Boosting

Matthew Drury

August 16, 2016

Introduction to Boosting

Boosting encompasses a highly successful set of learning algorithms.

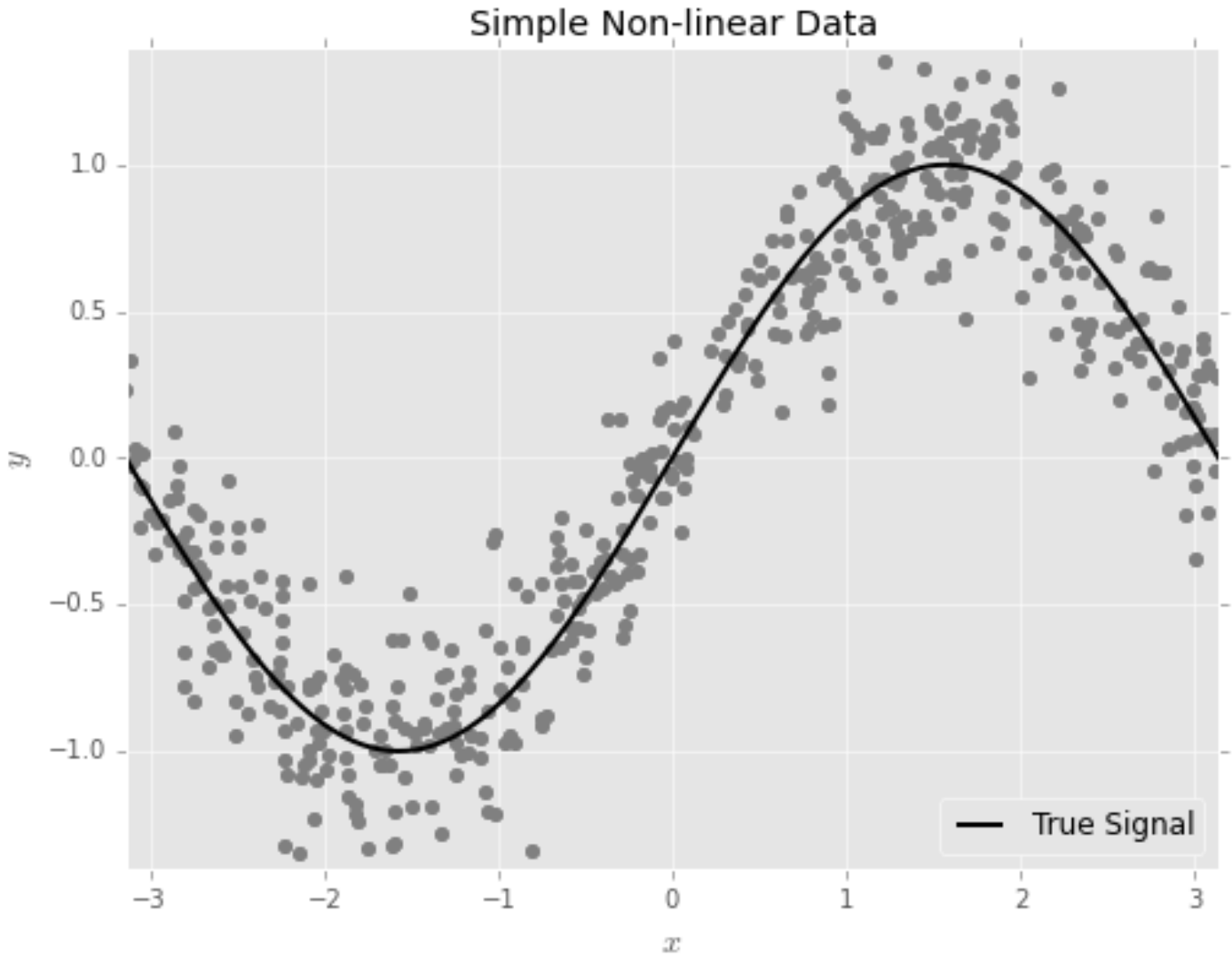
Boosting encompasses a highly successful set of learning algorithms.

- Allstate has held three Kaggle competitions. All three were won by algorithms incorporating gradient boosting as a fundamental component.

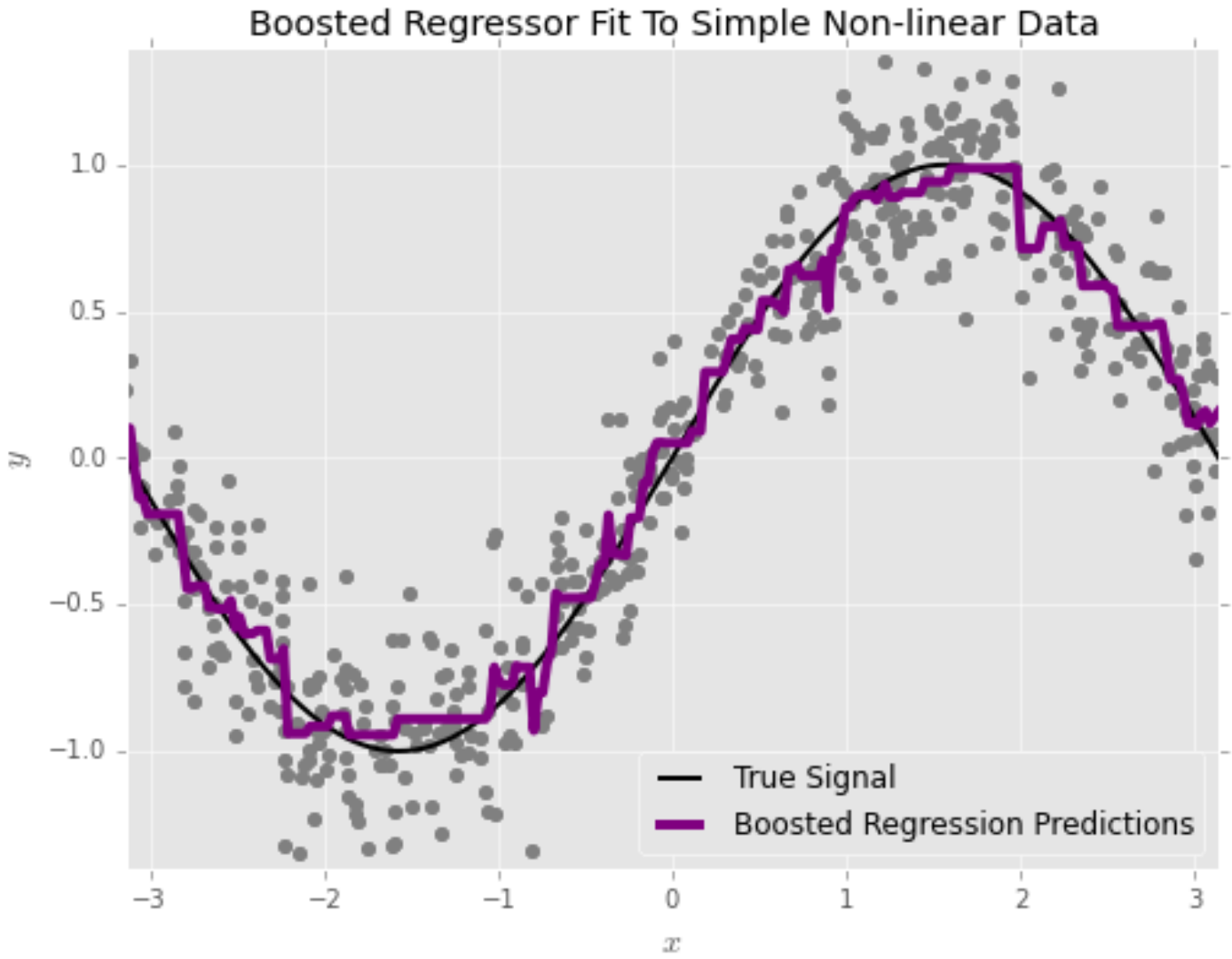
Boosting encompasses a highly successful set of learning algorithms.

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

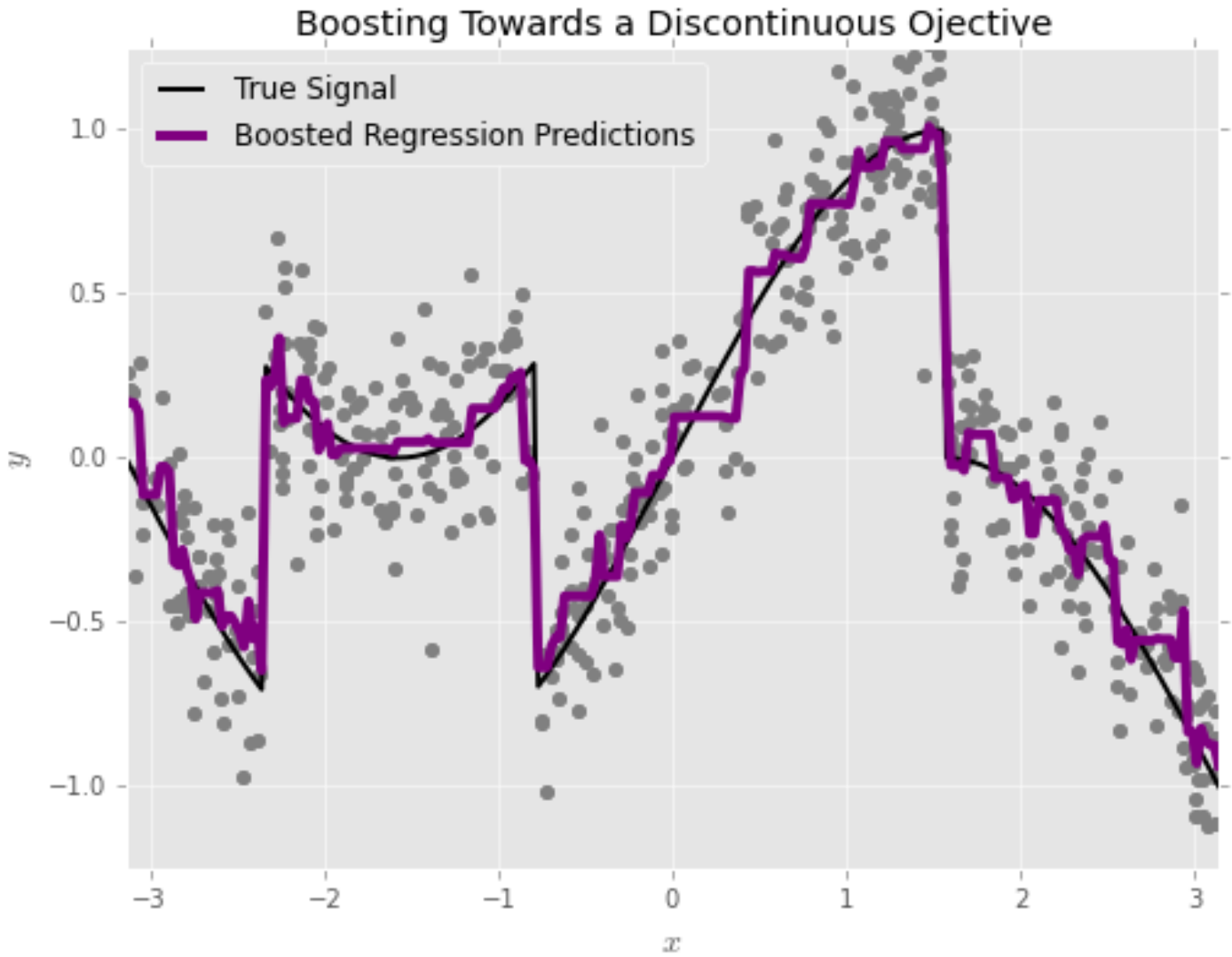
Boosting can adapt itself effortlessly to very non-linear objectives



