Boosting

Matthew Drury

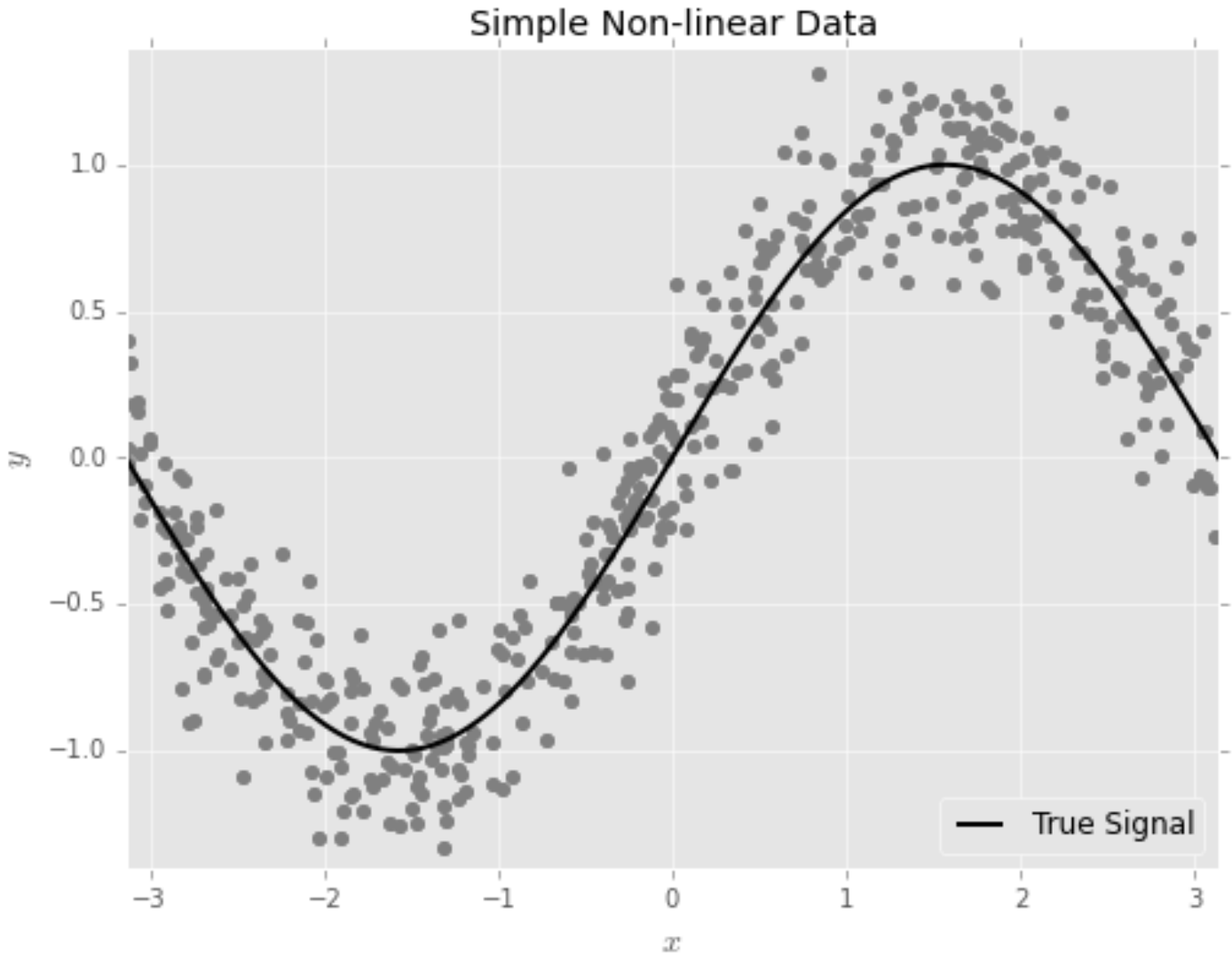
August 14, 2016

Introduction to Boosting

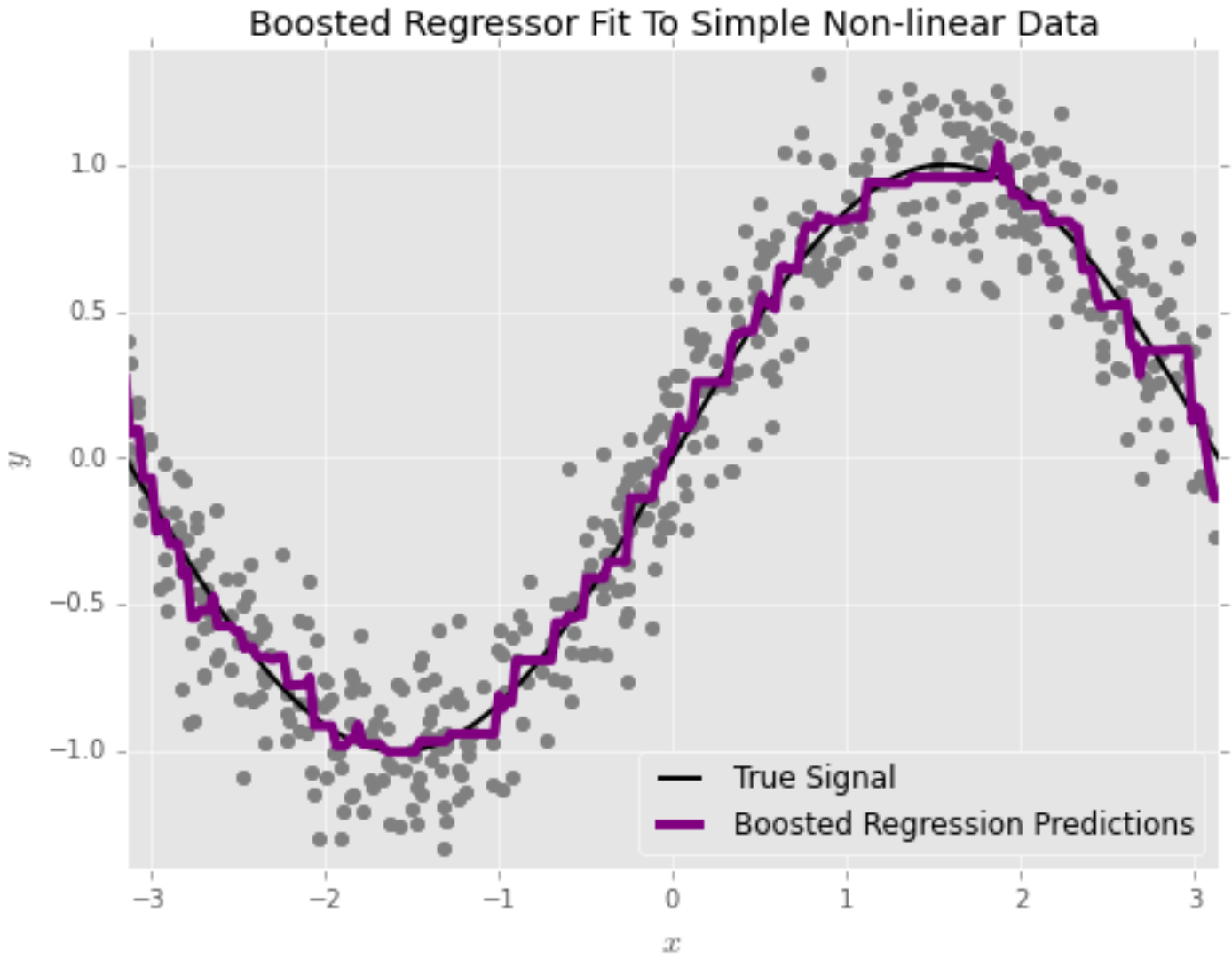
Boosting encompasses a highly successful set of learning algorithms.

