Initial Dataset $\{x_i, y_i\}$

Output: function $S(x)$ such that $S(x_i) \approx y_i$

Algorithm:

1) Initialize $S_0(x) = f_0(x) = 1/N * \Sigma y_i$
2) Iterate for k = 1, …, M:
   a) Create working data set $W_k = \{x_i, y_i - S_{k-1}(x_i)\}$
   b) Fit a model f_k (usually regression tree) to W_k minimizing some loss function (e.g. least squares)
   c) Set $S_k(x) = S_{k-1}(x) + \lambda f_k(x)$
3) Return final $S_M(x)$

Note: the loss function and the way to calculate the (pseudo-)residuals, depends on the exact boosting algorithm; how λ is set as well

The name “gradient” boosting comes from the idea to adjust the model towards the gradient of a loss function

In our previous case with the residuals, this looks like this:

$$S_{k+1}(x_i) = S_k(x_i) + \lambda(y_i - S_k(x_i))$$

$$\Downarrow$$

$$\xi_{k+1} = \xi_k + \lambda(y - \xi_k)$$

This corresponds to the gradient of the squared error loss function: $\frac{1}{2}*(y - S_k(x))^2$

Looks a lot like gradient descent!

Review: Gradient descent

galvanize

Objective: minimize some function L(x)

Input: L(x)

Output: point x_{opt} that optimizes L(x)

1) Compute derivative/gradient $\nabla L(x)$
2) Initialize x_0 arbitrarily
3) Iterate until convergence:
    a) $x_{i+1} = x_i - \lambda \nabla L(x_i)$ with step size λ

$\nabla L(x_i)$ will give us the direction and size of the step depending on current slope, λ adjusts the step size

In our case $\nabla L(x) = y - S_k(x)$ = residuals, tell us direction and size of step, adjusted by λ

Gradient boosting the squared error loss

In our case $L(x) = \frac{1}{2}*(y - S_k(x))^2$, the squared error loss

More formally, gradient descent on the squared error loss:

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

$$-\nabla_\xi L(\xi, y) = -\frac{1}{2}\frac{\partial}{\partial \xi}(y - \xi)^2 = y - \xi$$

$$\xi_{k+1} = \xi_k - \lambda \nabla_\xi L(\xi_k, y) = \xi_k + \lambda(y - \xi_k)$$

In other words: least squares gradient descent iteratively moved in the direction of the residuals!

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

$$\xi_{k+1} = \xi_k - \lambda \nabla_\xi L(\xi_k, y) = \xi_k + \lambda(y - \xi_k)$$

In other words: least squares gradient descent iteratively moved in the direction of the residuals!

This is why we called it gradient boosted regression

We “boost” the current estimate by adding an approximation to the gradient of the squared error loss function

Practical Gradient Boosting

Gradient boosting in scikit-learn

galvanize

Boosting algorithms are in scikit-learn's ensemble module

```
from sklearn.ensemble import GradientBoostingRegressor

model = GradientBoostingRegressor()

model.fit(X,y)

predictions = model.predict(X_new)
```

Gradient boosting in scikit-learn

galvanize

We can access the different stages of the boosting model through the staged_predict iterator:

```
for preds in model.staged_predict(X_new):

    #do what you would like with the different stages
```

Gradient boosting in scikit-learn: Hyperparameters

galvanize

Knobs to turn:

```
model = GradientBoostingRegressor(loss='ls',
                                  n_estimators=100,
                                  learning_rate=0.01,
                                  max_depth=3,
                                  subsample=1.0,
                                  min_samples_split=2,
                                  min_samples_leaf=1,
                                  min_weight_fraction_leaf=0.0,
                                  ...
                                  )
```

Gradient boosting in scikit-learn: Hyperparameters

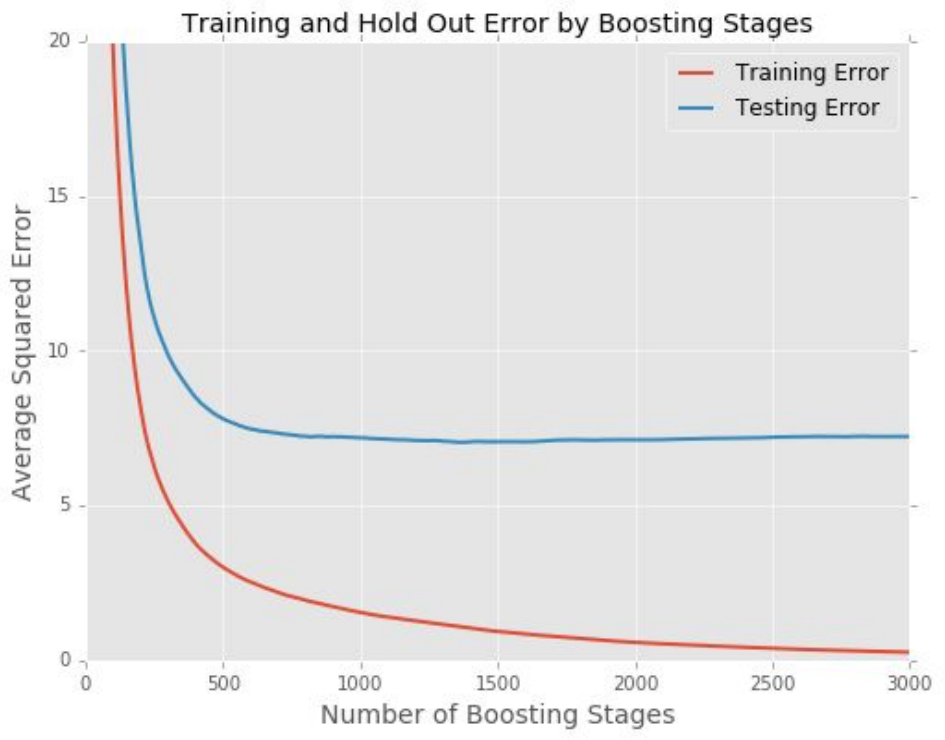
galvanize

Important knobs:

- `loss` specifies the loss function to minimize at each step. '*ls*' corresponds to least squares which is what we have seen so far
- `n_estimators` is the number of boosting stages
- `learning_rate` is the learning rate for the gradient update (step size)
- max_depth controls the depth of each tree. If `max_depth` > 1, we are including interactions between features. Frequently people only grow stumps (`max_depth` = 1)
- `sub_sample` allows us to grow to a random subset of the data at each step, similarly to bagging/RF and stochastic gradient descent (more on that later)
- We will now look at how each of these influences our model

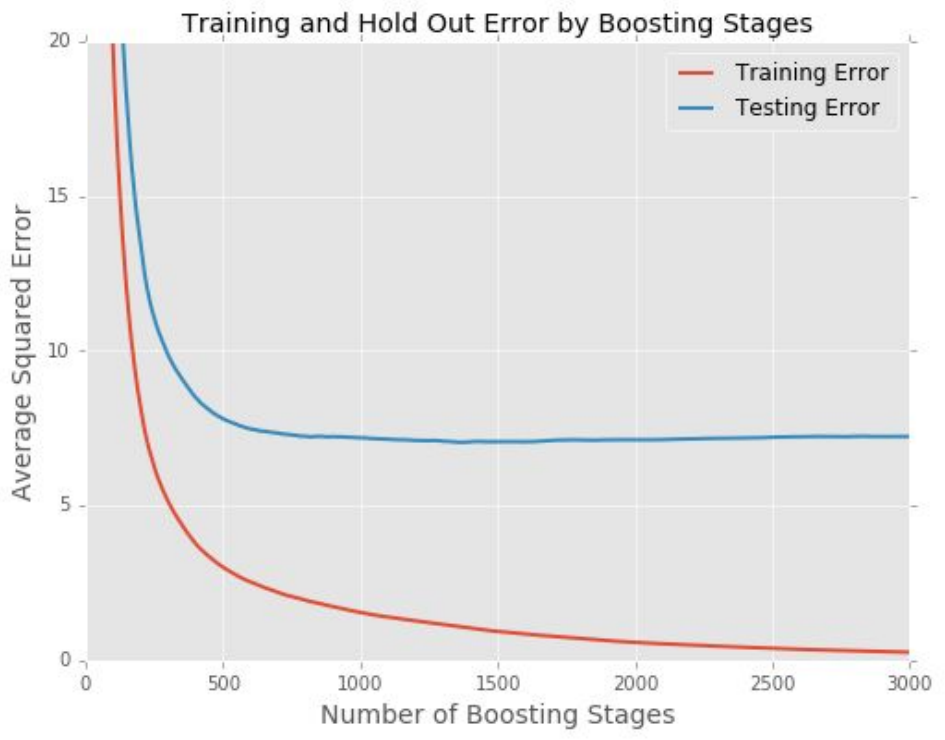
Hyperparameters: number of estimators

As `n_estimators` increases, training error will decrease monotonically, test error not necessarily!



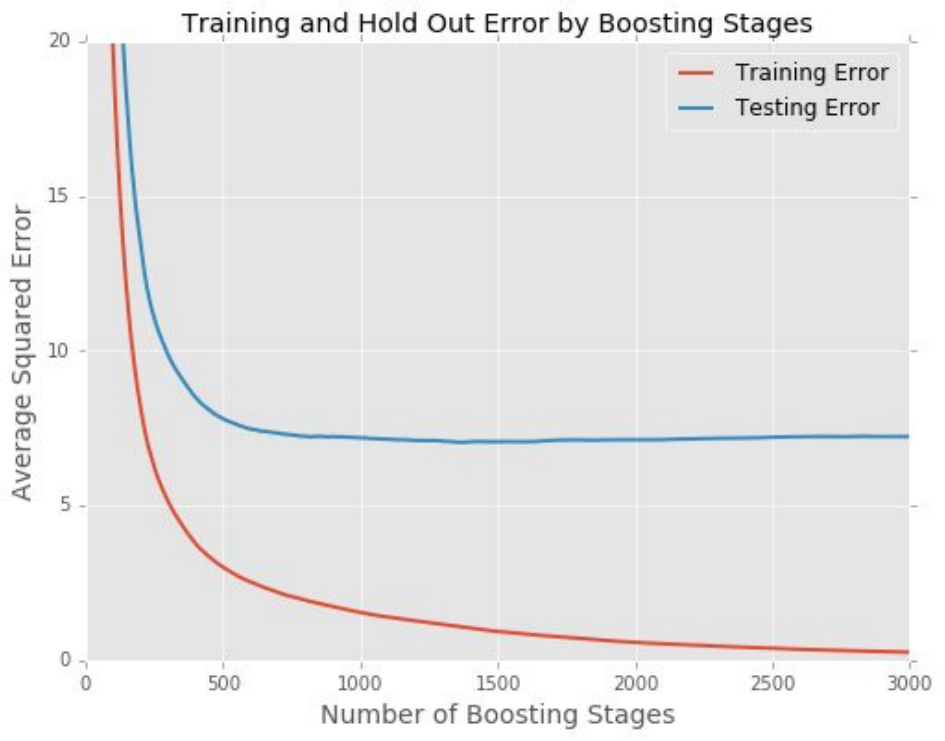
Hyperparameters: number of estimators

Can the training errors ever become *exactly* zero?