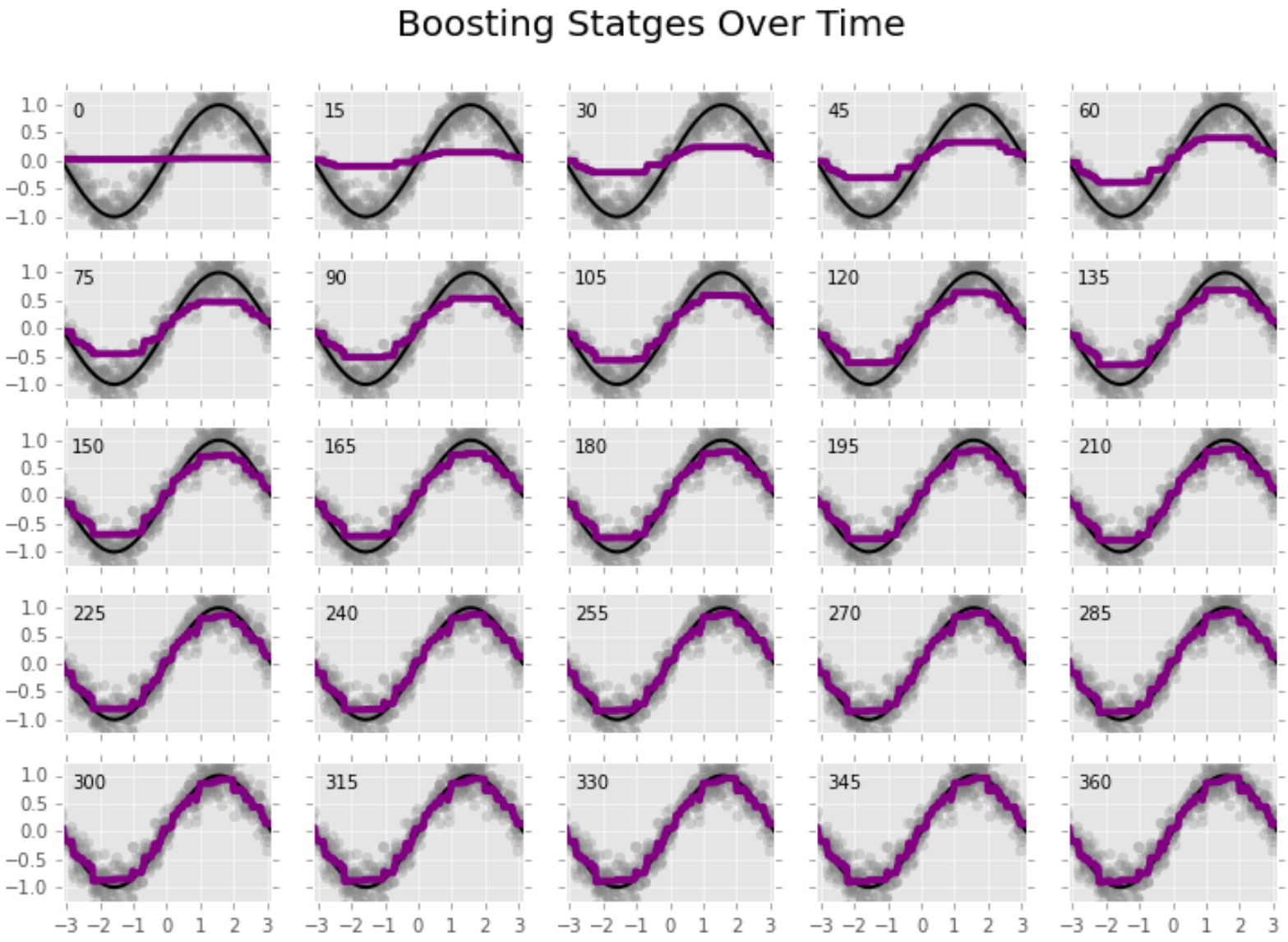
Boosting can adapt itself effortlessly to very non-linear objectives



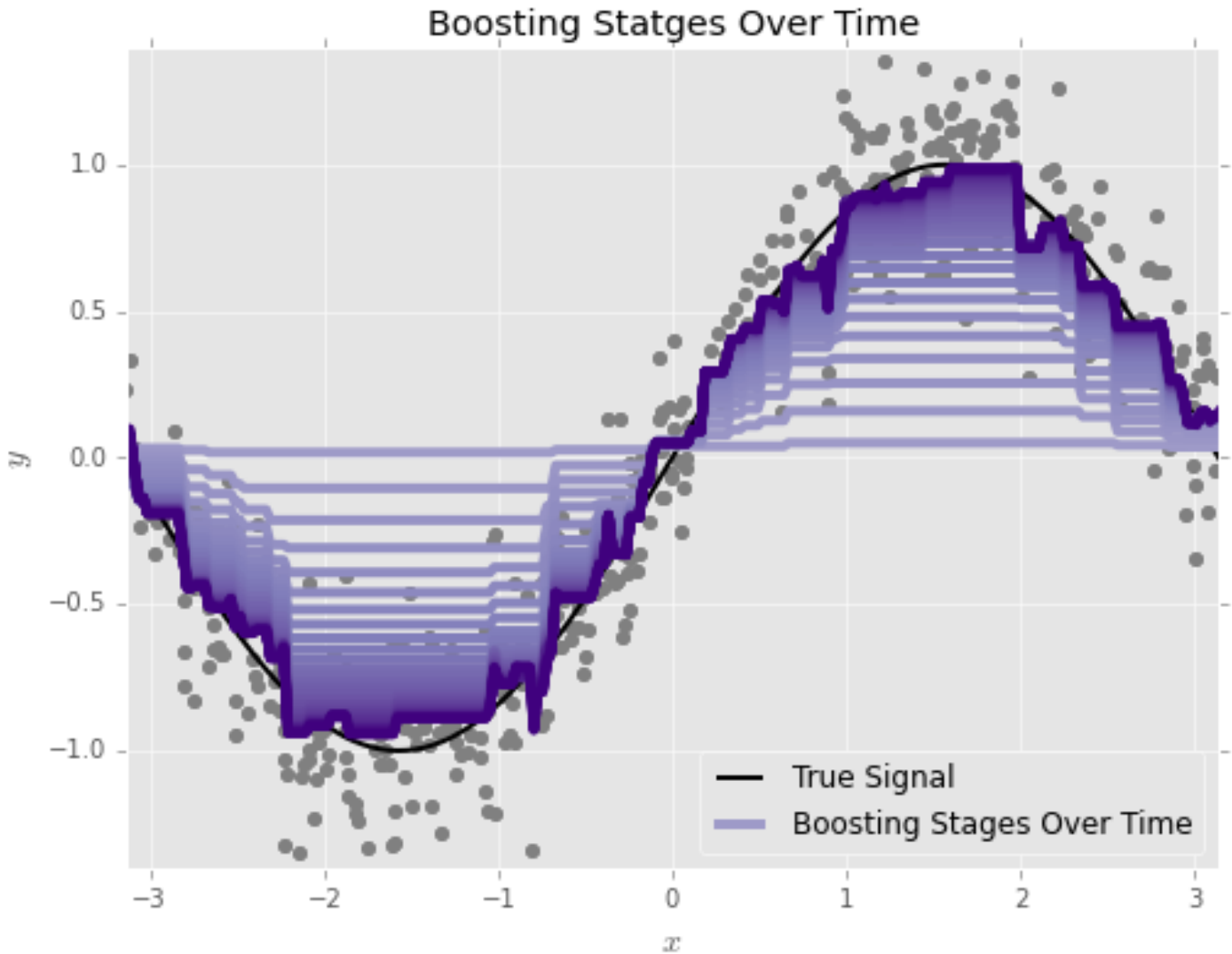
Boosting can adapt itself effortlessly to very non-linear objectives



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



Compared to:

Linear Models:

Lowers variance by using a regularization penalty.
Lowers bias by including more predictors.

Compared to:

Random Forest:

Lowers variance by growing many learners on different subsamples of the data and predictors, then averaging them.
Lowers bias by growing really big trees.

Compared to:

SVM:

Lowers variance by maximizing the margin between positive and negative cases.
Lowers bias by using a kernel, which projects the data into a very high dimensional space.

Boosting:

Lowers variance by growing the model slowly over time (along with a few other tricks).
Lowers bias by stacking many non-parametric models into the final result.

Outline of the Lesson

Agenda:

- Introduction
- You Could Have Invented Gradient Boosting
- Practical Gradient Boosted Regression
- Interpreting Gradient Boosted Regression
- Boosting Other Loss Functions
- Drawbacks of Boosting

Objectives:

- Understand the conceptual foundation of Boosting
- Understand the algorithm's 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 usual 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

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

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 samples.

"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 $L(x)$.

How does this go?

Algorithm: Gradient Descent to Minimize a function L.

Inputs: A function f.
Outputs: A point x_{optim} that minimizes f.

- Compute $\nabla L(x)$ somehow, on paper is good.
- Initialize $x_0 = 0$ (arbitrary, there may be more principled choices).
- Iterate until convergence:
  - Set $x_{i+1} = x_i - \nabla L(x_i)$.

Let's focus on a *single point in our domain*, and try to minimize 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.

Initialize f to the average value of y_i

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

Compute the gradient with respect to f by hand

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

...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.

Question:

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 a constant function

$$f_0(x) = \frac{1}{N} \sum_i y_i \text{ for all } x \in \mathbb{R}^n$$

The **update step fails**.

We cannot evaluate the gradient at any point where we have not observed a value of y.

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

We need to extend the gradient to points where we have not observed a value of y.

Fundamental Idea Of Gradient Boosting:

Replace the response y in our dataset with the **gradient of the loss evaluated at** y

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

Then, fit a model to this new **working response**.

The **predictions from this model** can be viewed as an (approximate) 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:

Comments:

- We didn't *have* to use decision trees, literally anything would work.

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.

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
```

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, used heavily in the plots to
      follow.
```

`GradientBoostingRegressor` has many knobs to turn.