- 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.

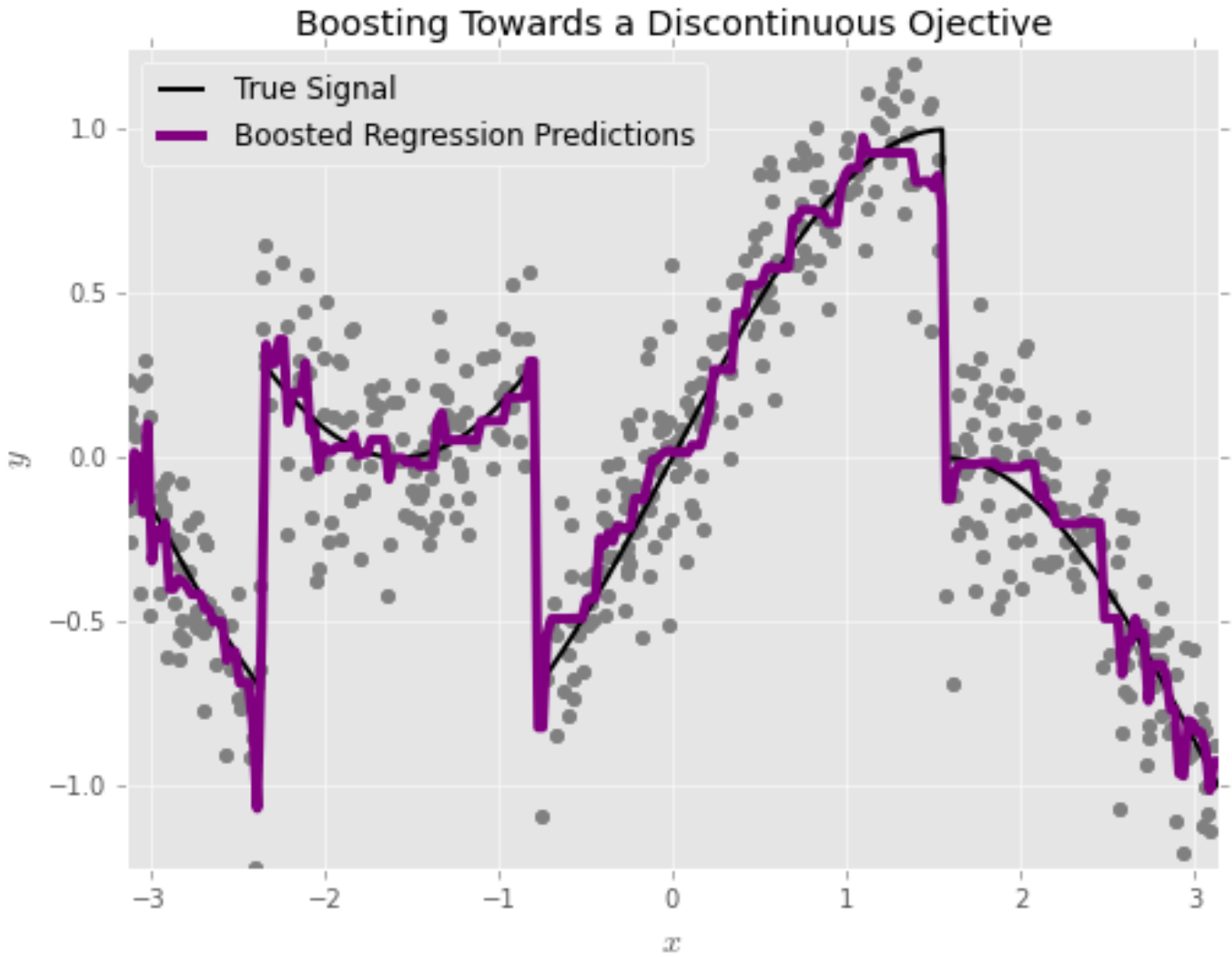
Boosting can adapt itself effortlessly to very non-linear objectives