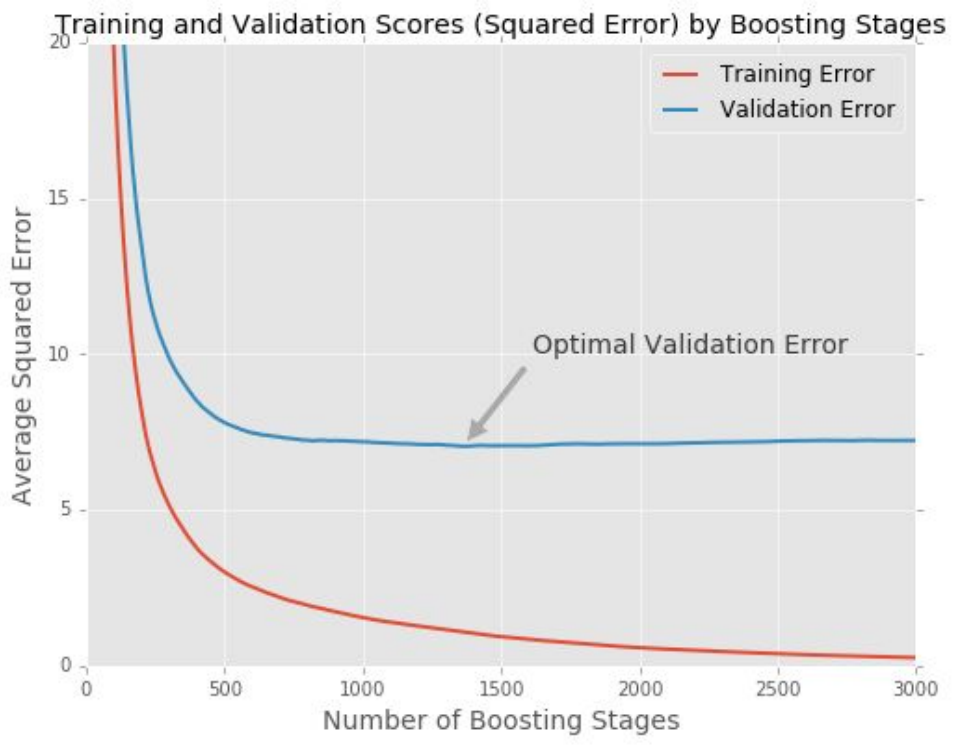
Hyperparameters: number of estimators

Even with a small learning rate, we can potentially overfit to our training set if we grow too many trees (but hard to do with very small learning rate)



Hyperparameters: number of estimators

We want to use a validation set to choose the best number of stages!



Hyperparameters: number of estimators

staged_predict together with the .loss_() function will help

```
def get_optimal_n_estimators(model, X_new, y_new):
    validation_loss = np.zeros(
        model.get_params('n_estimators'))
    for i, preds in enumerate(model.staged_predict(X_new)):
        validation_loss[i] = model.loss_(preds, y_new)
    optimal_tree = np.argmin(validation_loss)
    optimal_loss = validation_loss[optimal_tree]
    return optimal_tree, optimal_loss
```

We can find the optimal n_estimators this way

Hyperparameters: number of estimators

galvanize

Knowing the optimal n_estimators, we can truncate our model

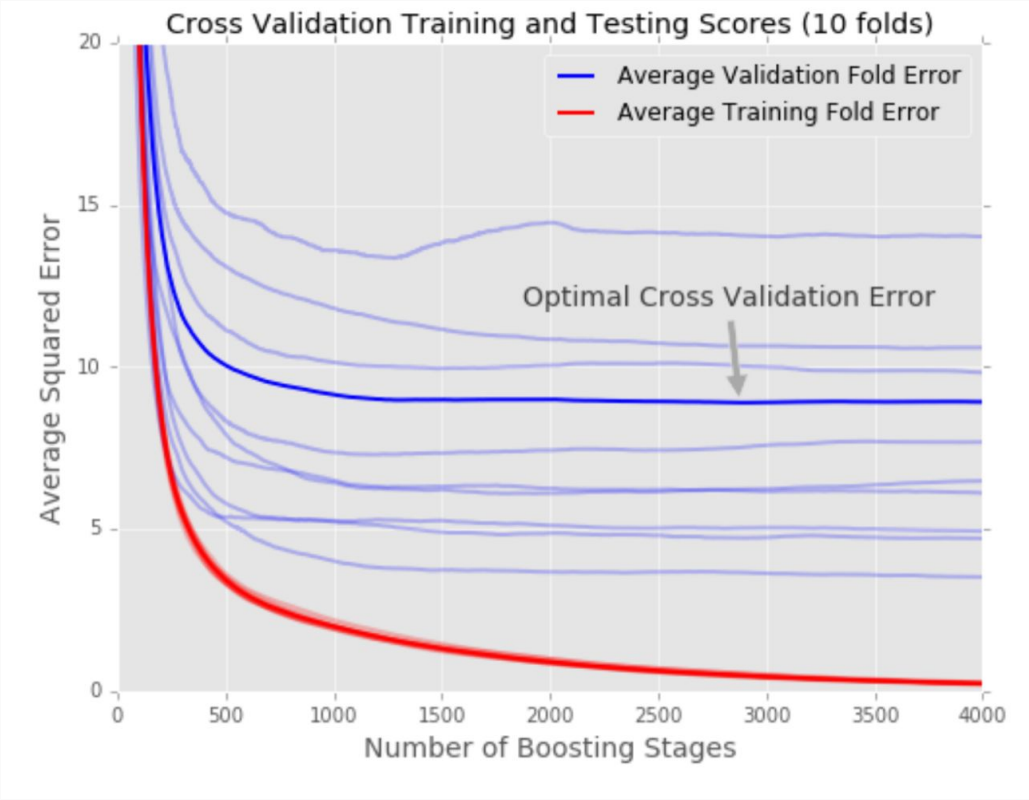
```
from copy import deepcopy

def truncate_boosted_model(model, n_estimators):
    new_model = deepcopy(model)
    # Two dimensions here for the case of multiple classes
    # in a GradientBoostedClassifier.
    new_model.estimators_ = model.estimators_[:n_estimators,
     :] For all classes
    new_model.n_estimators = n_estimators
    return new_model
```

Truncate at optimal n

We can find the optimal n_estimators this way

Alternative: Cross validation



Bold blue line is average over folds, grey arrow indicates where it is optimized

Tuning the learning rate

The learning rate allows us to grow our boosted model slowly.

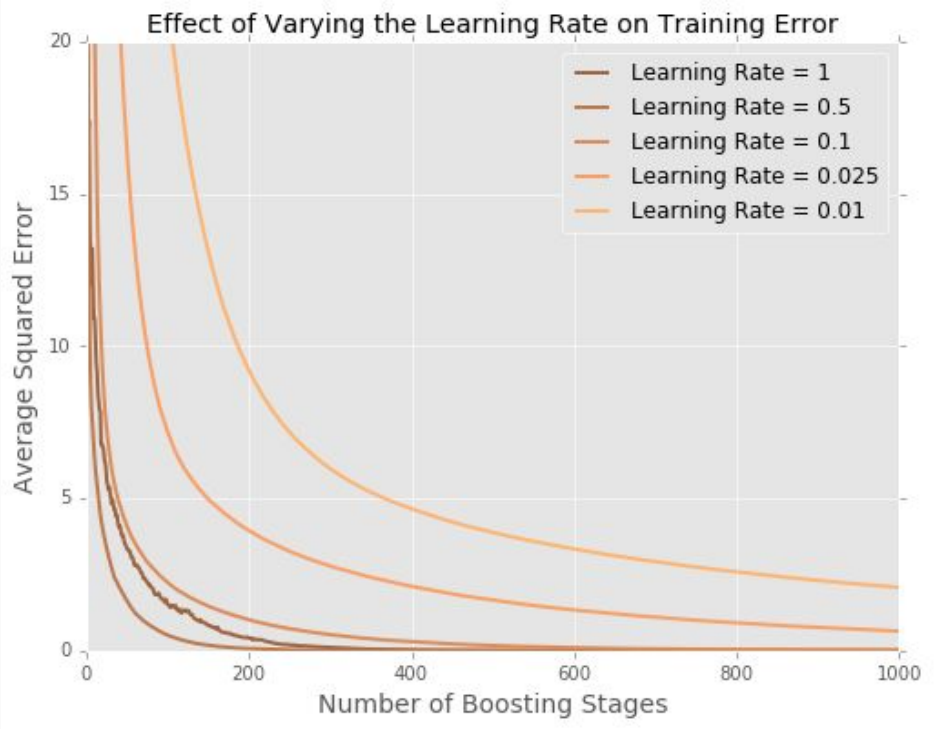
A learning rate of 1 will fit the model hard to the data and lead to high variance

A smaller rate reduces the sensitivity of the model to the training data

Smaller learning rate will reduce how fast the fitting drives down the training error (more n_estimators needed)

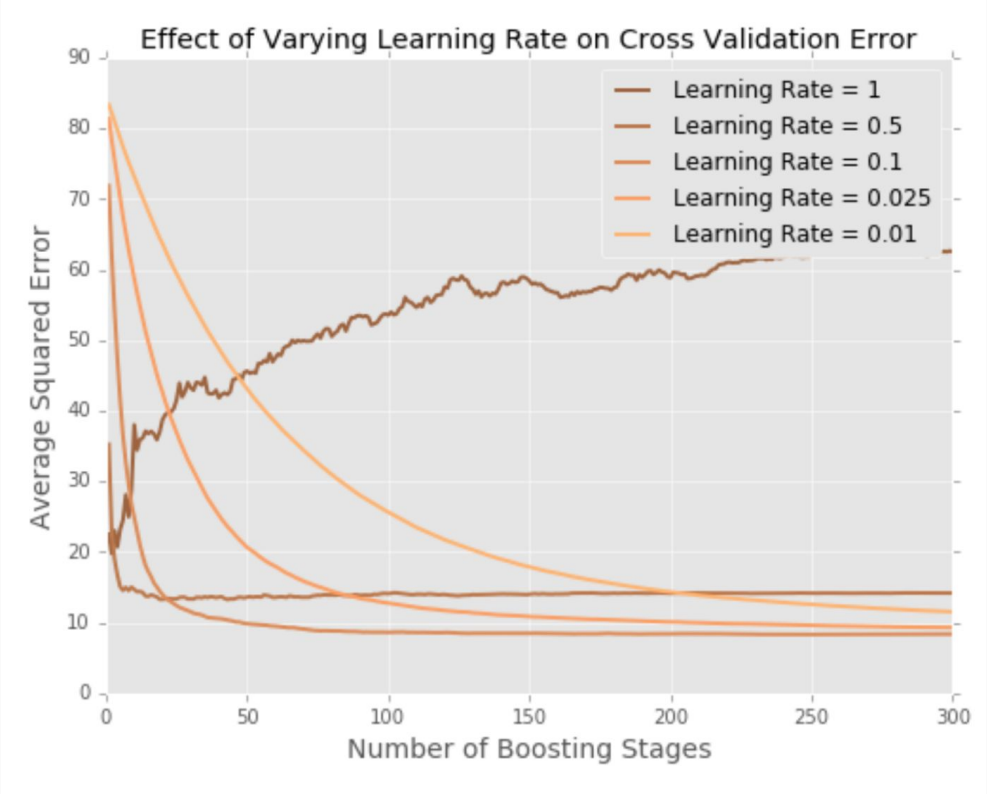
Tuning the learning rate

Smaller learning rate will reduce how fast the fitting drives down the training error



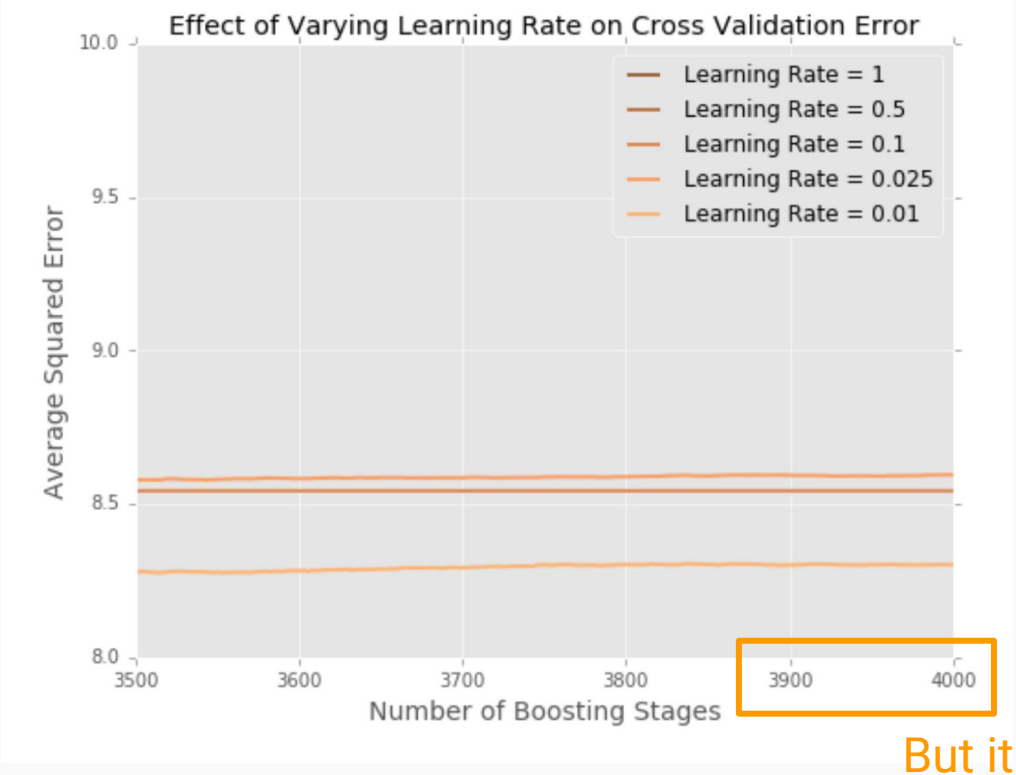
Tuning the learning rate

Smaller learning rate also means more trees are needed to reach the optimal point



Tuning the learning rate

Often a smaller learning rate will eventually yield the better model



But it could take a while!