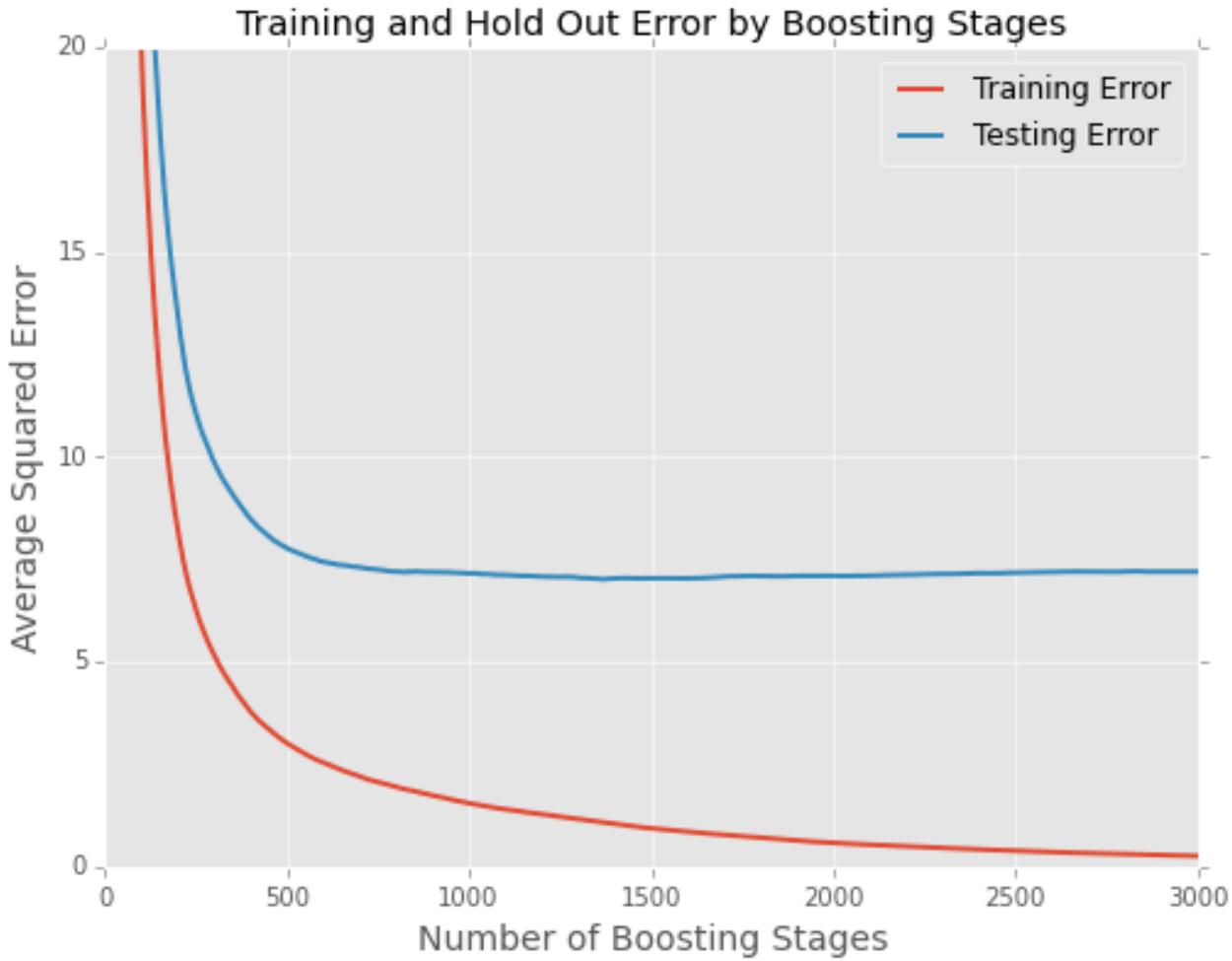
```
GradientBoostingRegressor(loss='ls',
                          n_estimators=100,
                          learning_rate=0.1,
                          max_depth=3,
                          subsample=1.0,
                          min_samples_split=2,
                          min_samples_leaf=1,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

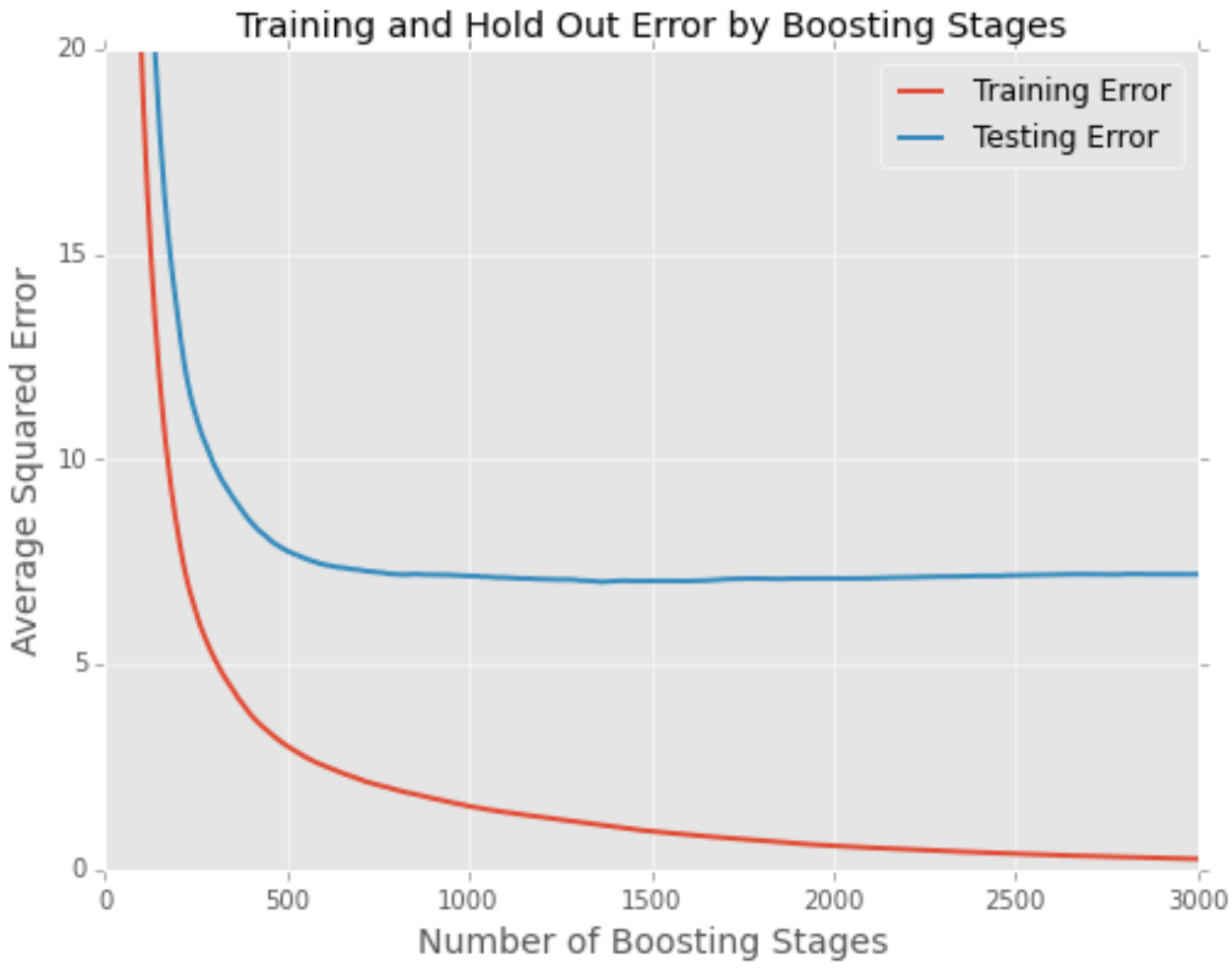
The most important options to `GradientBoostedRegressor` are