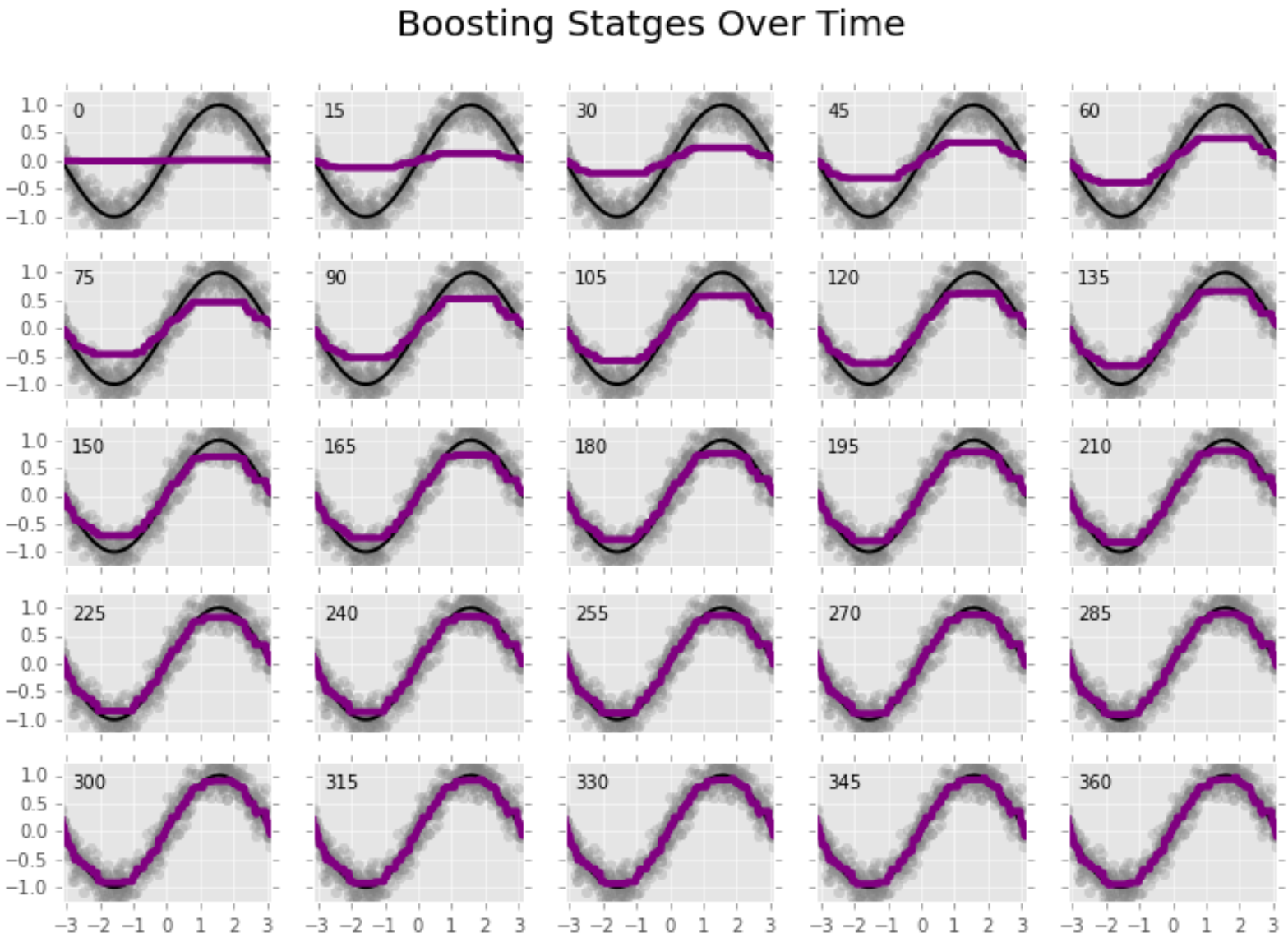
Boosting can adapt itself effortlessly to very non-linear objectives



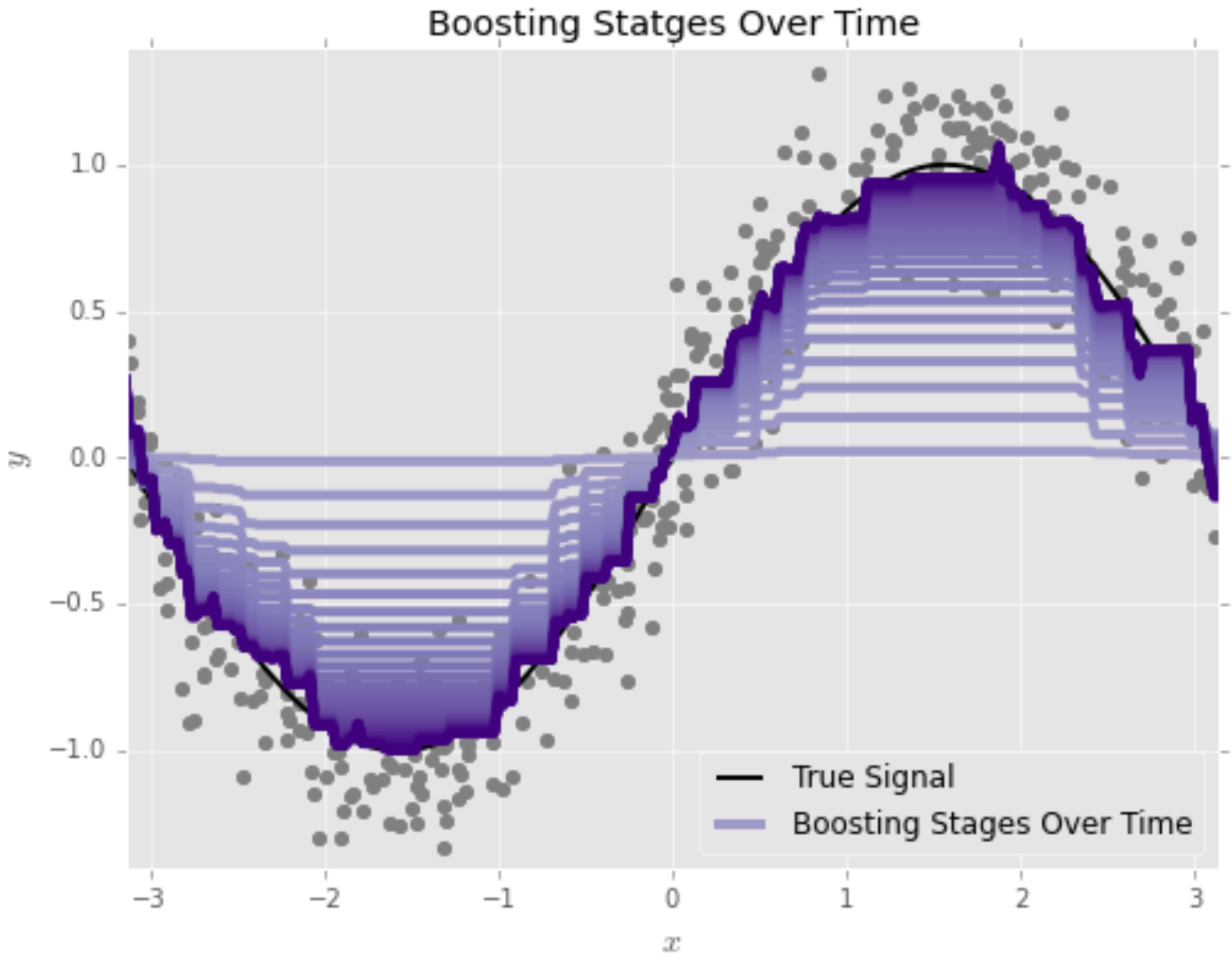
Boosting can adapt itself effortlessly to very non-linear objectives



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
- Interpreting 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

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

Practical Gradient Boosted Regression

Scikit-learn includes the gradient boosted regression algorithm in the `ensembles` module

```
from sklearn.ensemble import GradientBoostedRegressor
```

`GradientBoostingRegressor` has many knobs to turn.