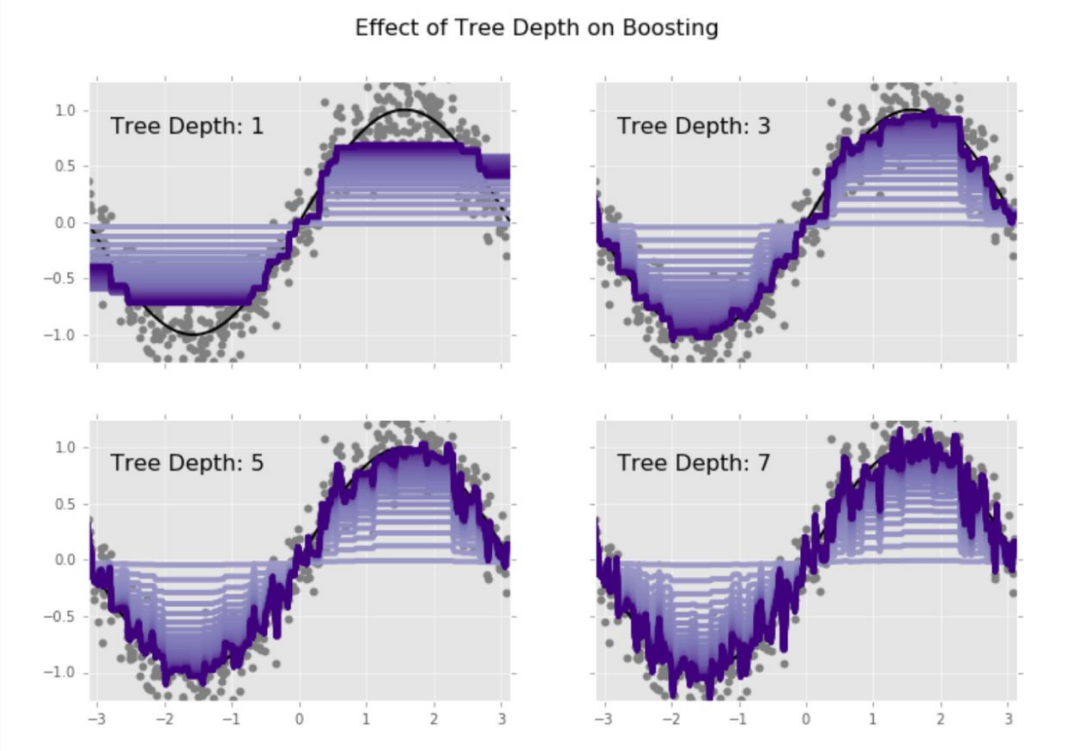
Tuning the learning rate

galvanize

- Initially set the learning rate to something not so small (like 0.1). This will allow you to iterate through ideas quickly
- Later, when tuning with other parameters using grid search, decrease a bit to about 0.01
- For your very final production model, after tuning all the other parameters, set the learning rate to a very small value and run over night, for example 0.001 or even less. Smaller is usually better.
- All your analysis should be done with your earlier models. At the end only run the final model over night for final statistics and plots to present.

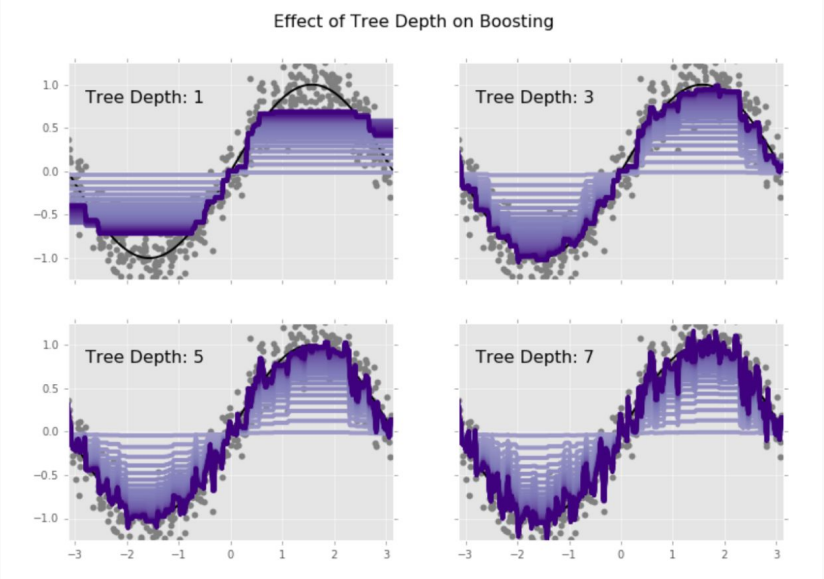
Tuning tree depth

- Larger tree depths mean deeper interactions between predictors are captured
- Lower bias, higher variance!



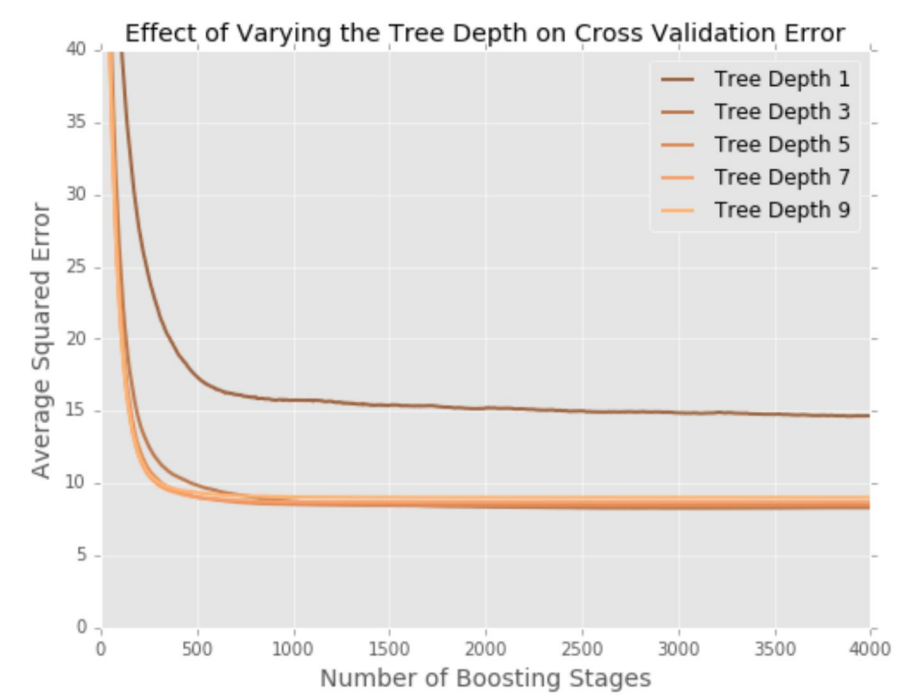
Tuning tree depth

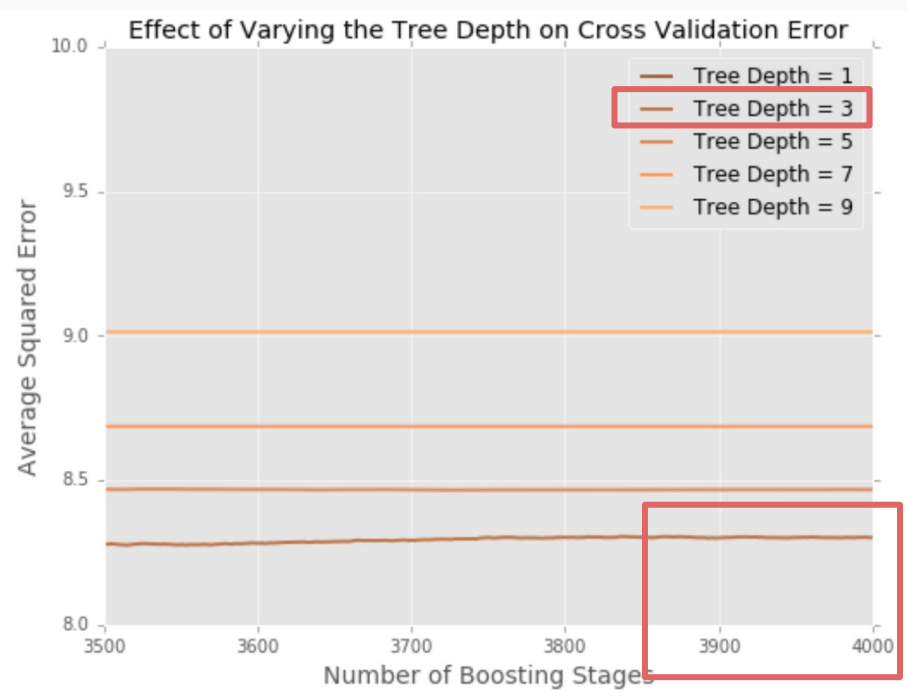
- Grows more complex trees
- As always this is a blessing (lower bias) and curse (higher variance) at the same time
- Somewhat combats the effect of decreasing the learning rate



Tuning tree depth

- It's rarely obvious what the tree depth should be
- A grid search with cross validation can help
- It's actually pretty common to work with stumps only (depth=1, no interactions)





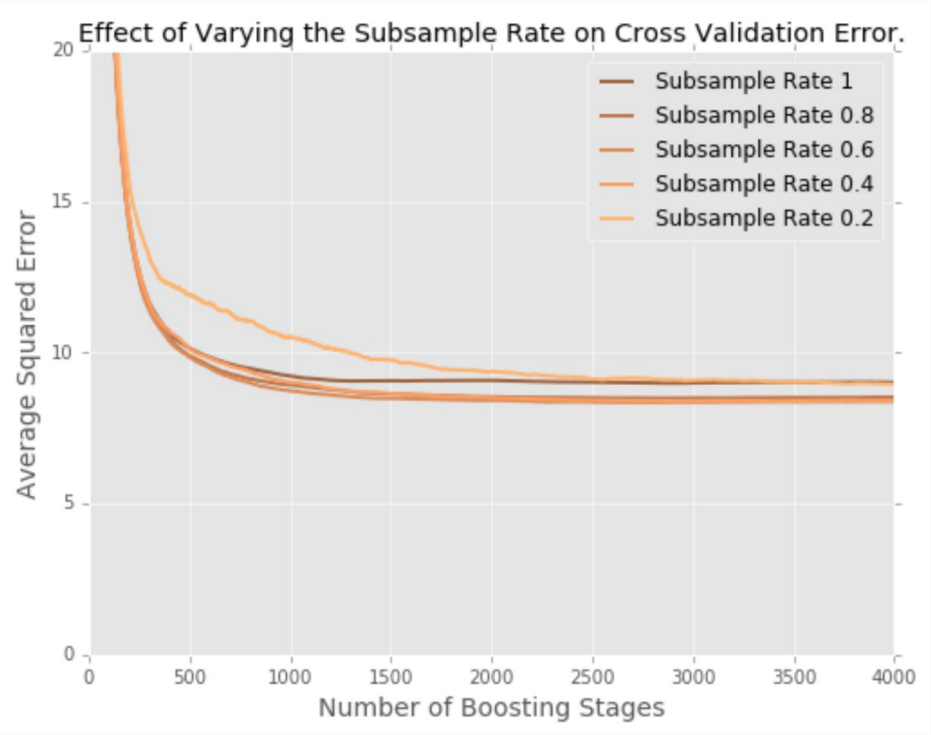
Subsampling/Stochastic Gradient Boosting