- `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

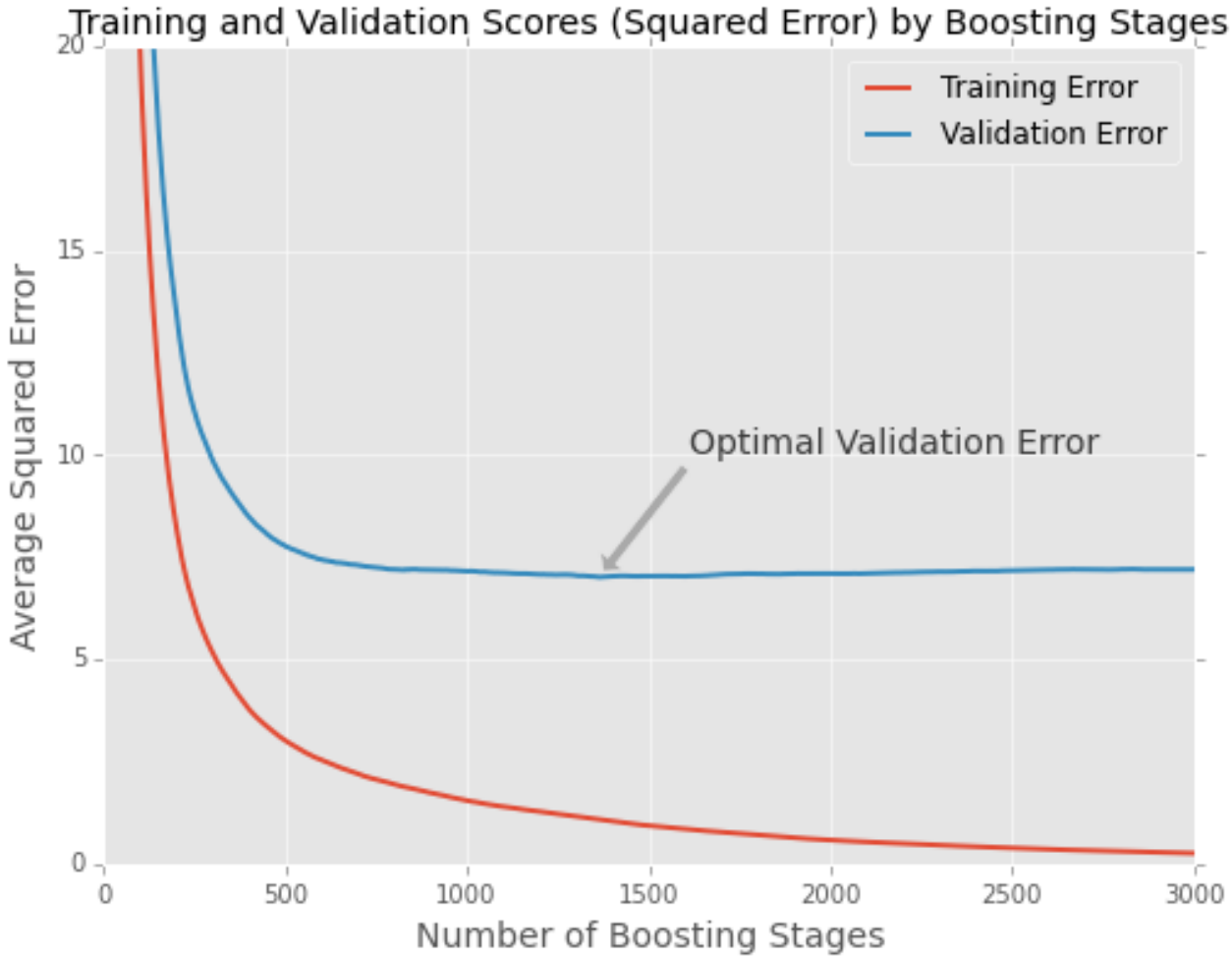
As more and more trees are added to the model the training set error will be driven down monotonically, but the same is not true for the testing error





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

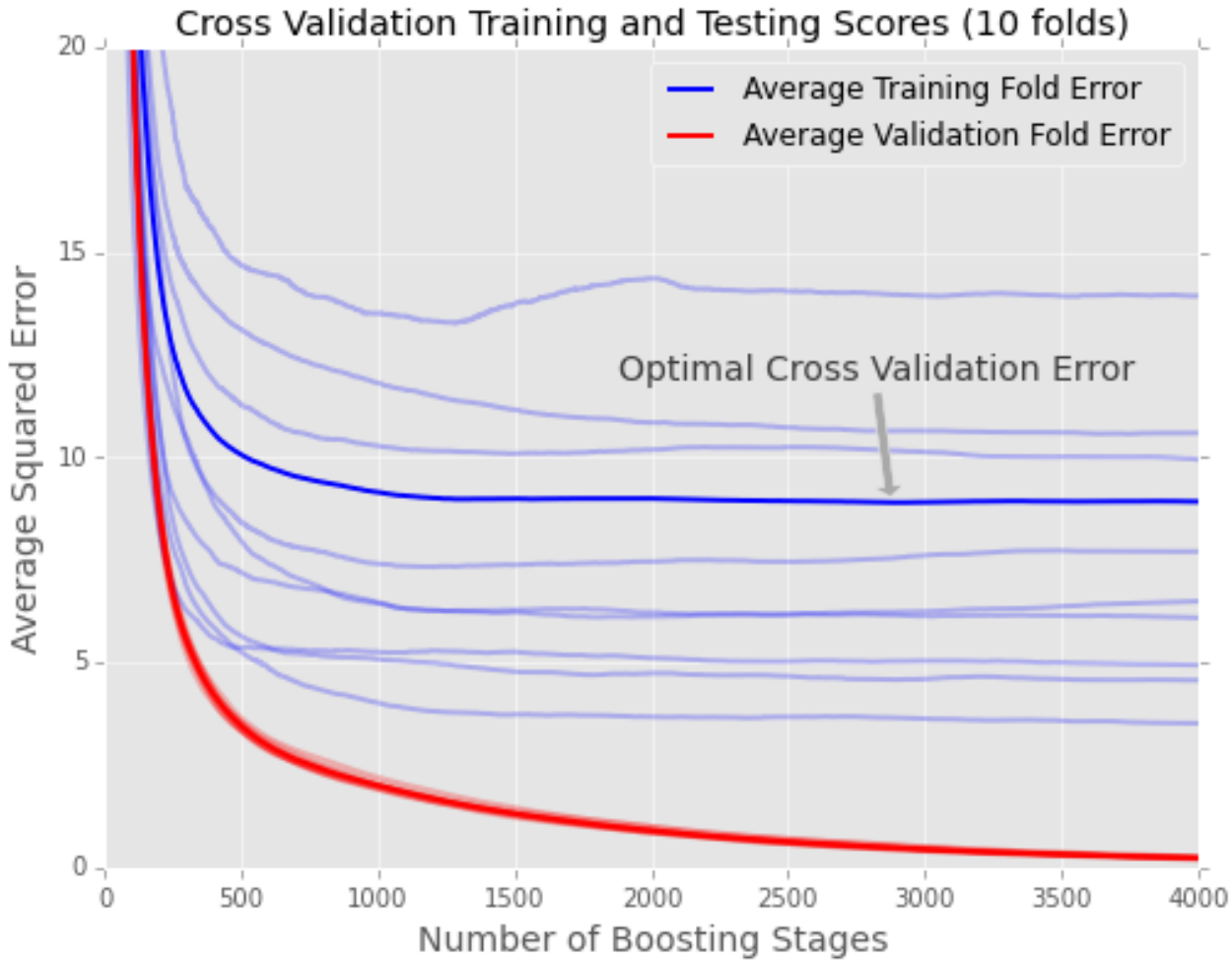


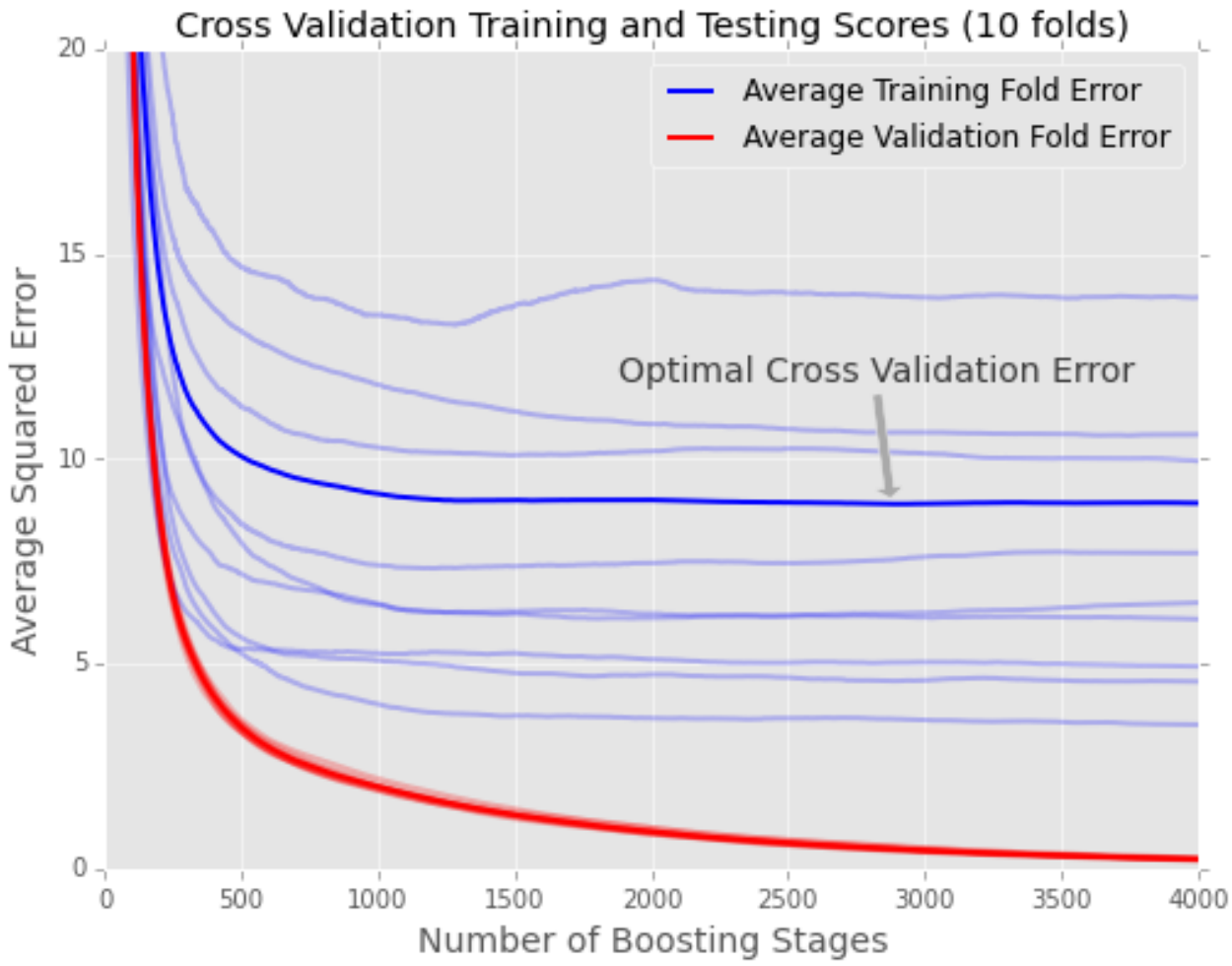
The `loss_` method is important here, it allows us to compute the loss function on held out data