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

A `GradientBoostingRegressor` object is fit in the same way as every other leaning model in sklearn

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

The `predict` method returns predictions on new data

```
preds = model.predict(X_new)
```

Especially useful is the iterator `staged_predict` which creates predictions from models created by truncating series of trees

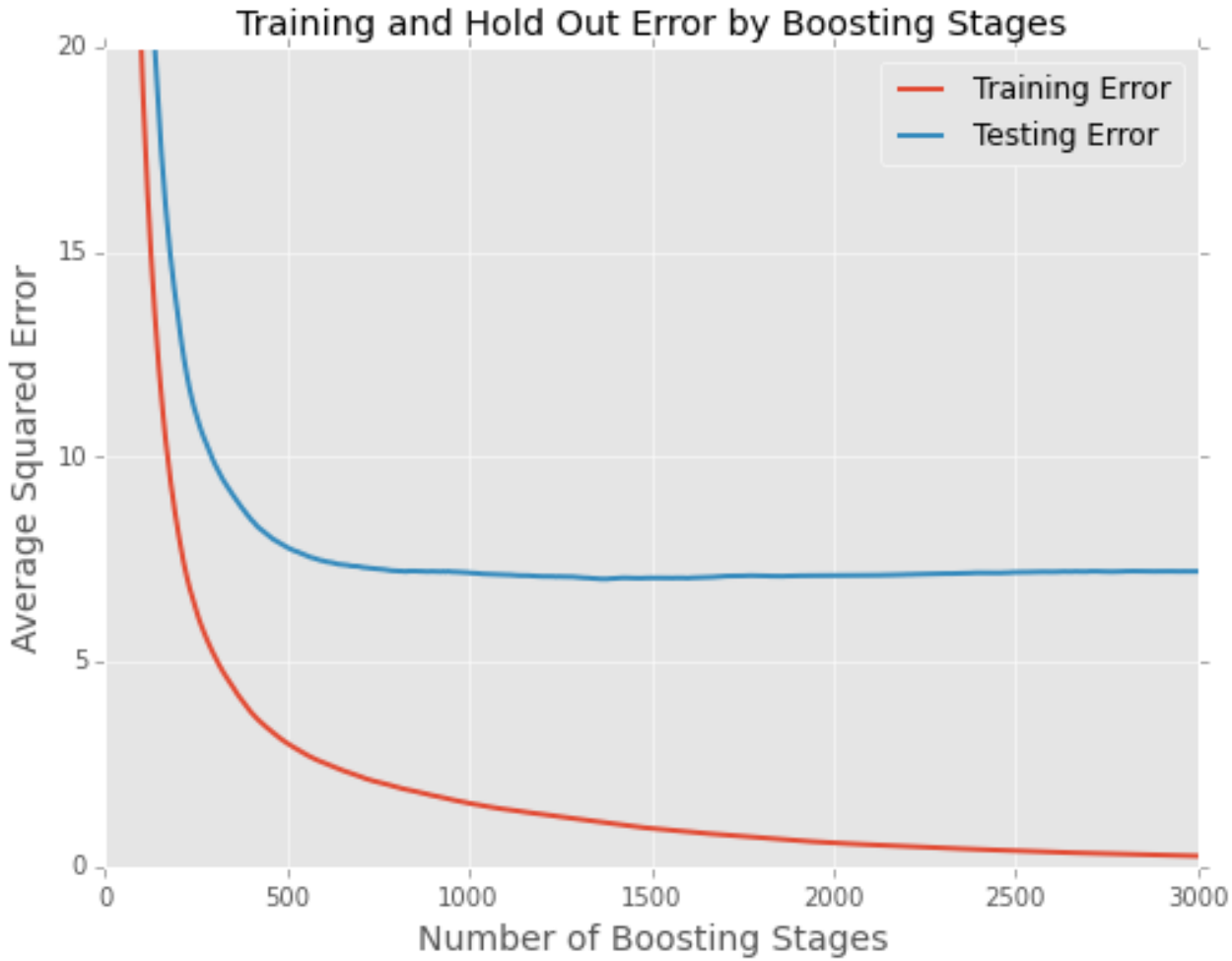
```
for preds in model.staged_predict(X_new):
    # Do something interesting, see the plots below.
```

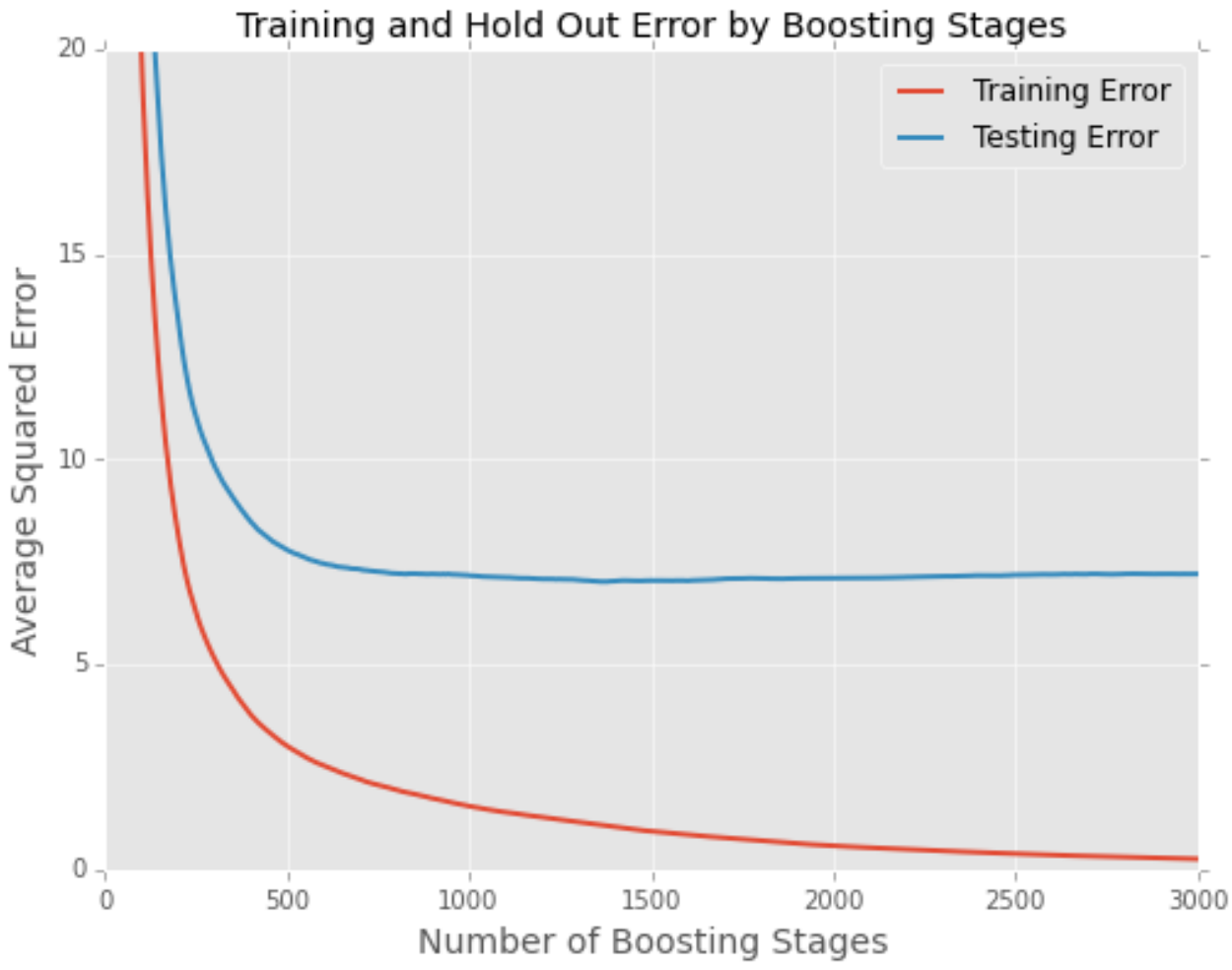
The most important options to `GradientBoostedRegressor` are