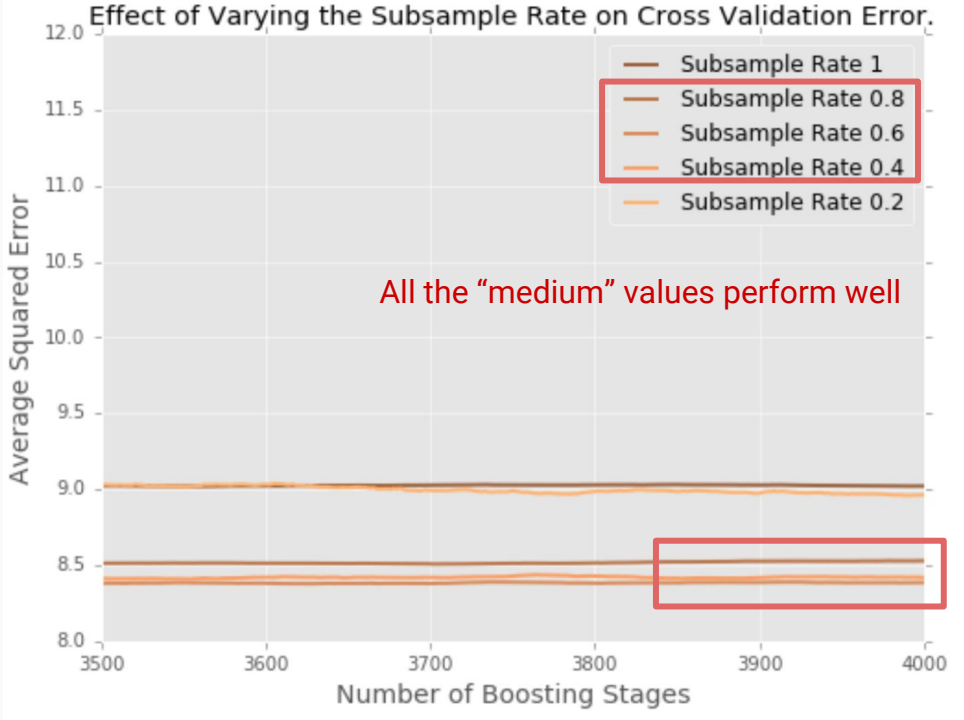
galvanize

- The `subsample` parameter allows us to train our current tree on only a subsample of the entire data
- The principle is similar to random forests, and will also yield lower variance of the final model (but unlike RF and like stochastic gradient descent on subset of data not of features)
- This is because it makes it harder to overfit on the data set because some data is always left out but in a random way so that overfitting to certain data points will average out in the final model
- This simple idea makes boosting even more powerful and one of the most widely used and strongest “out-of-the-box” ML algorithms
- (“out-of-the-box” means no custom code, simply use sklearn)

Subsampling in action

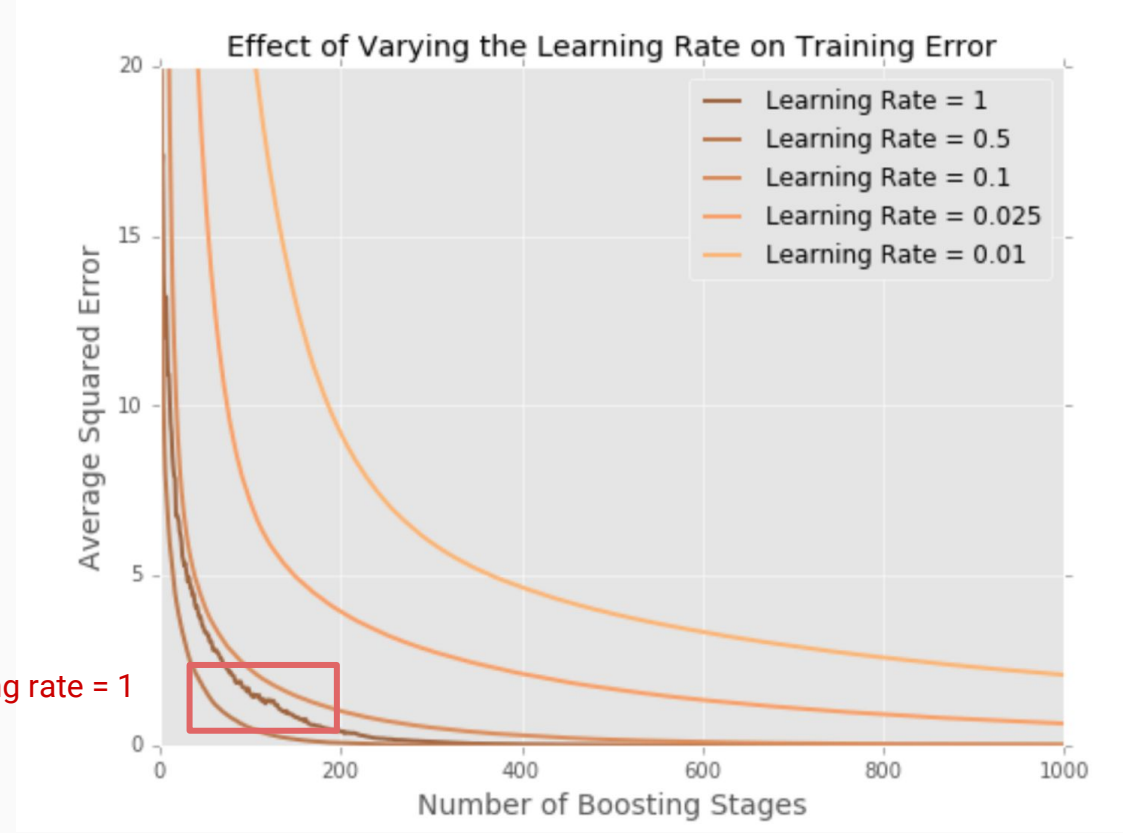
- Choosing a proper subsample size is crucial, too low and too high are bad





Subsampling in action

- Why does the training error sometimes increase in the subsampling setting?



Very obvious for learning rate = 1

Strategy for subsample tuning

- Setting the subsample size to 0.5 works well most of the time
- If you have a massive amount of data and not much time, decrease it (smaller sample means less time fitting!)
- Sklearn default is 1.0, make sure to change it!!!

Less important parameters

Remaining parameters less crucial but can be tuned for your final model with grid search:

- `min_samples_split`: any node with less samples will not be considered for split
- `min_samples_leaf`: all terminal nodes must contain more samples than this
- `min_weight_fraction_leaf`: same as above but expressed as fraction of total number of training examples
- All of these give us stopping conditions
- Generally those aren't that important because we tend to not grow gigantic trees when we boost (depth 1 to 8 is common), sklearn defaults should be fine

Less important parameters

- Any risk in including these extra parameters in a grid search with a validation set?
- The risk is that the more parameters we try to tune the more likely we are to overfit to our validation set. The number of model comparisons grows exponentially with the number of parameters included!!!
- (Additionally, this can become computationally very expensive)

Max_features

- One more parameter worth mentioning: `max_features`
- Number of features we consider for each split
- Same as in random forest!
- How will varying this parameter influence the model? Should we include it in our grid search? Discuss!

Interpreting Boosting Models

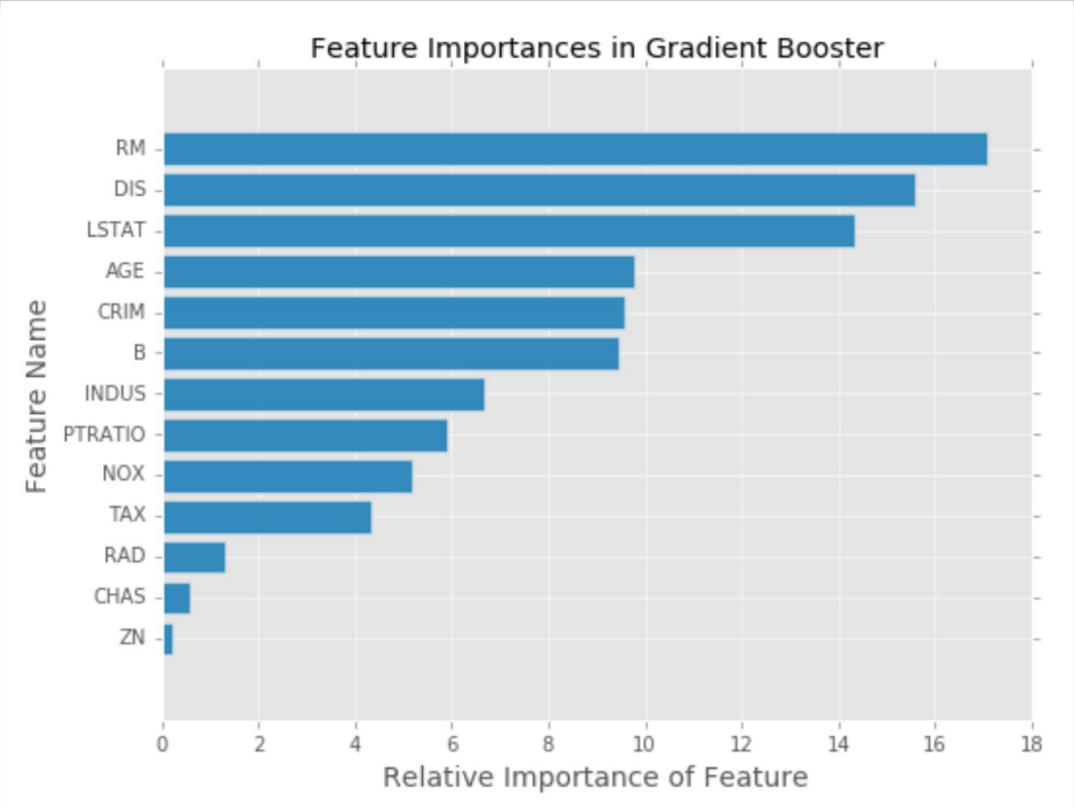
- Boosting models, while very powerful are also very complex and hard to interpret
- There are two main summarization and diagnosis techniques:
  - **Relative Variable Importance:** Measures how much a particular predictor "participates" in the model fitting
  - **Partial Dependence Plots**: Analogous to model parameter estimates in linear regression, these can give us an idea of the effect of a single predictor while holding all other predictors fixed

Relative Variable Importance

galvanize

- Concept the same as for Random Forests
- Each time we do a spit, we measure how much our error metric decreases and attribute it to a predictor
- The importance of a predictor in a tree is how much the total error metric decrease came from splits on that predictor
- For the final boosted model, the importance of a predictor is it's importance averaged over all trees in the model
- We then usually normalize the importances of all predictors to add to 100