```
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.

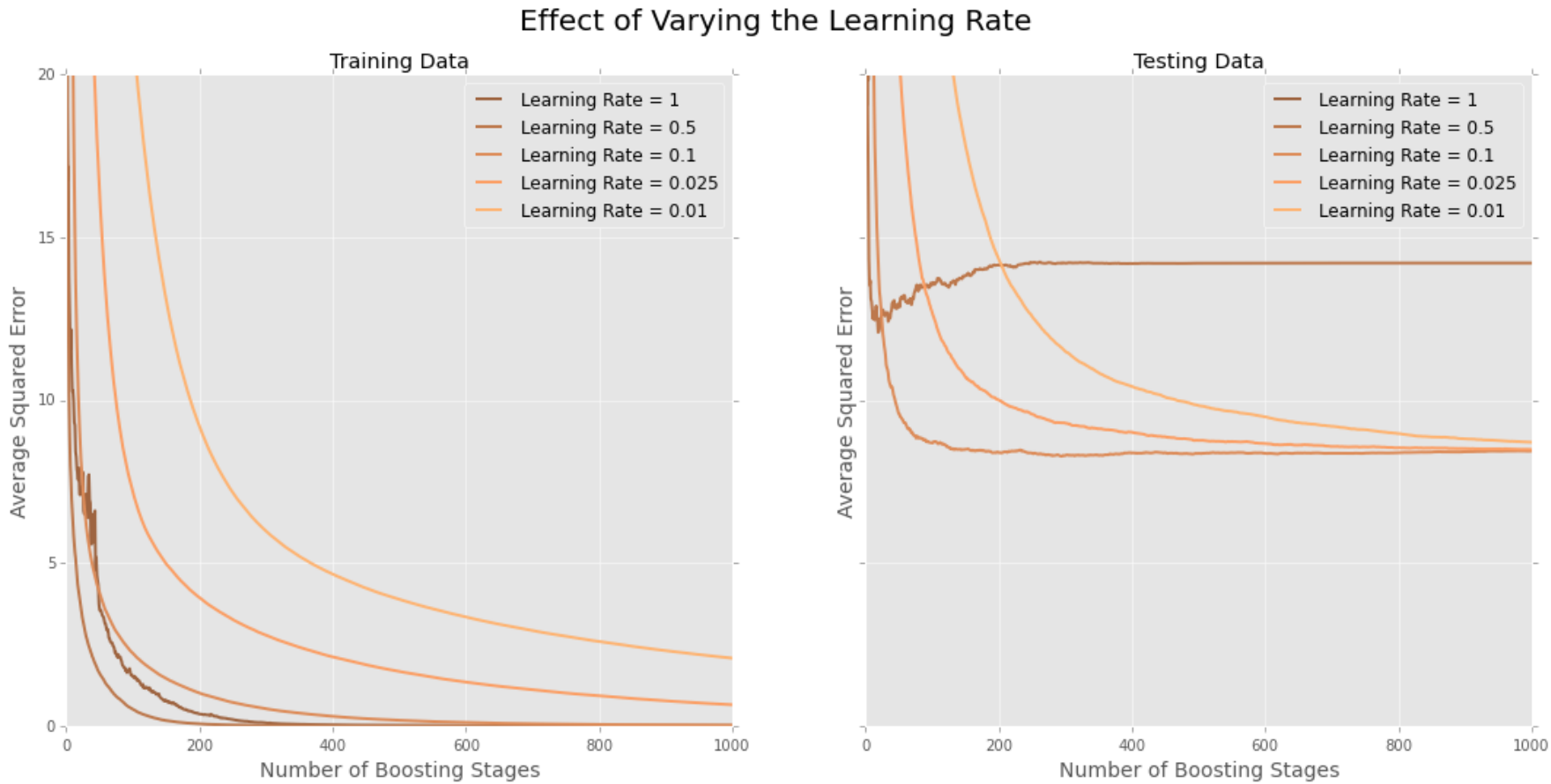
Tuning the Learning Rate

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

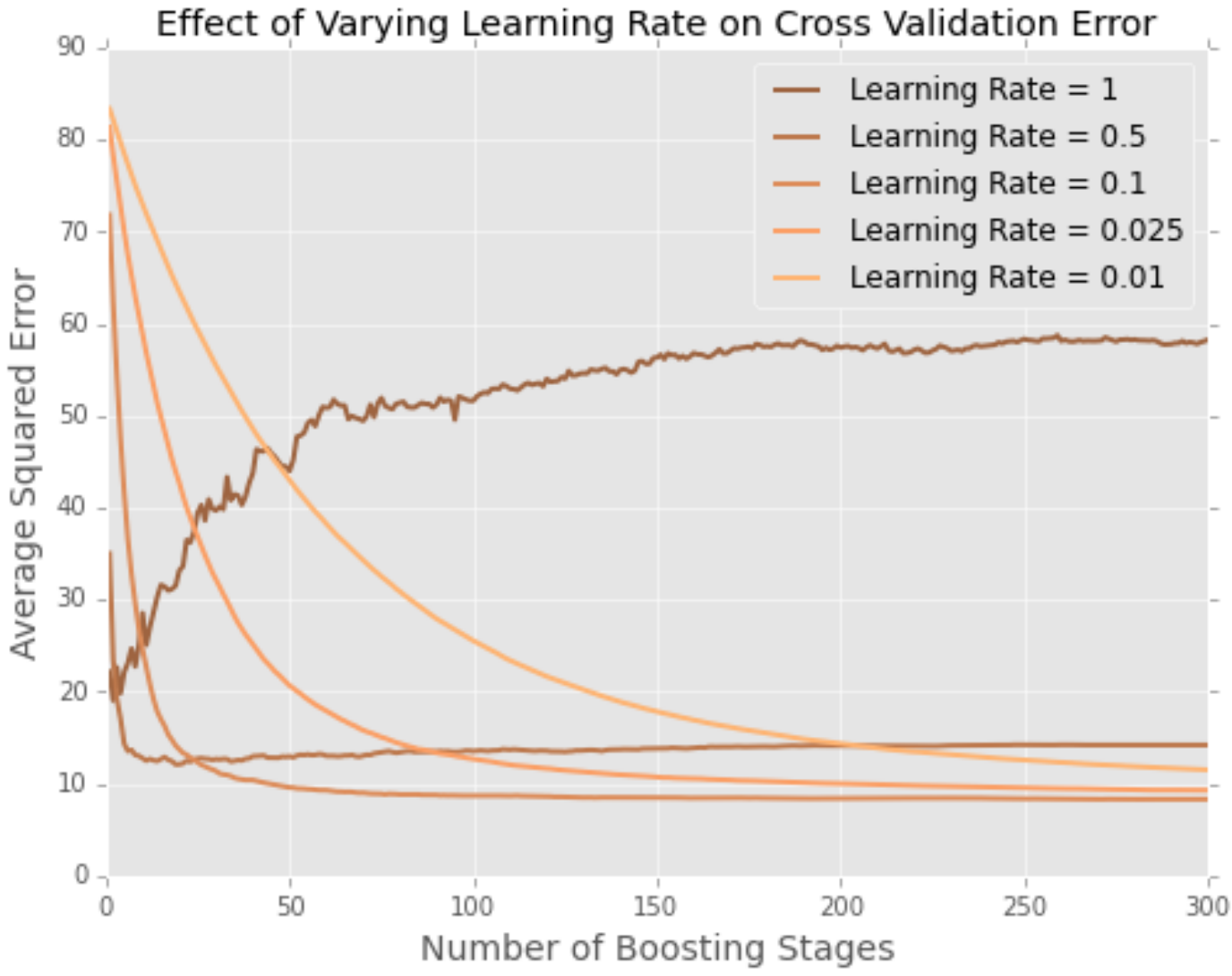
A large learning rate will cause the model to fit hard to the training data, which creates a *high variance* situation.

A smaller learning rate reduces the boosted models sensitivity to the training data.

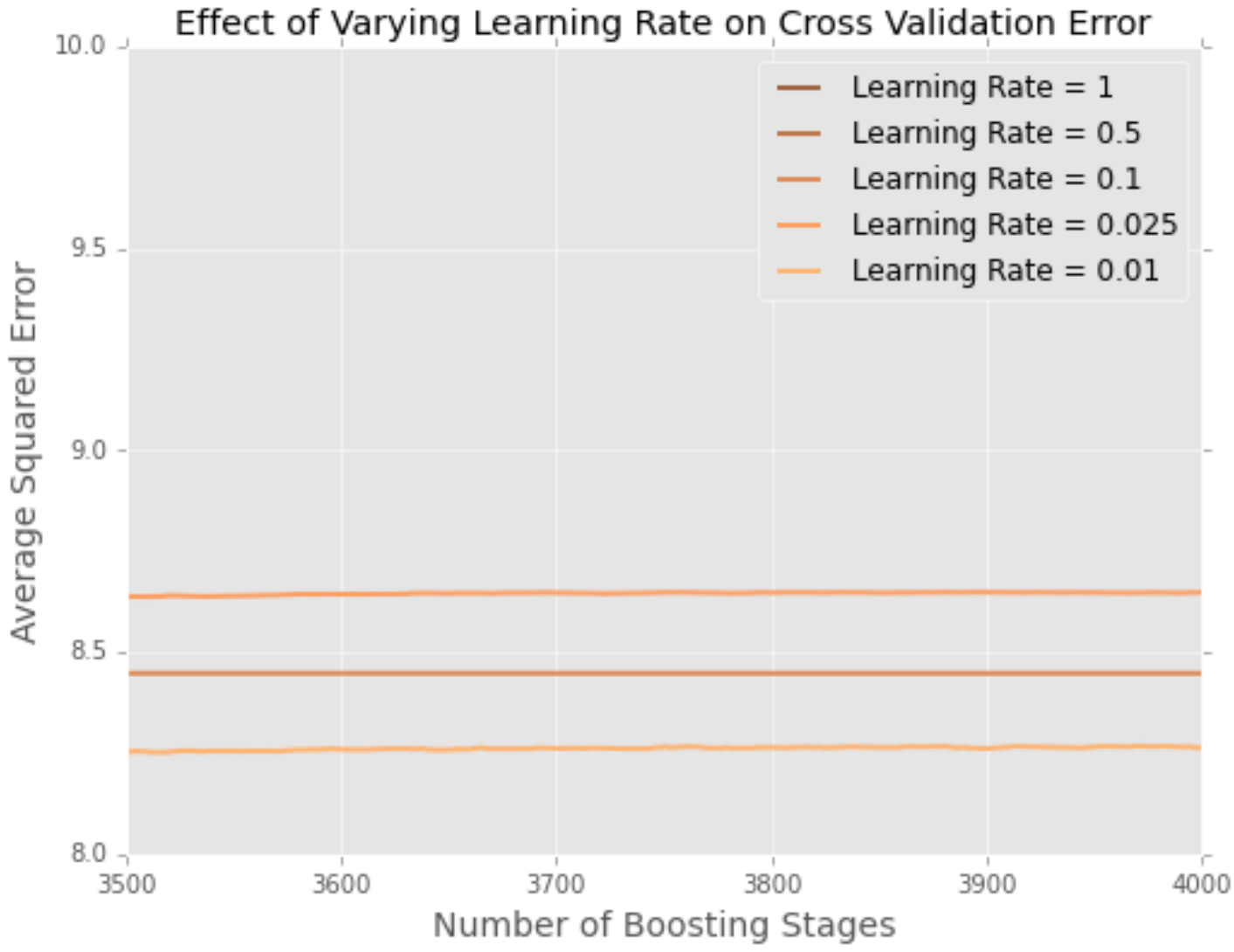
Decreasing the learning rate reduces how fast the booster drives down the training error rate



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



In general, a smaller learning rate eventually (over enough time) results in a superior model



Strategy for Learning Rate:

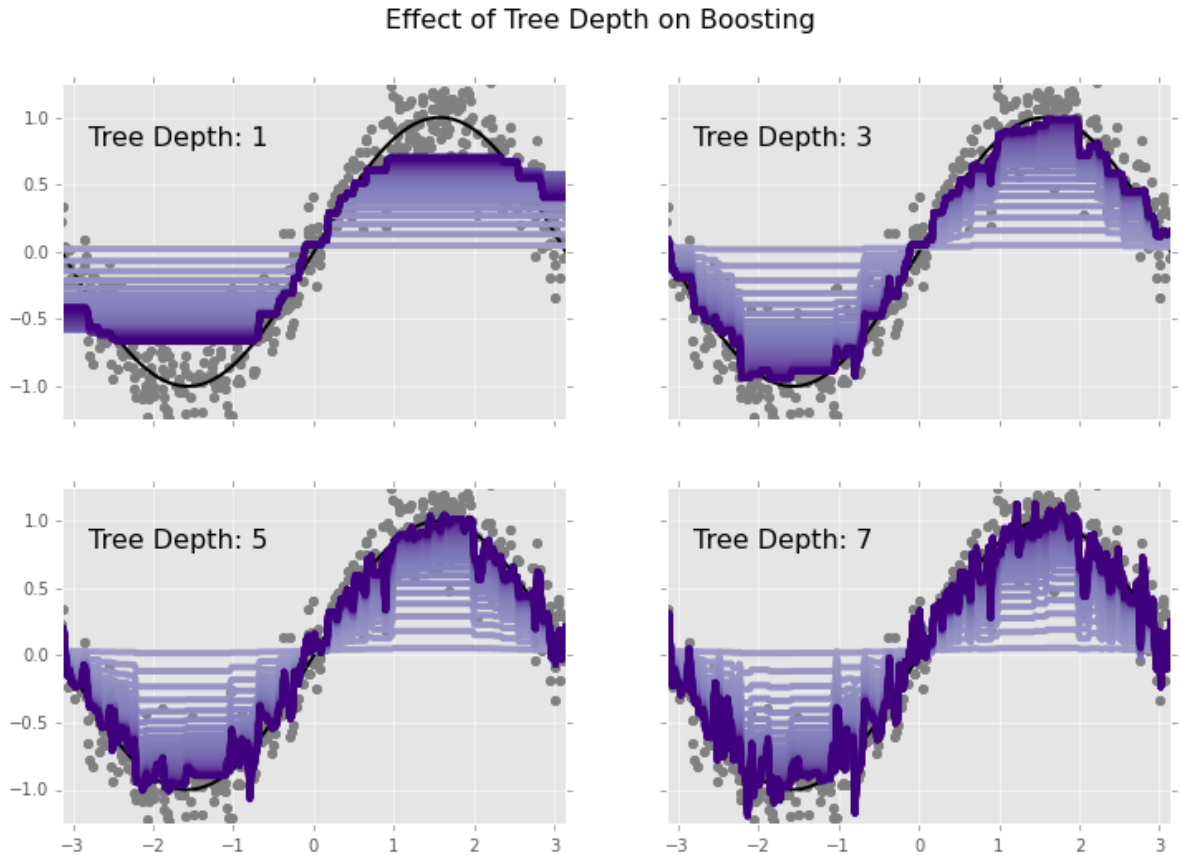
- In the initial exploratory phases of modeling, set the learning rate to some large value, say 0.1. This allows you to iterate through ideas quickly.
- When tuning other parameters using grid search, decrease the learning rate to a more sensible value, 0.01 works well.
- When fitting the *final* production 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 plots and statistics.

Tuning the Tree Depth

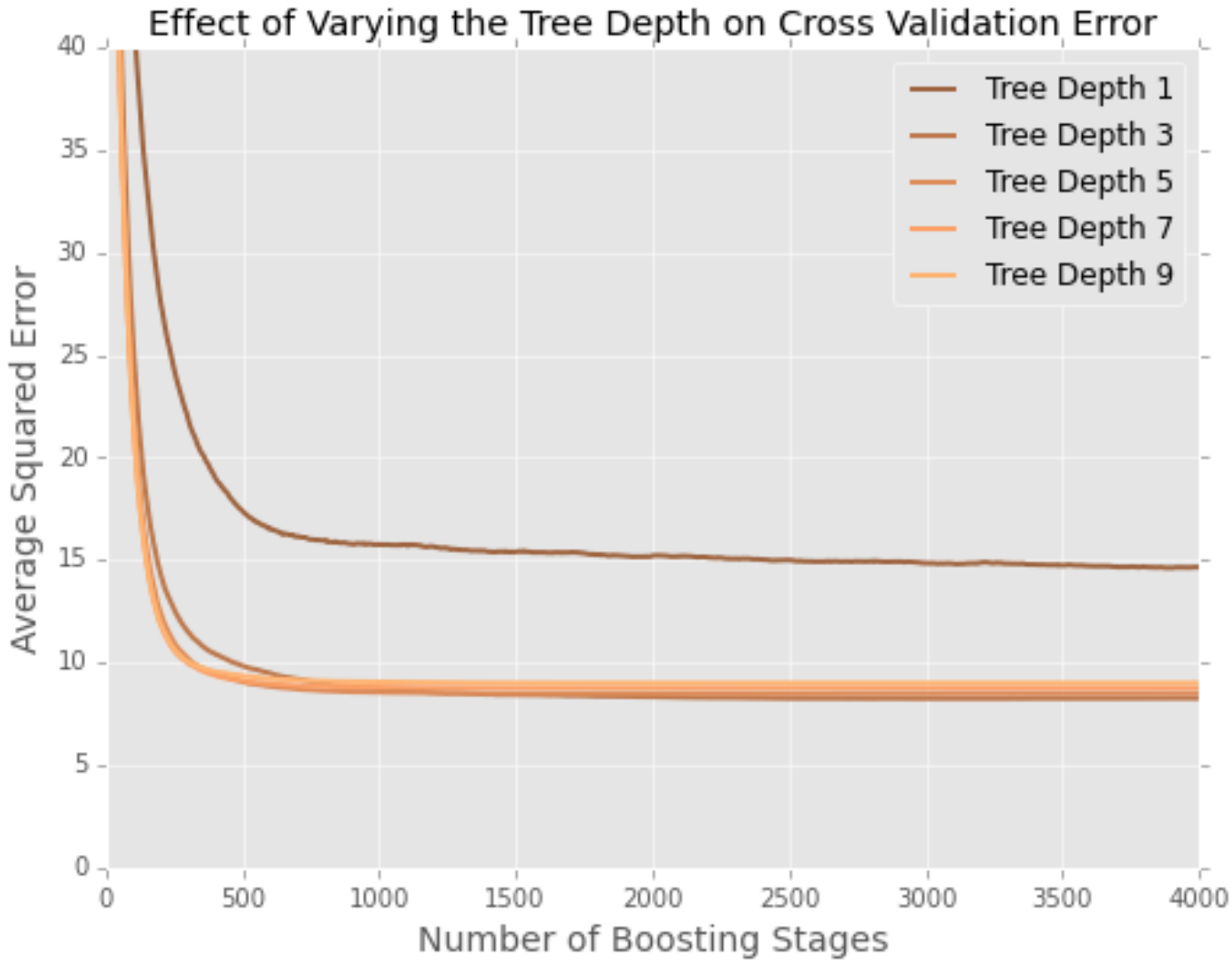
A larger tree depth allows the model to capture deeper interactions between the predictors, resulting in lower bias.

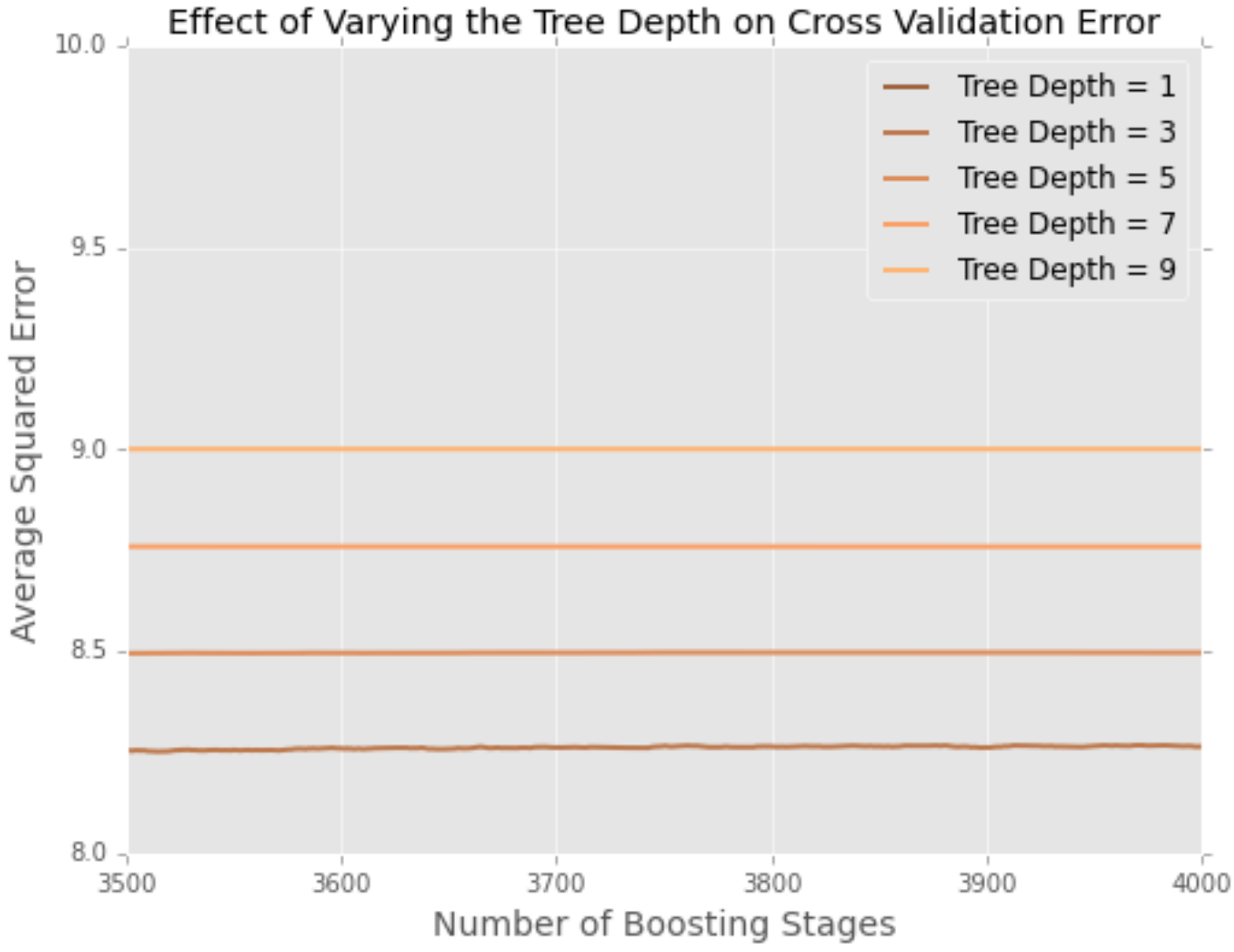


Unfortunately, a deeper tree depth also

- Causes the model to fit faster, increasing the variance and somewhat combating the effect of the learning rate.
- Allows the model to assume a more complex for at the same number of trees. This is a blessing (low bias) and curse (high variance).