- `loss` controls the loss function to minimize. `ls` is the least squares minimization algorithm we discussed in the previous section.
- `n_estiamtors` is how many boosting stages to compute, i.e. how many regression trees to grow.
- `learning_rate` is the learning rate for the gradient update.
- `max_depth` controls how deep to grow each individual tree.
- `subsample` allows to fit each tree on a random sample of the training data (like bagging in random forests).

Tuning the Number of Estimators

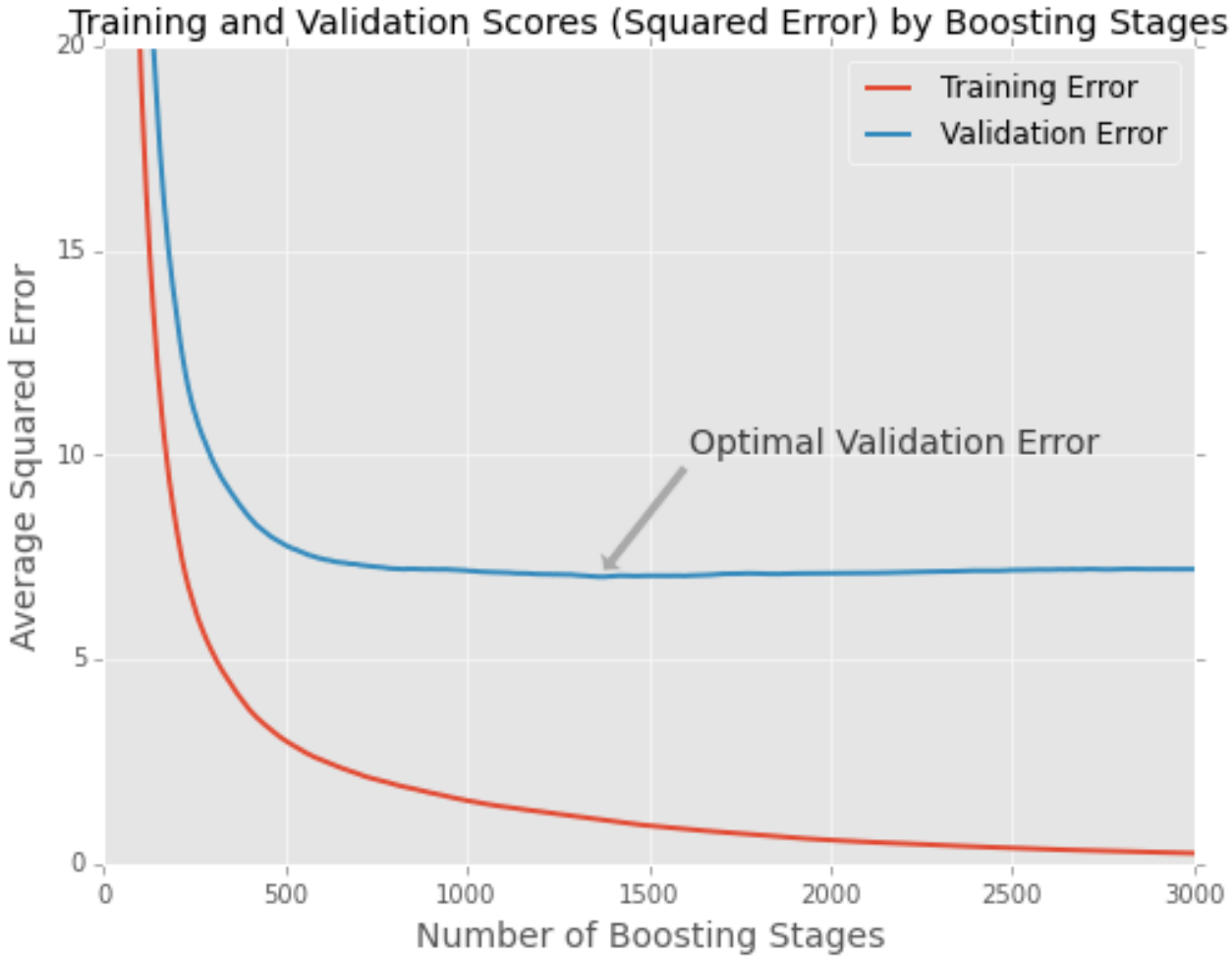
As more and more trees are added to the model the training set error will be driven down monotonically, but the same is not true for the testing error





This means that it is essential to determine the proper number of trees to grow, as too many may lead to overfitting

One way to tune the numer of trees is to make use of a validation set, held out from both training and testing