Relative Variable Importance



All add up to 100

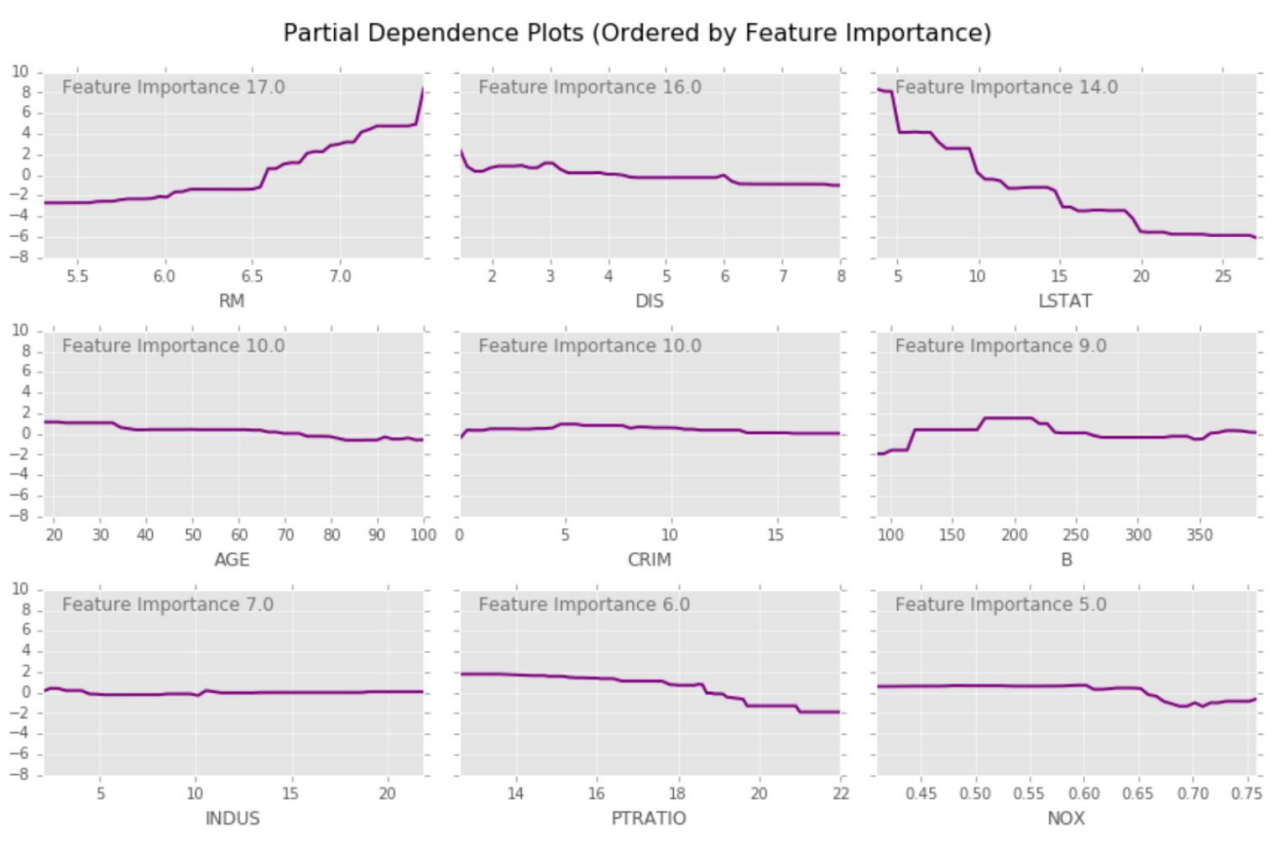
Relative Variable Importance interpretation

galvanize

- If your model contains both numeric and categorical features, feature importances will be higher for numeric ones. Don't try to compare between the two!
- Feature importance rankings can have very high variance, make sure they are robust to different random number generator seeds or different training sets
- Make sure you only calculate feature importances for the model truncated to the optimal n_estimators, otherwise you attribute feature importance to overfitting!
- Dominant features should be treated with suspicion and inspected, they could be a sign of data leakage

Partial dependence plots

Visualization of the effect of a single predictor, averaging out the effects of all other predictors



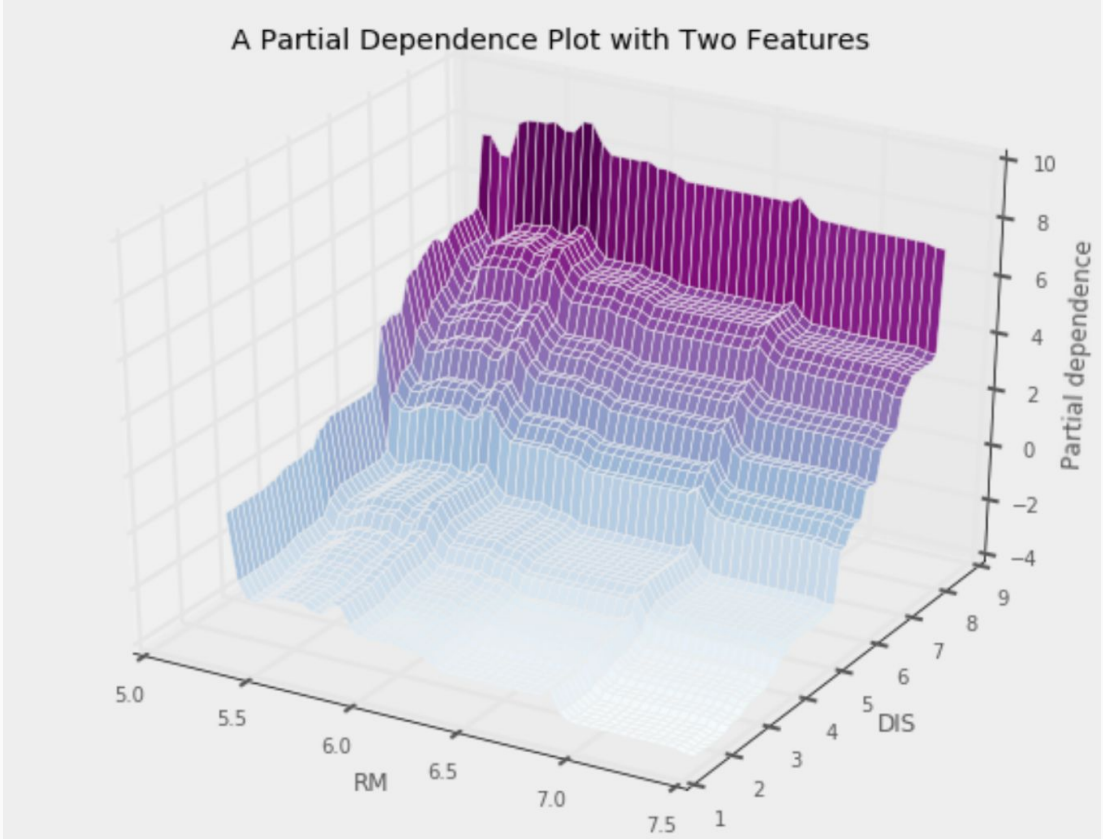
Partial dependence plots

Visualization of the effect of a single predictor, averaging out the effects of all other predictors

$$\underset{j}{\text{pd}}(x) = \frac{1}{N}\sum_{i} f(\overbrace{x_{i1}, x_{i2}}^{\text{The training data points}}, \ldots, \overbrace{x}^{\text{The j'th spot}}, \ldots, x_{iM})$$

(all features averaged out of all data points except position j, which is kept as a variable to plot against)

Partial dependence plots for two predictors

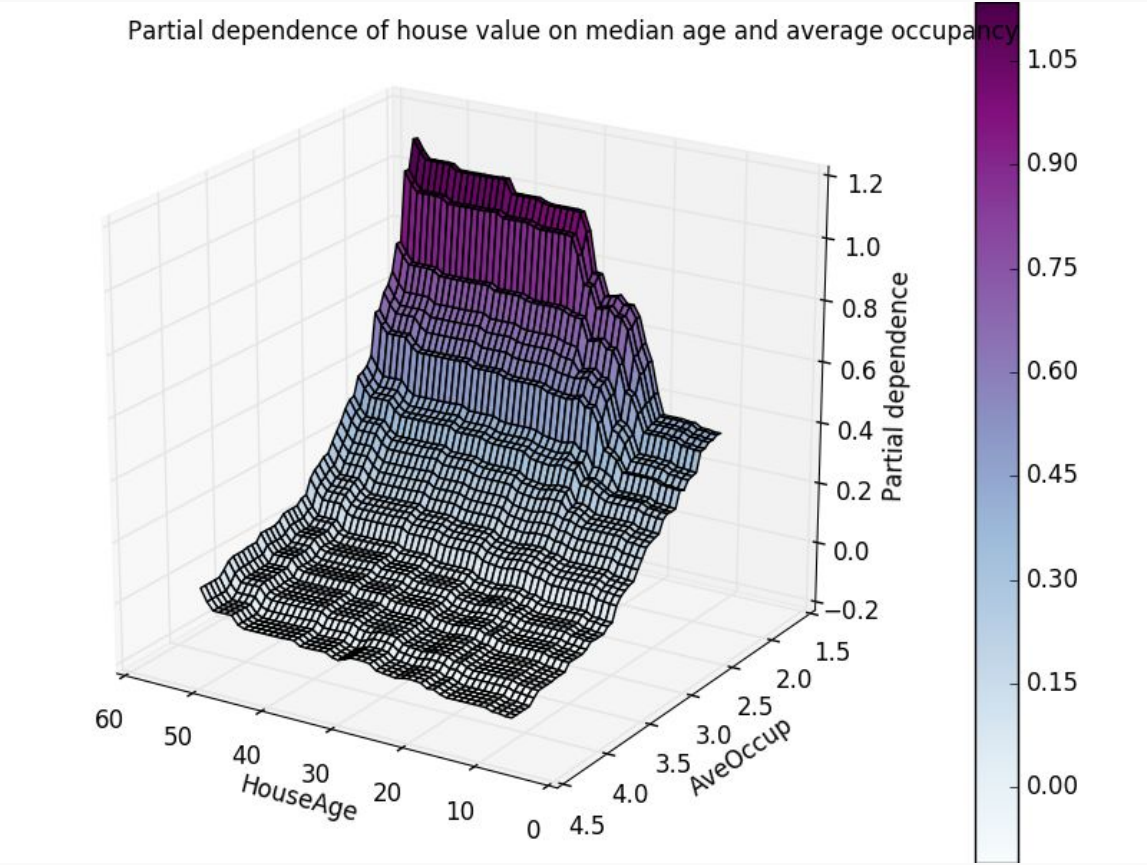


Partial dependence plots for two predictors can reveal synergies

Does it look like any trees split on both dimensions?

Does the dependence on RM depend on where you are on DIS?

Partial dependence plots for two predictors



Partial dependence plots for two predictors can reveal synergies

Does it look like any trees split on both dimensions?

Does the dependence on House Age depend where you are on Average Occupancy?