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




Effect of Varying the Tree Depth on Cross Validation Error
Tree Depth = 1
Tree Depth = 3
Tree Depth = 5
Tree Depth = 7
Tree Depth = 9
Average Squared Error
10.0
9.5
9.0
8.5
8.0
3500
3600
3700
3800
3900
4000
Number of Boosting Stages

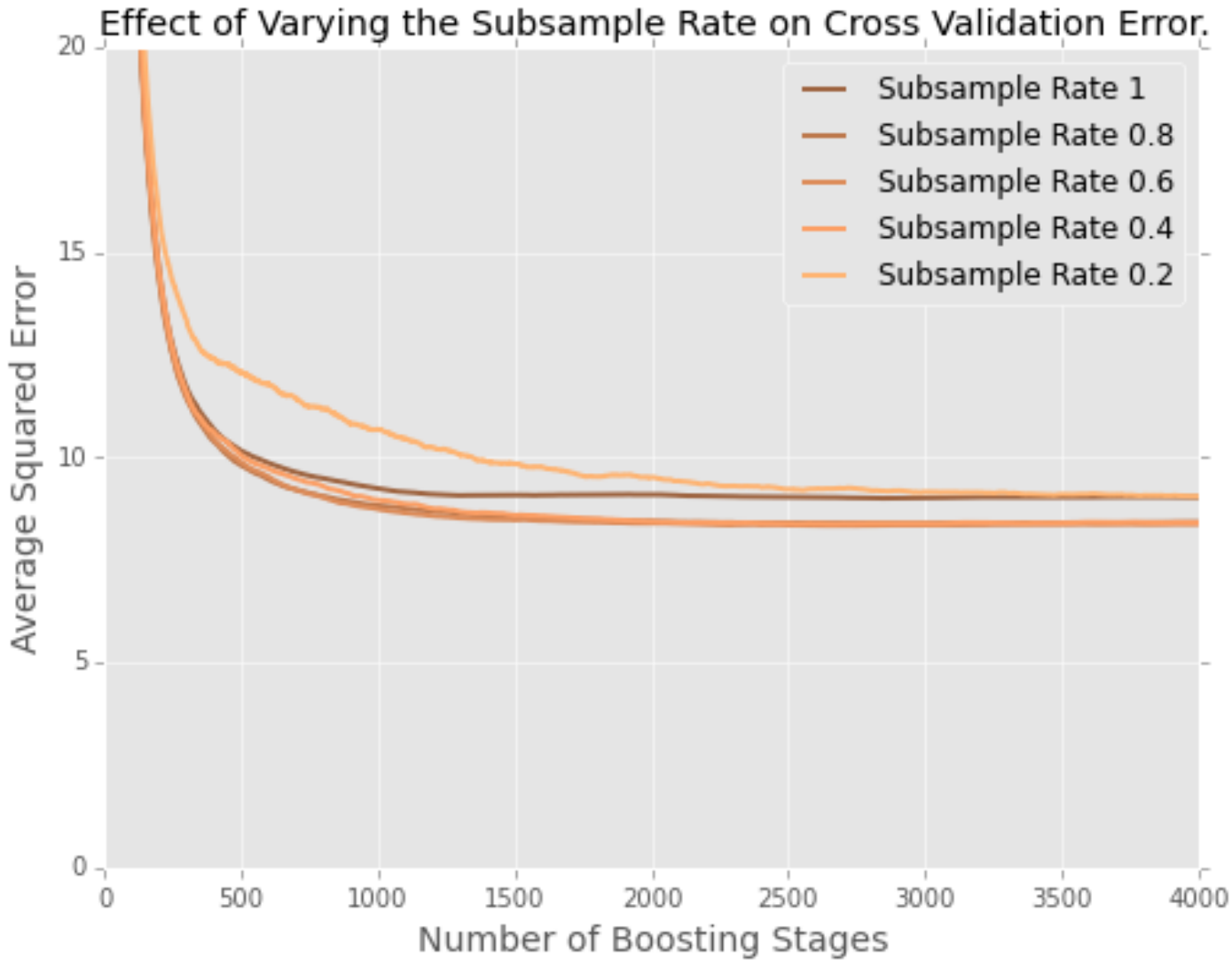
Strategy for Tree Depth:
Tune with a grid search and cross validation.

Tuning the Subsample Rate

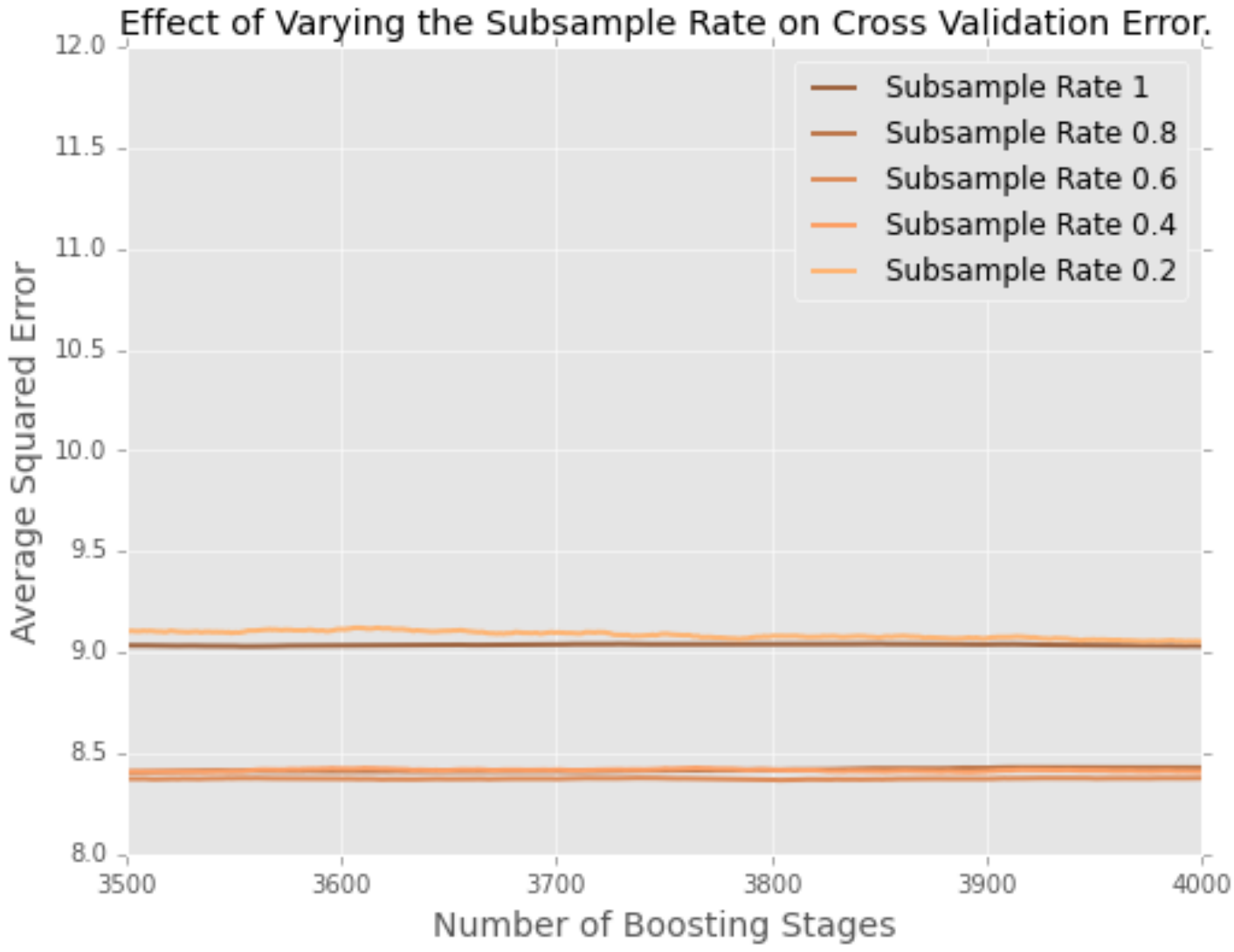
The `subsample` parameter allows one to train each tree on a subsample of the training data.

This is similar to bagging in the random forest algorithms, and has the same result: it lowers the variance of the resulting model.

Not subsampling at all, or over subsampling are both bad ideas



Between these two extremes, different levels of subsampling generally give the same performance



Strategy For Subsample:

Set to 0.5, it almost always works well.

Tuning Other Gradient Boosting Parameters

The other parameters to `GradientBoostingRegressor` are less important, but can be tuned with grid search for additional improvements in importance.

- `min_samples_split`: Any node with less samples than this will not be considered for splitting.
- `min_samples_leaf`: All terminal nodes must contain more samples than this.
- `min_weight_fraction_leaf`: Same as above, but a expressed as a fraction of the total number of training samples.

Generally these are less important because *you shouldn't be growing super gigantic trees*!

If you *do* decide to include these in a grid search, *be wary of overfitting to your validation set*.

The number of model comaprisons grows exponentially in the number of parameters tuned.

Max Features

There's one more parameter worth mentioning

`max_features`: The number of features to consider for each split, as in random forest.

Thought Exercise: Think about how varying this parameter will influence the model. Run some experiments to see if you're right. Should you include this in a grid search?

Interpreting Gradient Boosting

Gradient boosting models, while offering massive predictive power, are very complex and hard to interpret.

There are two high level summarization techniques that are very popular, and can help understand the high level content of the model and diagnose issues

- **Relative Variable Importance**: Measures the amount a predictor "participates" in the model fitting procedure.
- **Partial Dependence Plots**: Are analogous to parameter estimates in linear regressions, they summarize the effect of a single predictor while controlling for the effect of all others.

Relative Variable Importance

The concept here is the same as in random forest.

Relative Variable Importance

The concept here is the same as in random forest.

Each time we grow a tree, we keep track of how much the error metric decreases at each split, and allocate that decrease to a predictor.

Relative Variable Importance

The concept here is the same as in random forest.

Each time we grow a tree, we keep track of how much the error metric decreases at each split, and allocate that decrease to a predictor.

The importance of a predictor `in a tree` is the total amount the error metric decreased over all splits on that predictor