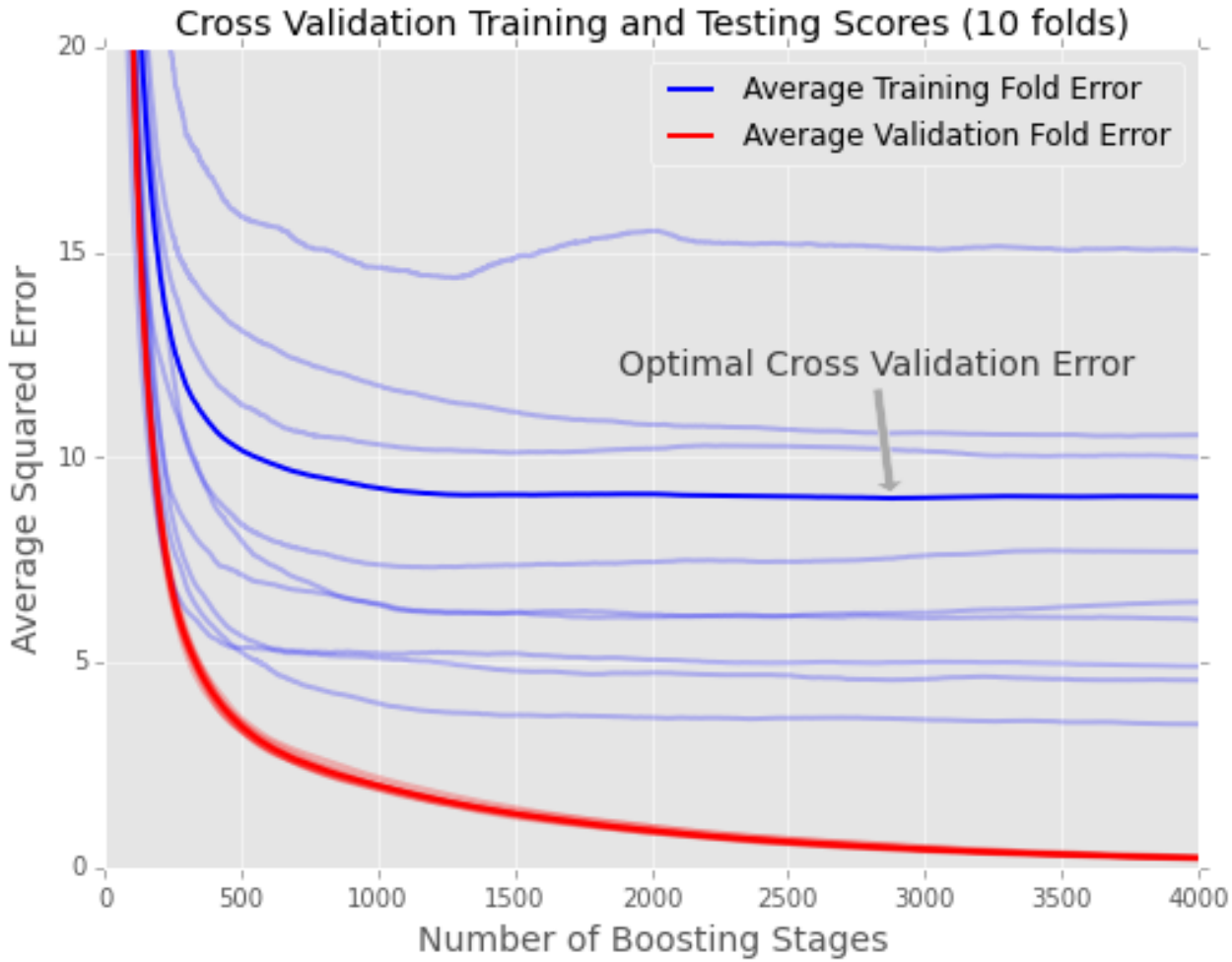


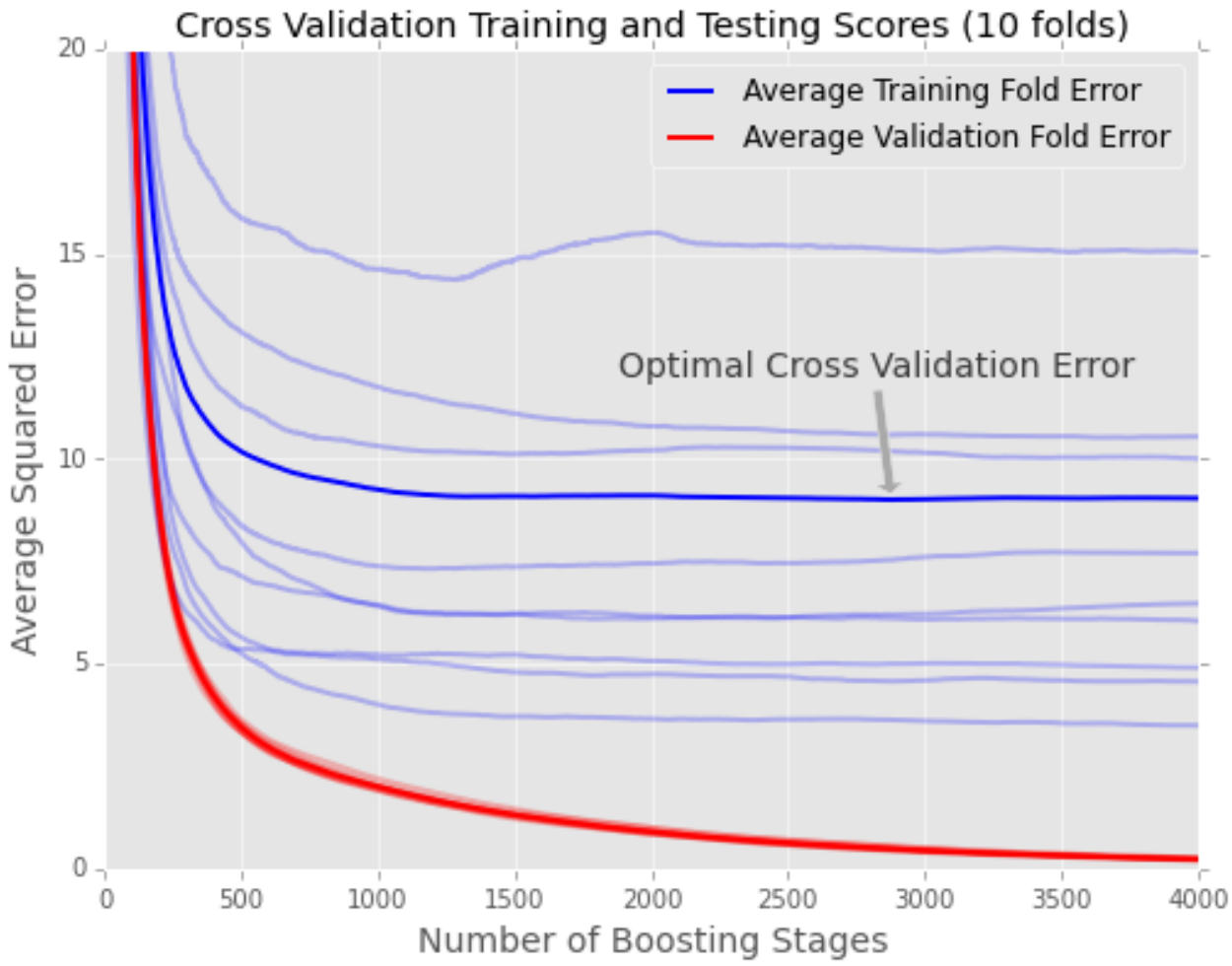
The `loss_` method is important here, it allows us to compute the loss function on held out data

```
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]
```