Other boosting algorithms

The great thing about gradient boosting is that it generalizes well to other loss functions

Examples:

- Gradient boosting logistic regression
  - Minimizes binomial deviance (logistic log likelihood) loss function
- AdaBoost
  - Minimizes exponential loss function

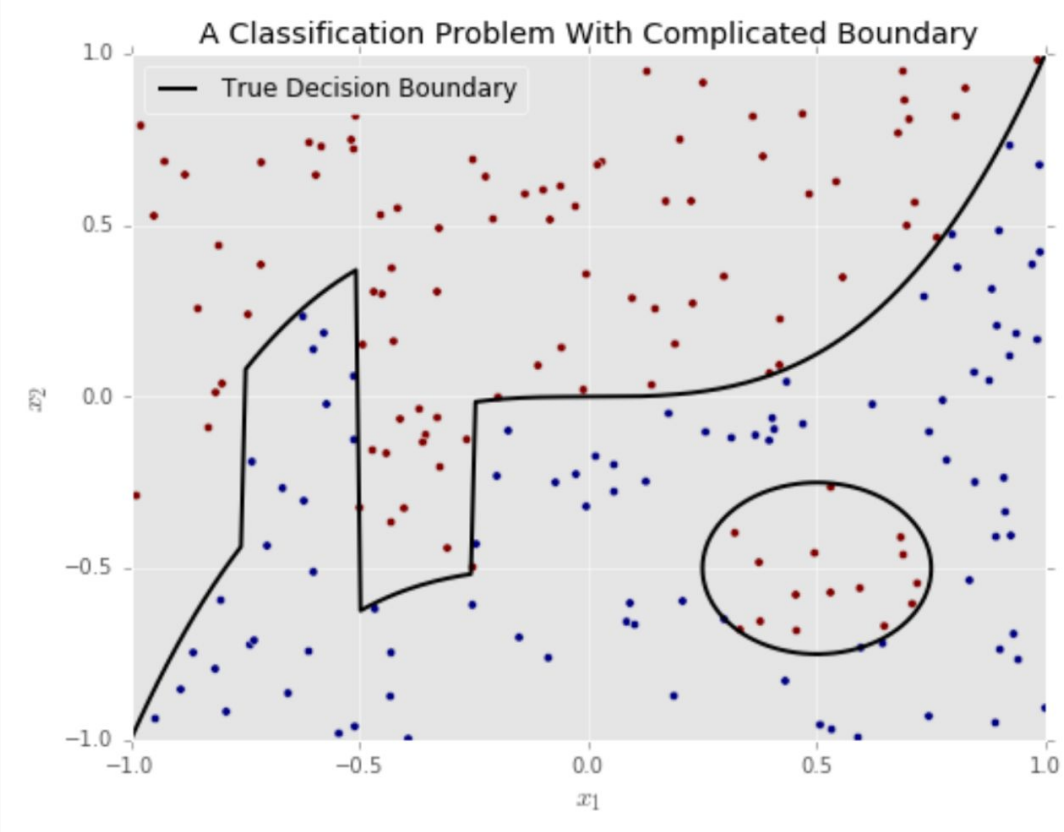
Gradient boosted logistic regression

We want to generalize our boosting algorithm to classification problems:

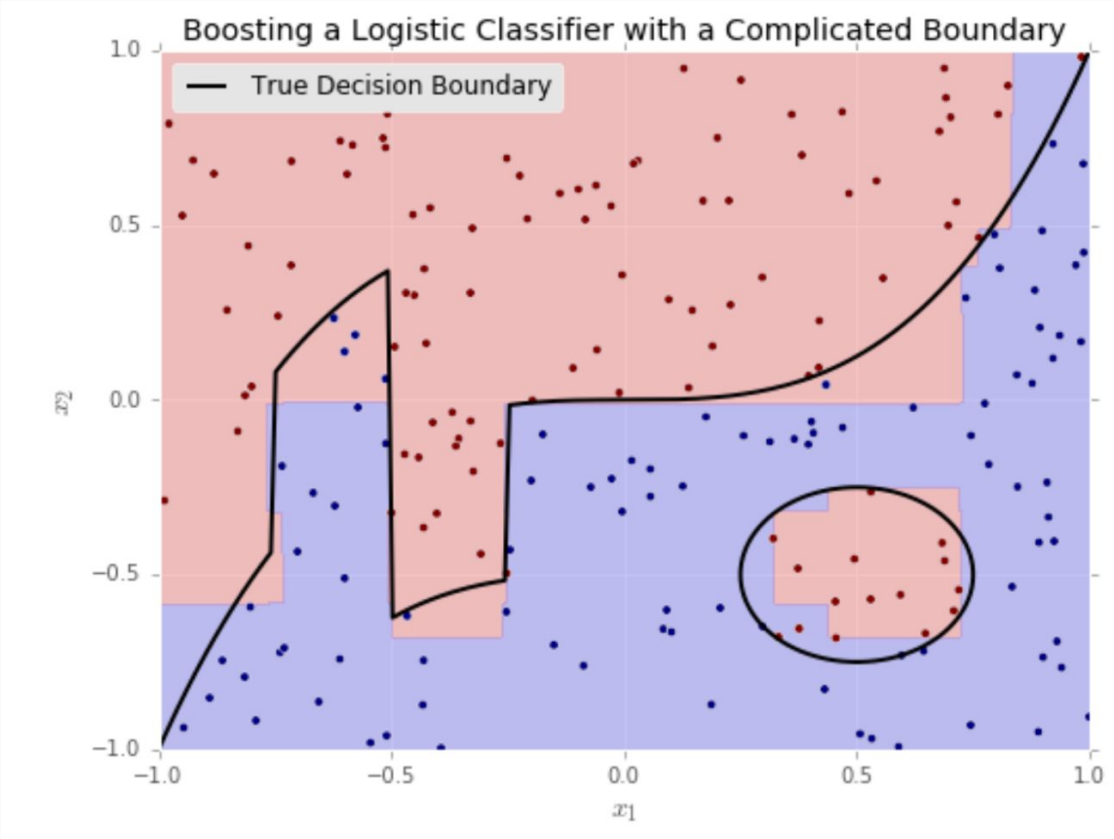
$y \in \{0,1\}$

We want to estimate $f(x) = Pr(y=1|x)$

How would we do this with logistic regression?



Gradient boosted logistic regression has no problem with this!



Logistic regression is minimizing the logistic loss function

$$\hat{\beta} = \arg\min_{\beta} \sum_{i} \left(y_i \nu(\beta, x_i) - \log(1 + e^{\nu(\beta, x_i)}) \right)$$

where $v(\beta, x_i) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$

v is called the “linear predictor”, in Logistic Regression it represents the log-odds of the outcome

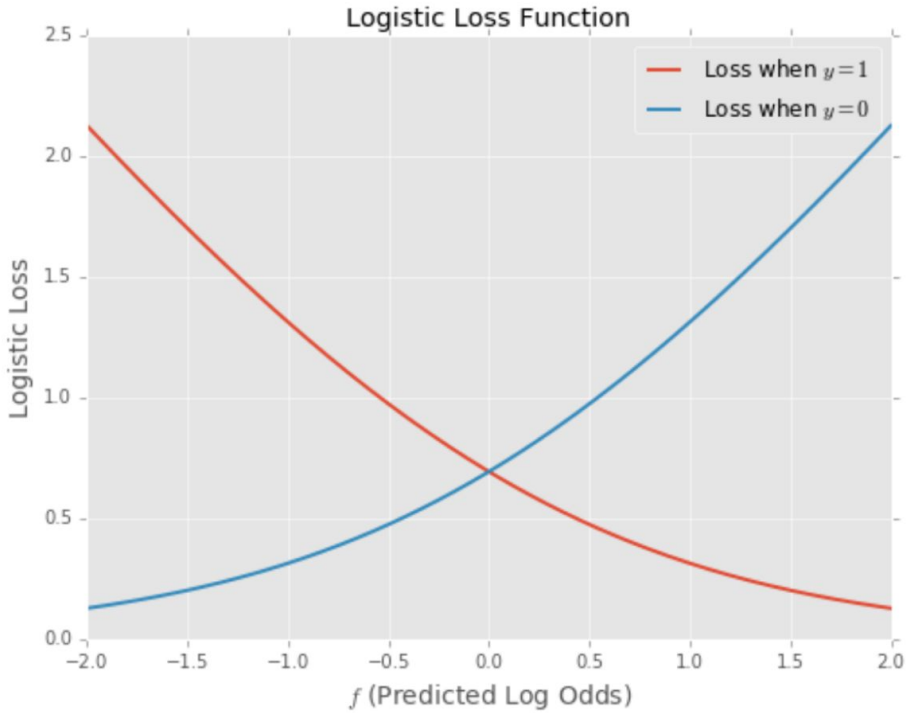
If we have solved for **β**, we can make predictions using

$$p(x) = \frac{1}{1 + e^{-(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_M x_M)}}$$

This can be interpreted as the conditional probability that y=1 given x: $p(x) = Pr(y = 1 \mid x)$

We are minimizing the logistic loss

Logistic regression is minimizing the logistic loss function and can be solved by gradient descent



$$L(f, y) = yf - \log(1 + e^f)$$

$f = v(\beta, x_i) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$

L is just another way of penalizing distance in the case of classification (it is penalizing misclassification by how far it is from the boundary)

f represents the distance from the boundary (like SVM)

We simply replace our linear predictor with a sum of regression trees

$$\nu(\beta, x) = \beta_0 + \beta_1 x_1 + \cdots + \beta_M x_M$$

becomes

$$\nu(x) = T_0(x) + T_1(x) + \cdots + T_{\max}(x)$$

And replace the square loss with the logistic loss:

$$L(f, y) = \frac{1}{2}(f - y)^2 \longrightarrow L(f, y) = yf - \log(1 + e^f)$$

What is the gradient of the logistic loss?

$$\begin{aligned}\nabla_f L(f, y) &= \frac{\partial}{\partial f}\left(yf - \log(1 + e^f)\right) \\ &= y - \frac{e^f}{1 + e^f} \\ &= y - p(f)\end{aligned}$$

This basically means we are boosting to the “residual probabilities” or the “probability residuals”.

Gradient Boosted Logistic Regression in practice

galvanize

```
from sklearn.ensembles import GradientBoostingClassifier
model = GradientBoostingClassifier()
# Now y must be a np.array of 0 and 1's!
model.fit(X, y)
```

```
GradientBoostingClassifier(loss='deviance',
                           n_estimators=100,
                           learning_rate=0.1,
                           max_depth=3,
                           subsample=1.0,
                           min_samples_split=2,
                           min_samples_leaf=1,
                           min_weight_fraction_leaf=0.0,
                           ...)
```

Everything we said before generalizes

Gradient Boosted Logistic Regression in practice

galvanize

To make predictions use `predict_proba()`

```
model.predict_proba(X)
```

The `predict()` methods will return 0 or 1 based on the probabilities

`predict_proba()` is more useful

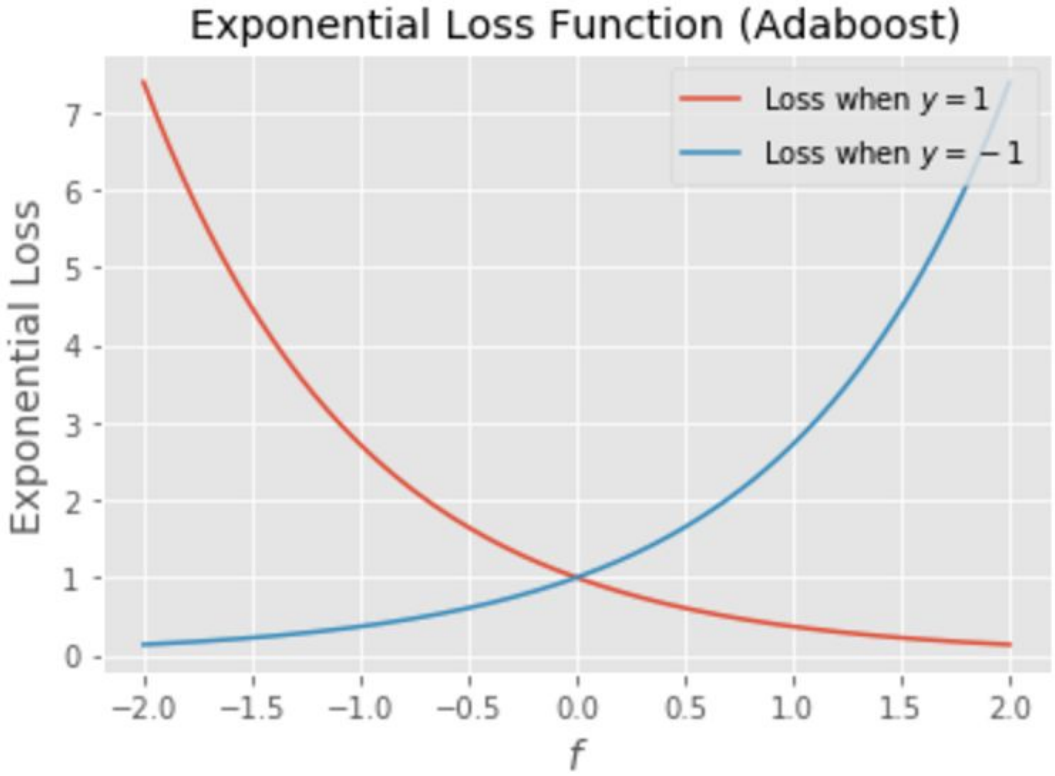
There is particular type of boosting classifier called AdaBoost (adaptive booster), which is a historically separate boosting algorithm