Another way to choose the optimal number of trees is to replace the validation set with cross validation

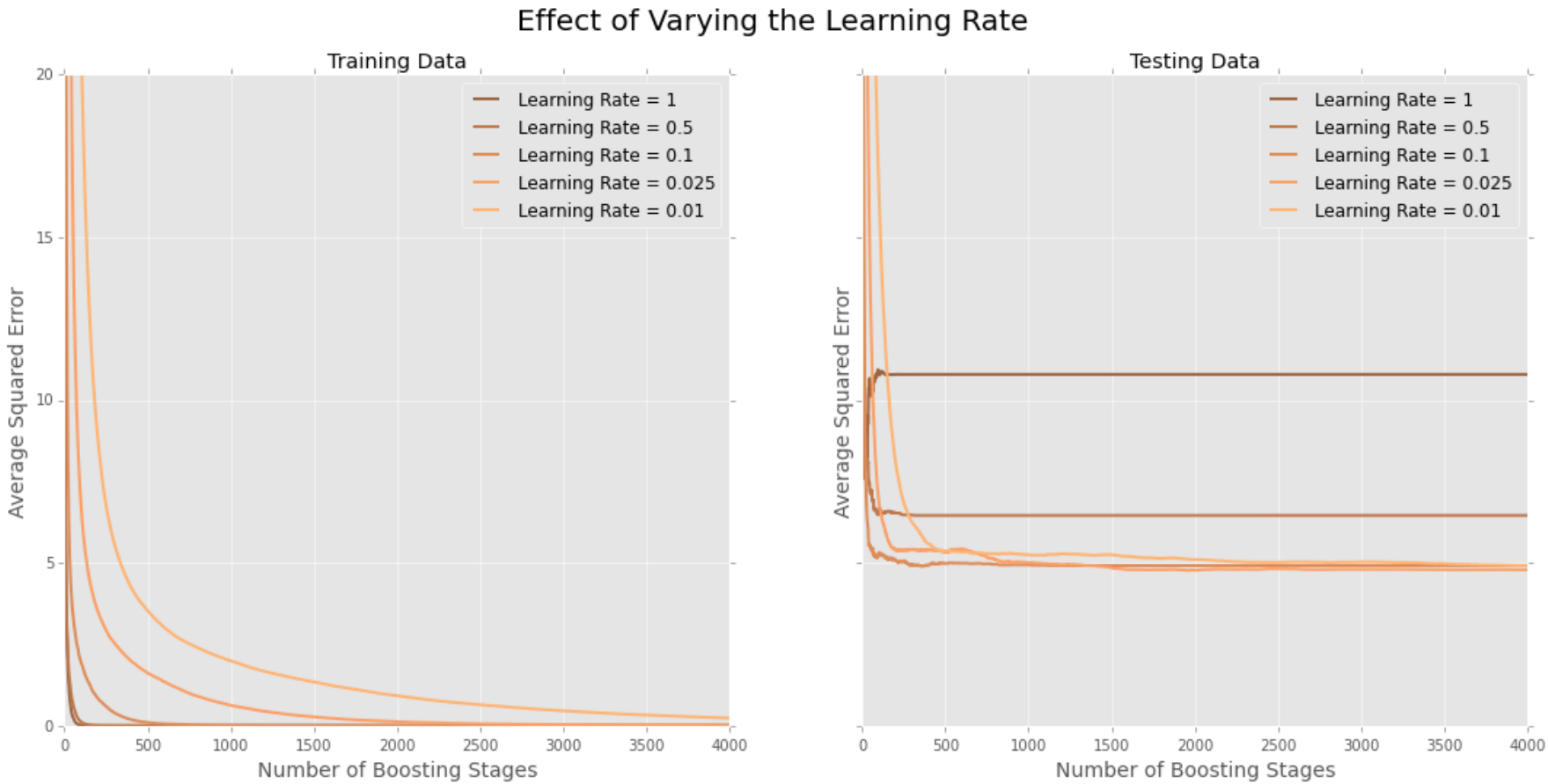




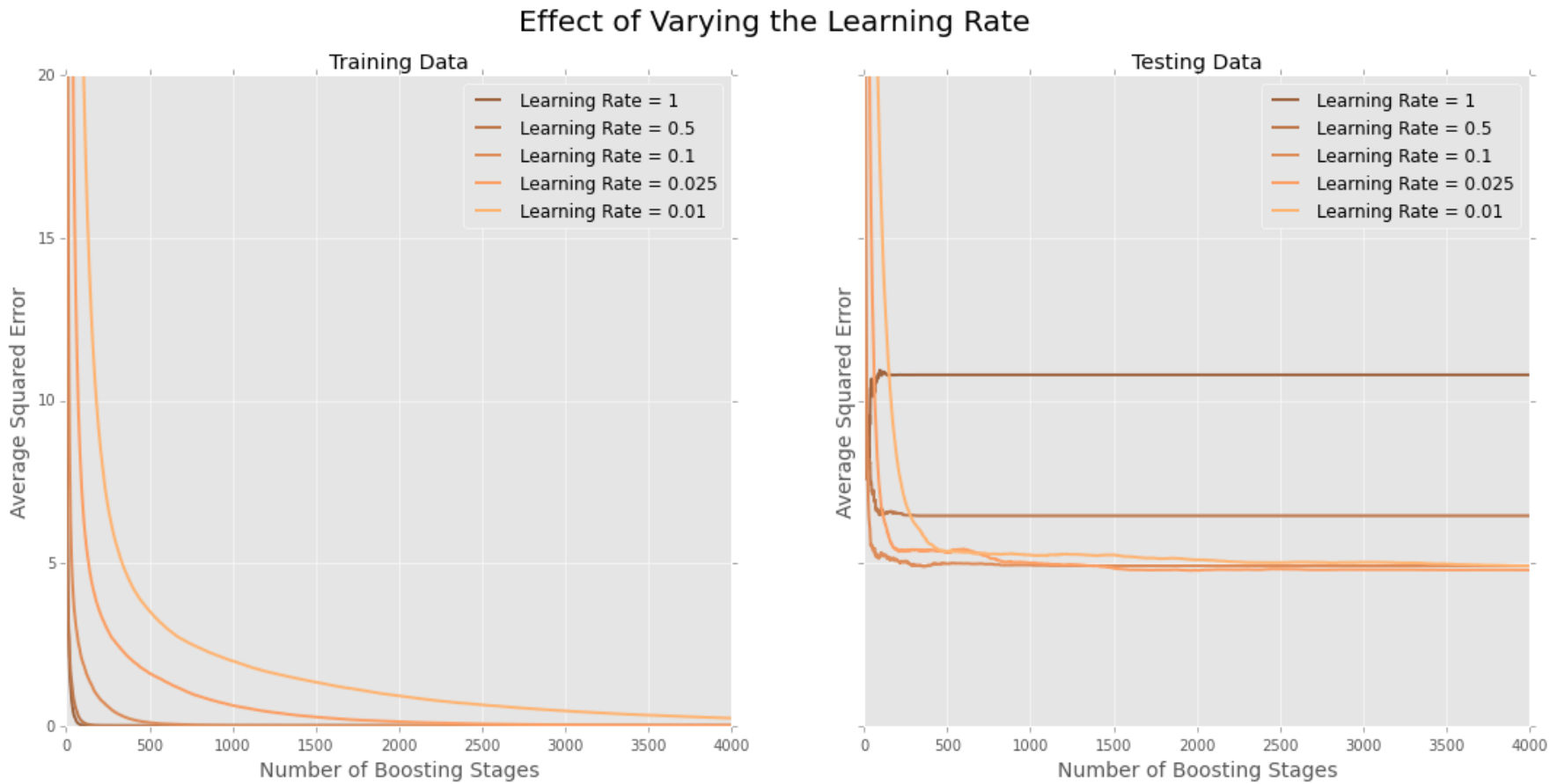
We generally choose the number of trees minimizing the *average* out validation fold error.

Tuning the Learning Rate

In general, a smaller learning rate results in a more accurate model in the long run



On the other hand, a smaller learning rate means more trees are needed to reach the optimal point



Strategy for Learning Rate:

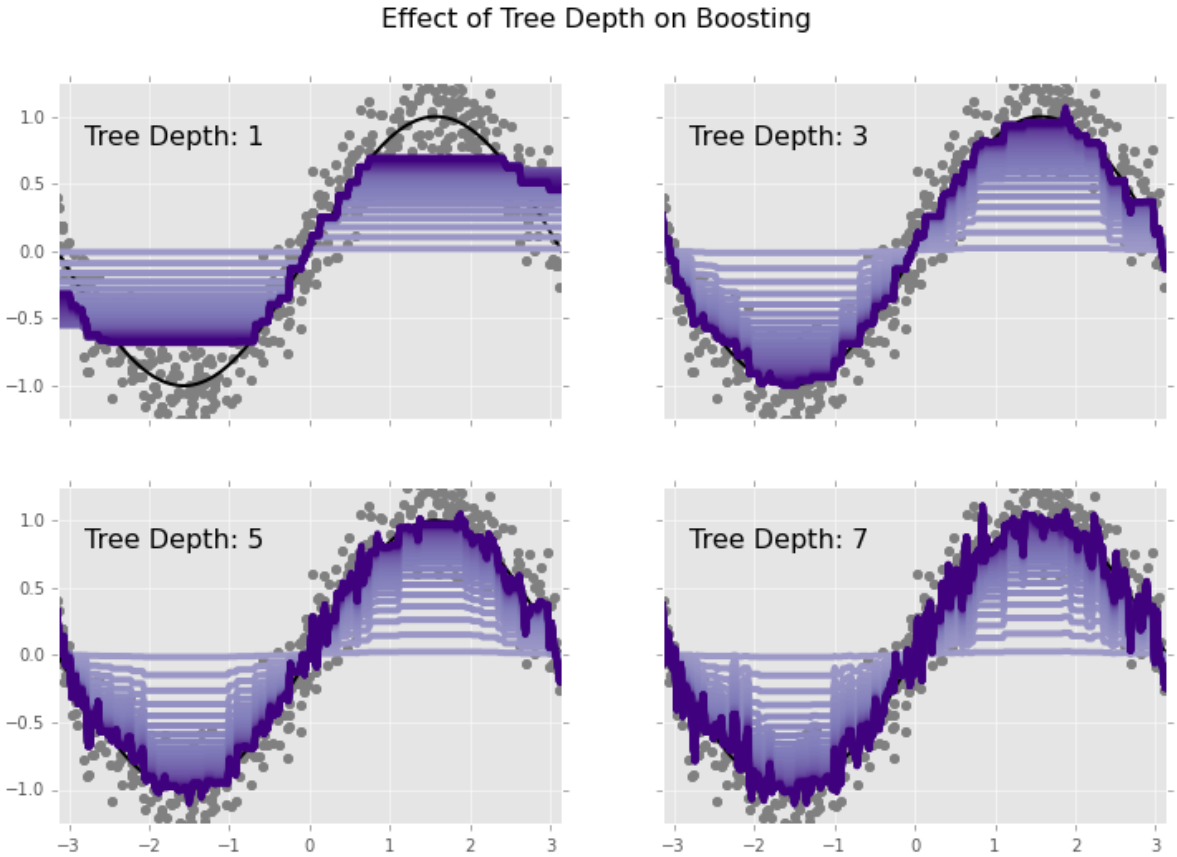
- In the initial exploratory phases of modeling, set the learning rate to some large value, say 0.1. This allows you to iterate through ideas quickly.
- When tuning other parameters using grid search, decrease the learning rate to a more sensible value, 0.01 works well.
- When fitting the *final* production model, set the learning rate to a very small value, 0.001 or 0.0005, smaller is better.

Run the final model overnight! It will fit while you are sleeping!

Make sure you've written all your analysis code up front, based on your initial models. Then all that's left is to run your final model through and see your final statistics.

Tuning the Tree Depth

A larger tree depth allows the model to capture deeper interactions between the predictors.



Unfortunately, a deeper tree depth also causes the model to fit faster, somewhat combating the effect of the learning rate.

It's never obvious up front what tree depth is best for a given problem, so a grid search is needed to determine the best value