It turns out to be equal to Gradient Boosting with an exponential loss function

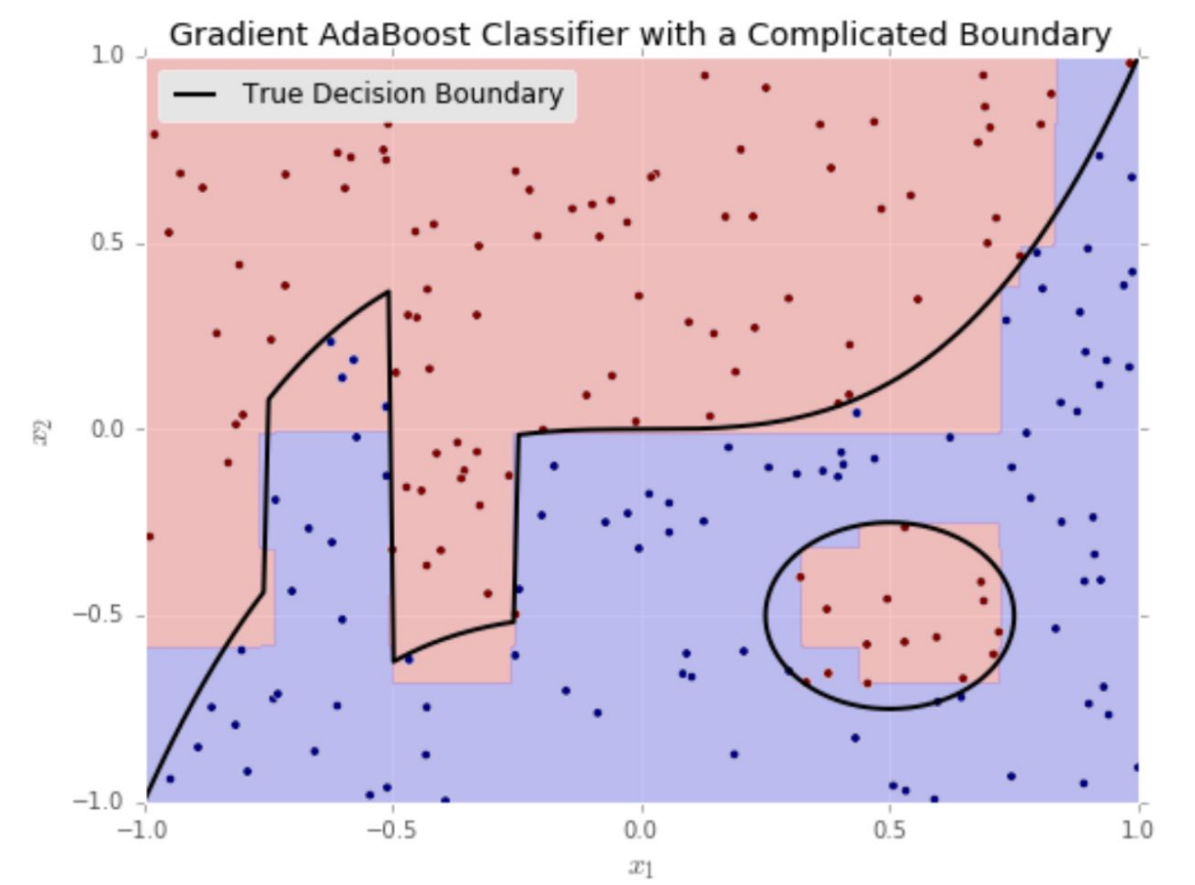
```
model = GradientBoostingClassifier(loss='exponential')
model.fit(X, y)
```

AdaBoost by convention uses labels {-1,1} and minimizes this loss function:

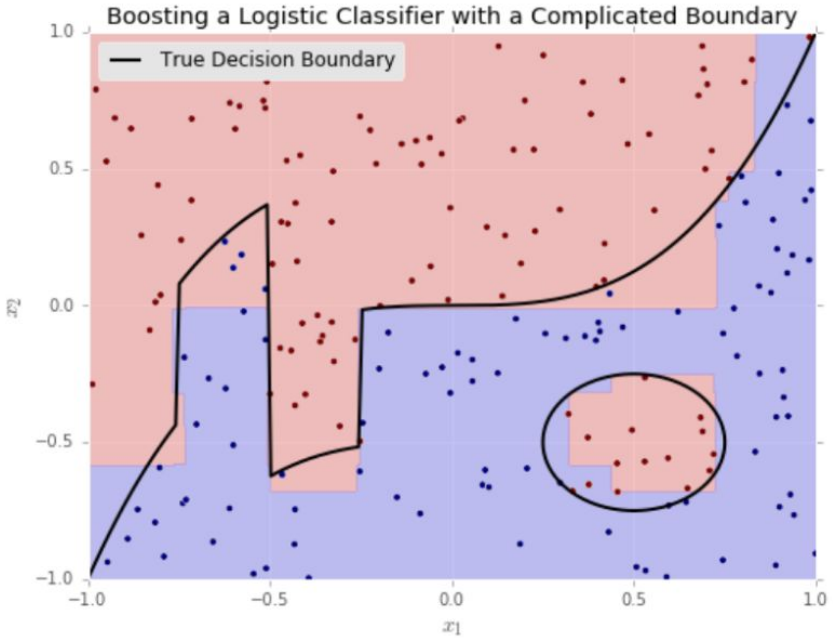
$$L(f, y) = \exp(-yf)$$

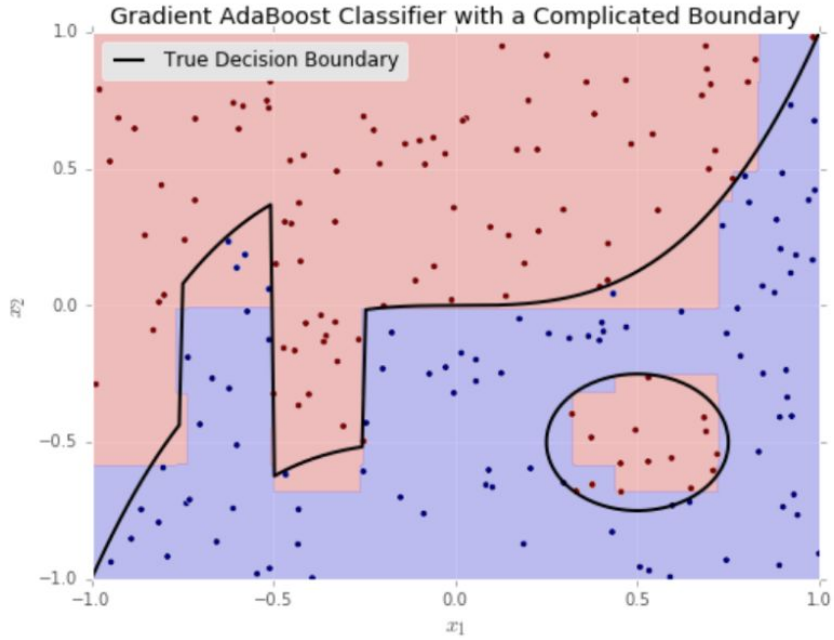


AdaBoost performance is comparable to Gradient Boosted LogReg



In fact the boundaries are identical

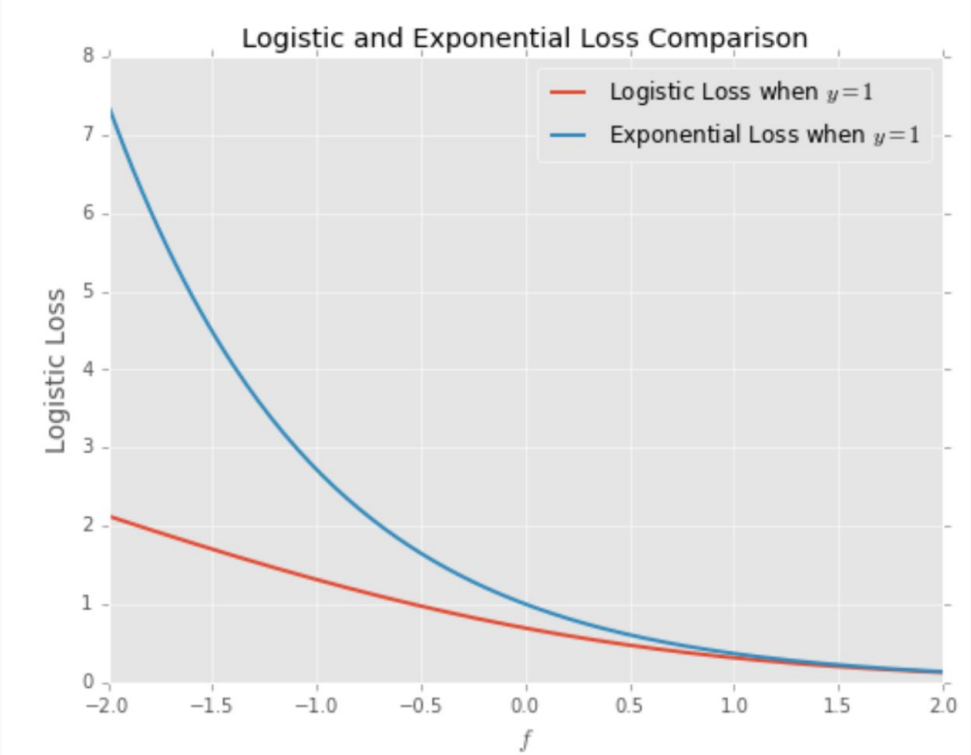




AdaBoost vs. Gradient Boosted Logistic Regression

Gradient boosted logistic regression is **less** sensitive to outliers

Can you see why?



Historical AdaBoost Classifier

The implementation of AdaBoost as a Gradient Booster is a more recent development

Historically (before the invention of Gradient Boosting), AdaBoost used a complicated reweighting scheme

The traditional AdaBoost is also included in sklearn:

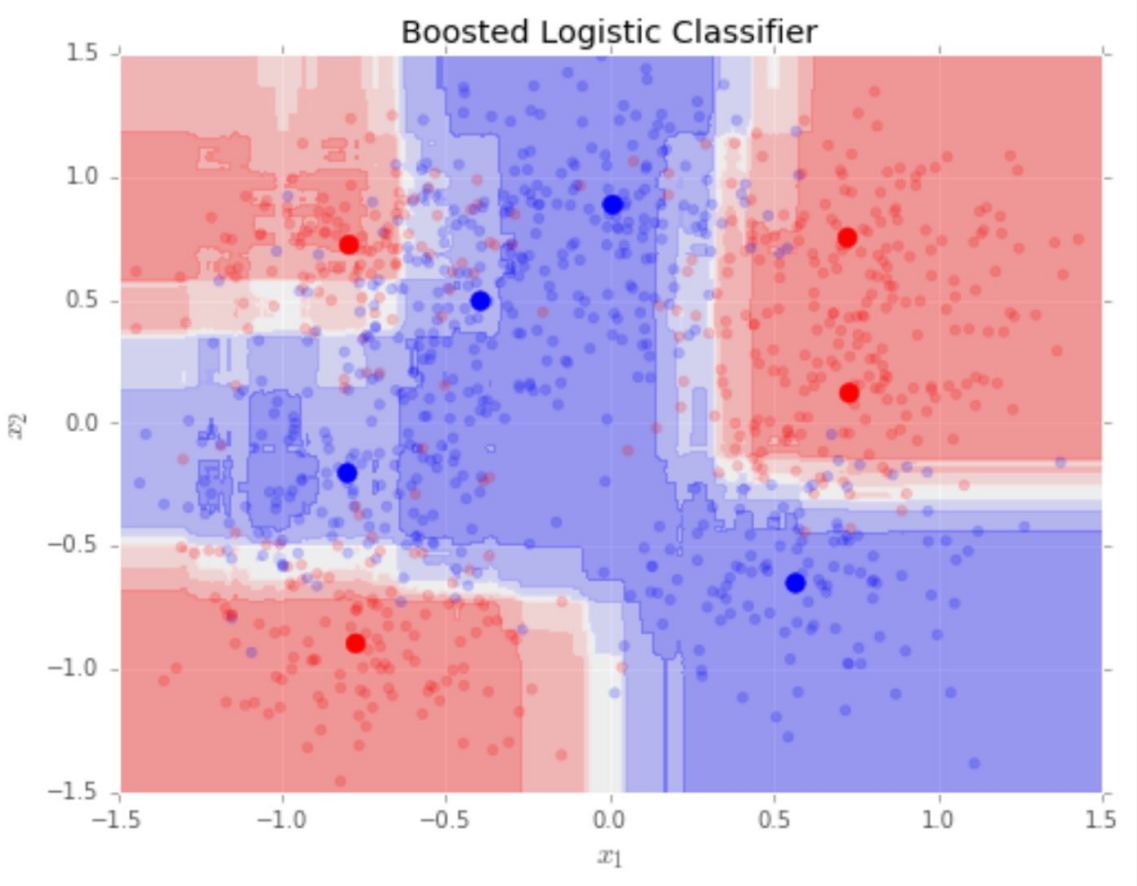
```
model = AdaBoostClassifier()
model.fit(X, y)
```

The traditional AdaBoost has no max_depth hyperparameter and always uses “stumps” (depth=1)

Traditional AdaBoost is outperformed by Gradient boosting, but it retains historical and mathematical significance

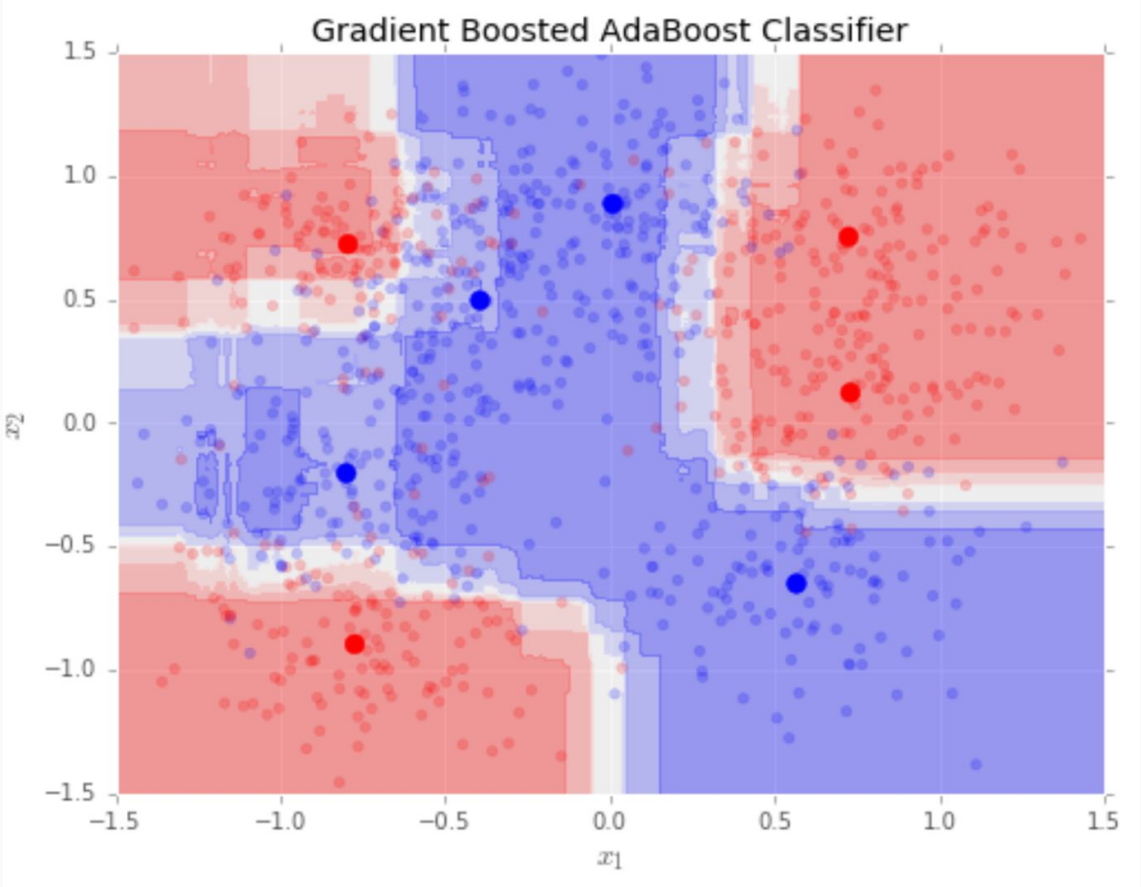
Comparison of Classification Algorithms

galvanize



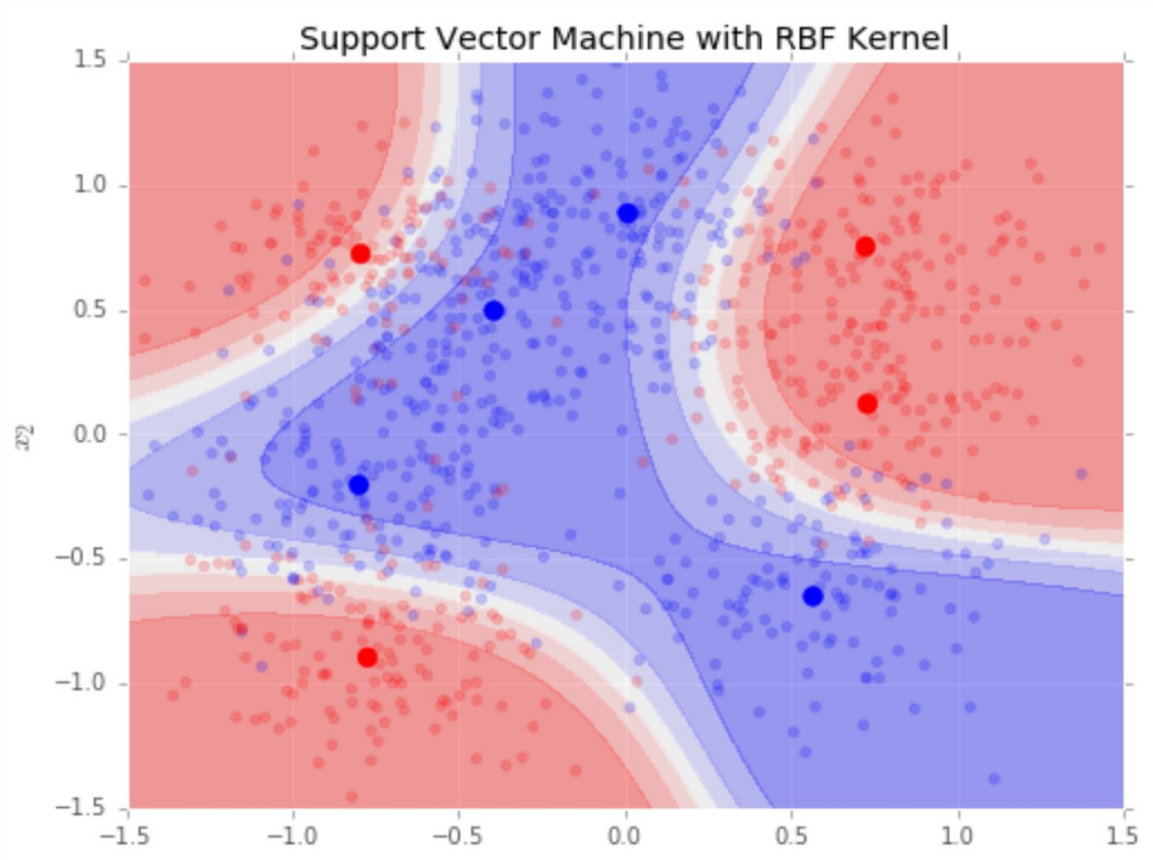
Comparison of Classification Algorithms

galvanize



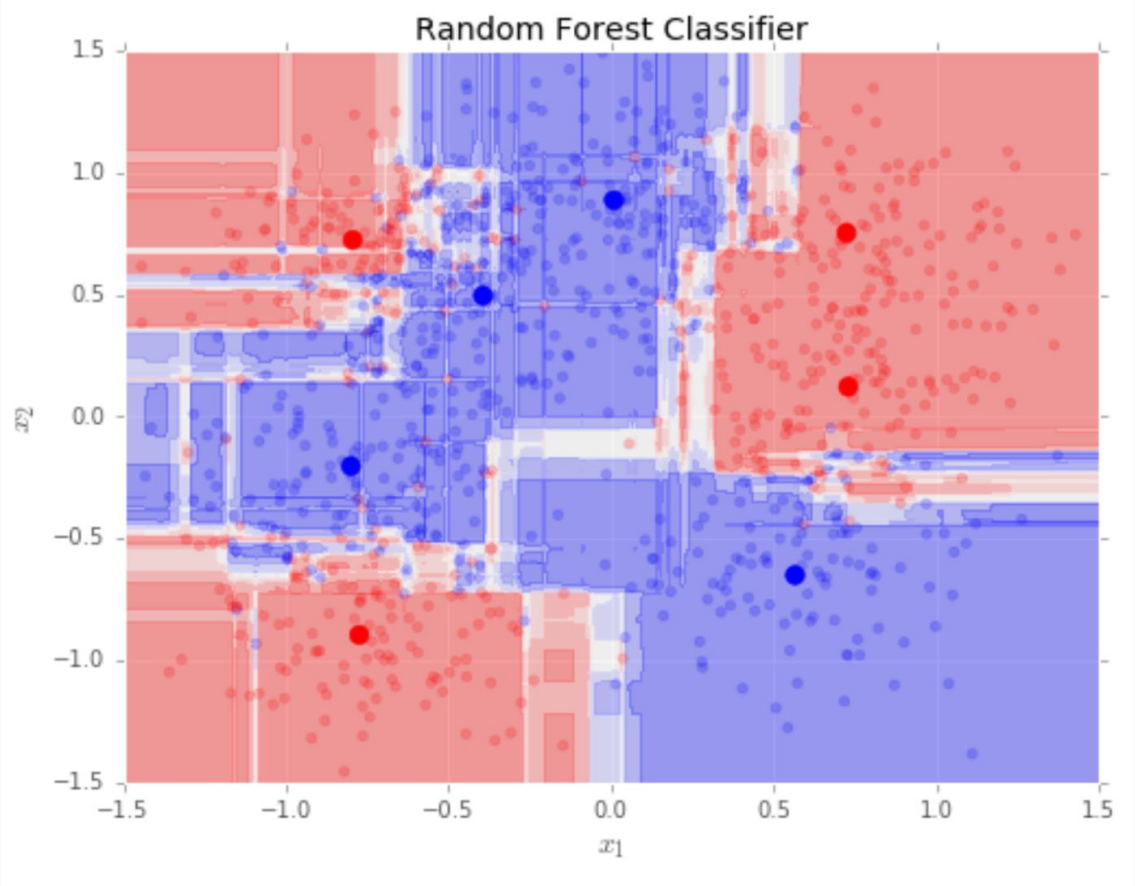
Comparison of Classification Algorithms

galvanize



Comparison of Classification Algorithms

galvanize



Last words about boosting

galvanize

Gradient boosting is one of the best “off the shelf” or “out of the box” machine learning algorithms available today

It effortlessly produces accurate models

But it has drawbacks:

- Models are very complex, it is hard to extract intuitive or conceptual conclusions
- The algorithm is difficult to explain to non-experts (hard to convince business people to accept conclusions from such a black box model)
- Boosting models may be difficult to implement in production
- The sequential nature makes them non parallelizable (as opposed to random forests). Recent improvements (XGBoost)

Boosting

galvanize

Objectives

- Get an intuitive understanding of boosting
- Understand the algorithm's hyperparameters
- State the difference between boosting and bagging/RF
- Understand the bias/variance tradeoff of boosting and other ensemble algorithms
- State basic strategies for interpreting a booster