Relative Variable Importance

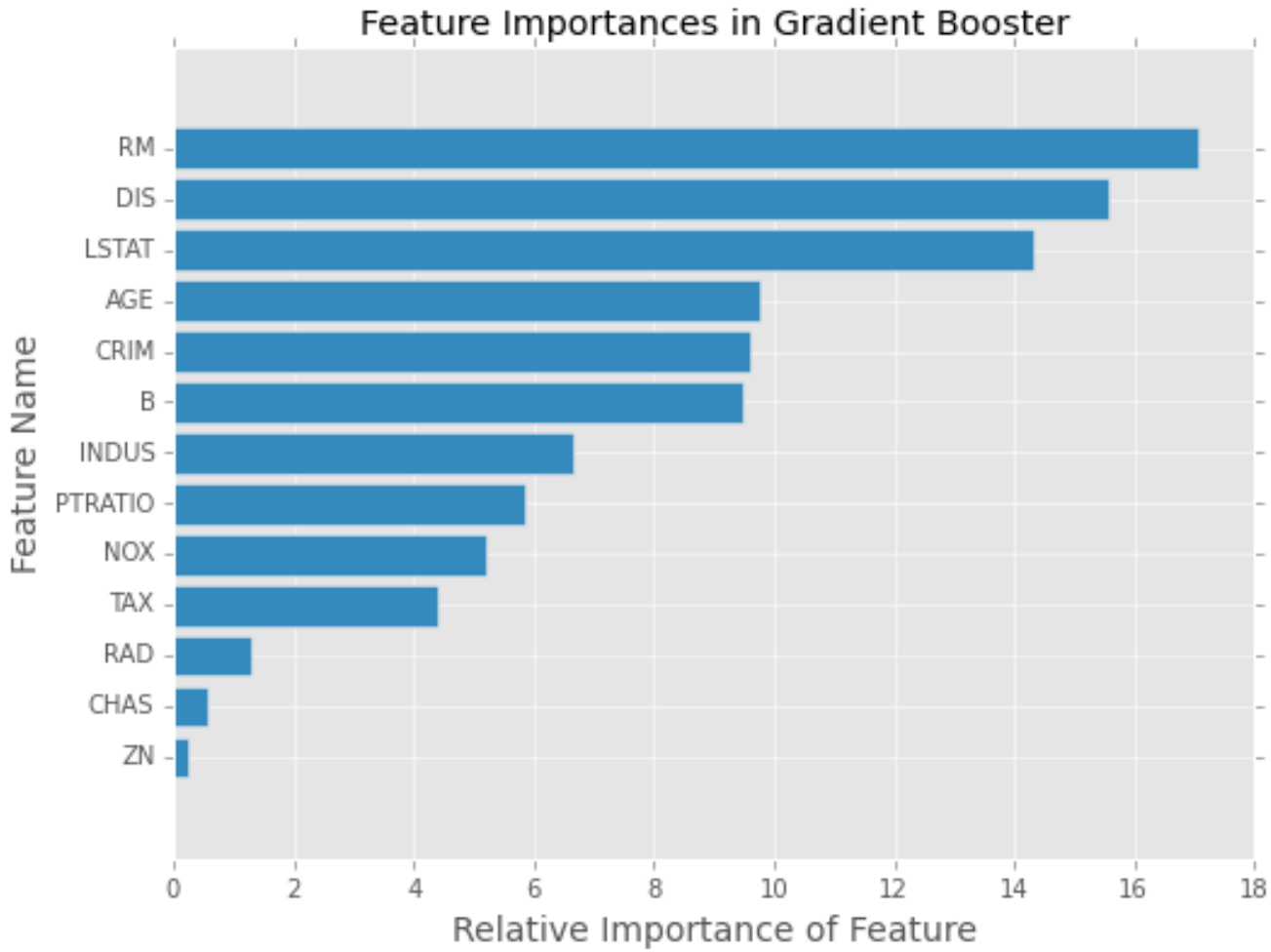
The concept here is the same as in random forest.

Each time we grow a tree, we keep track of how much the error metric decreases at each split, and allocate that decrease to a predictor.

The importance of a predictor `in a tree` is the total amount the error metric decreased over all splits on that predictor

The importance of a predictor in the boosted model is the `average` importance of the predictor over all the trees.

It is traditional to normalize the importances so that they sum to 100.



Comments:

- The name "feature importances" is pretty awful. It invites misinterpretation.

  Don't reason about things from their names, make sure the statistic actually answers your question.

Comments:

- If your model contains both numeric and binary predictors, the importance metric is biased to assign higher values to the numeric predictors (do you see why?). Try not to compare feature importances across these two classes.

Comments:

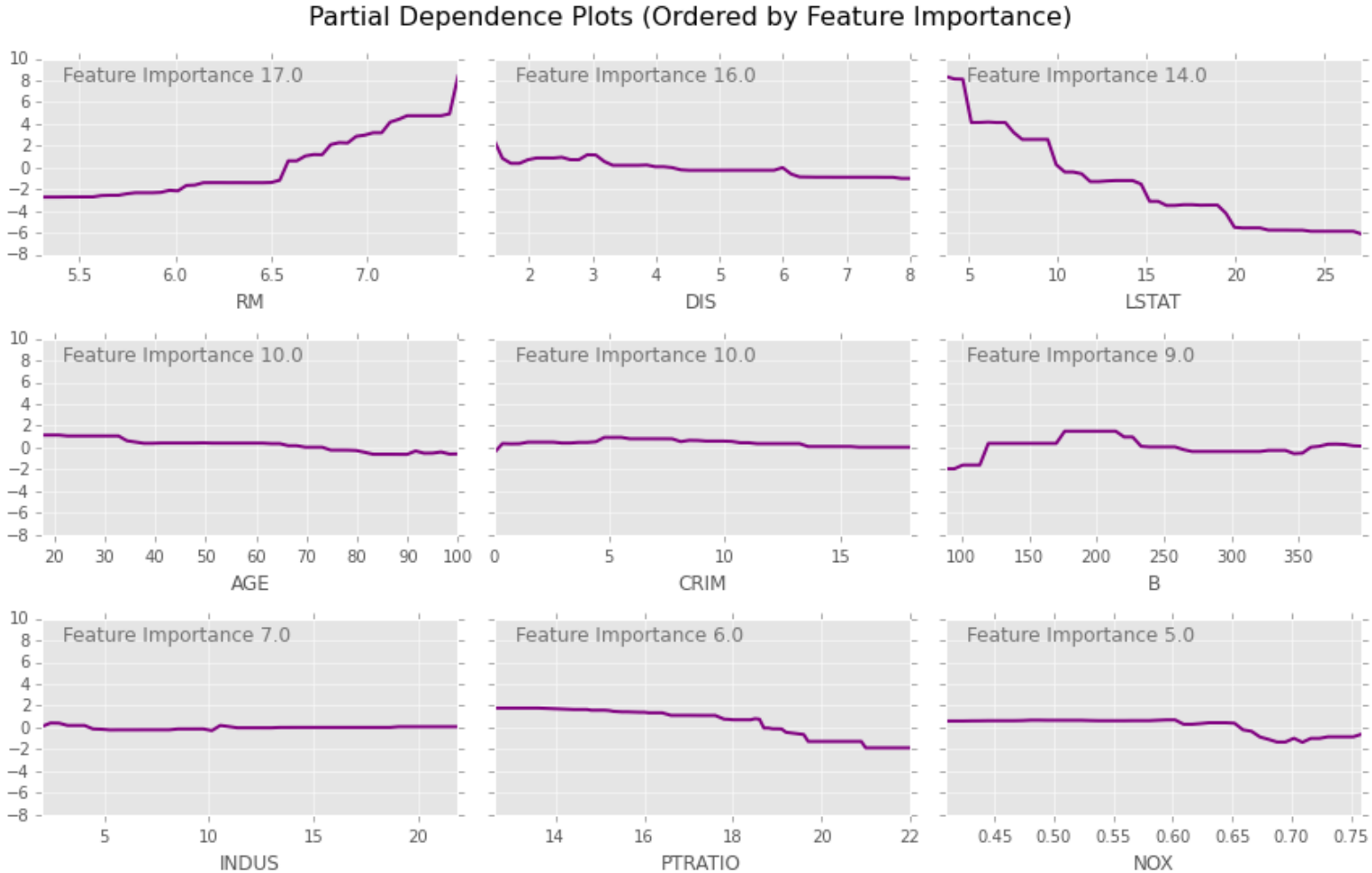
- Feature importance rankings can have very high variance. Make sure any important conclusions are robust to different RNG seeds and training sets.

Comments:

- Make sure your model only includes trees *up to the optimal point*. Otherwise you'll allocate importance to overfitting.

Partial Dependence Plots

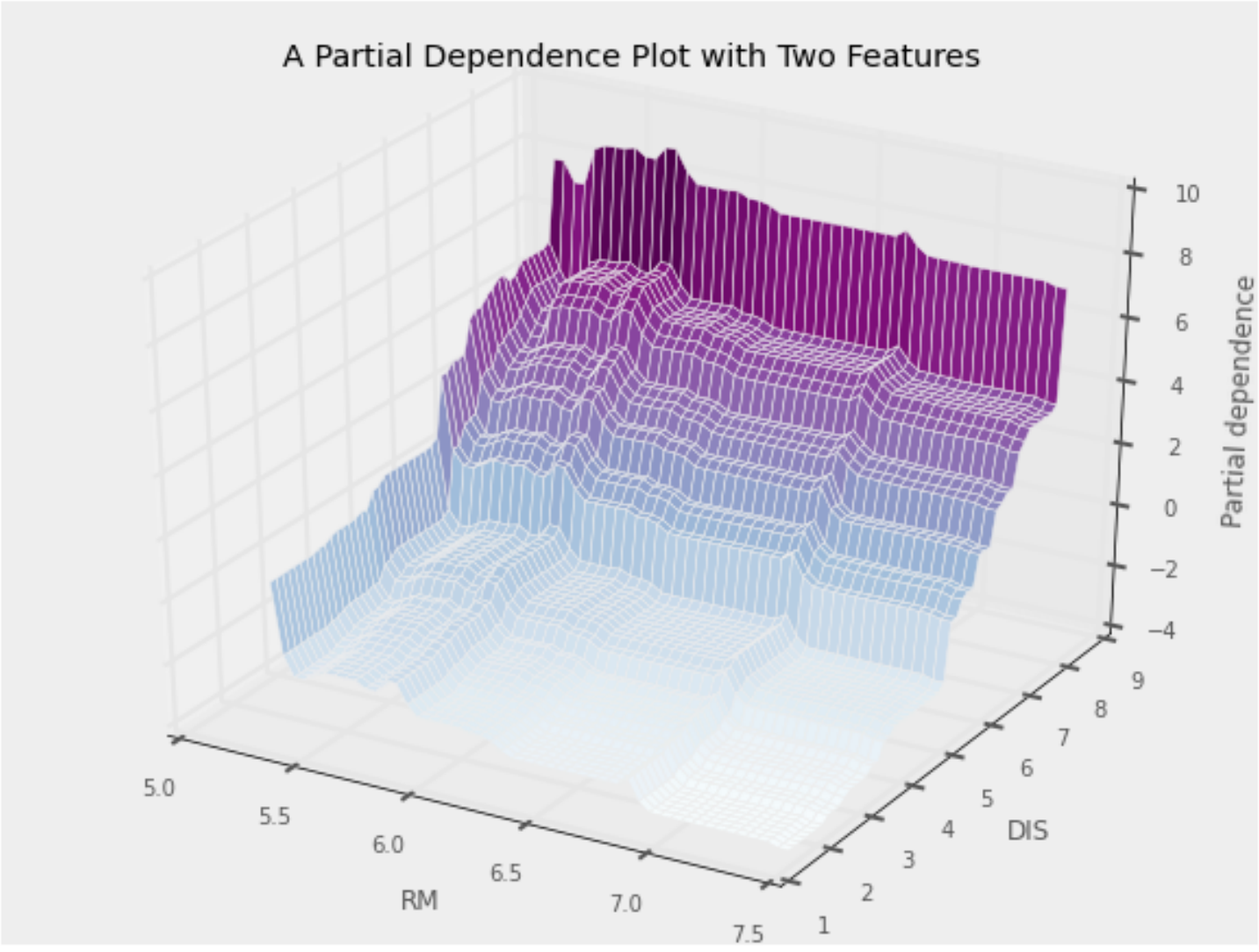
Visualizations of the effect of a single predictor, averaging out the effects of all the rest



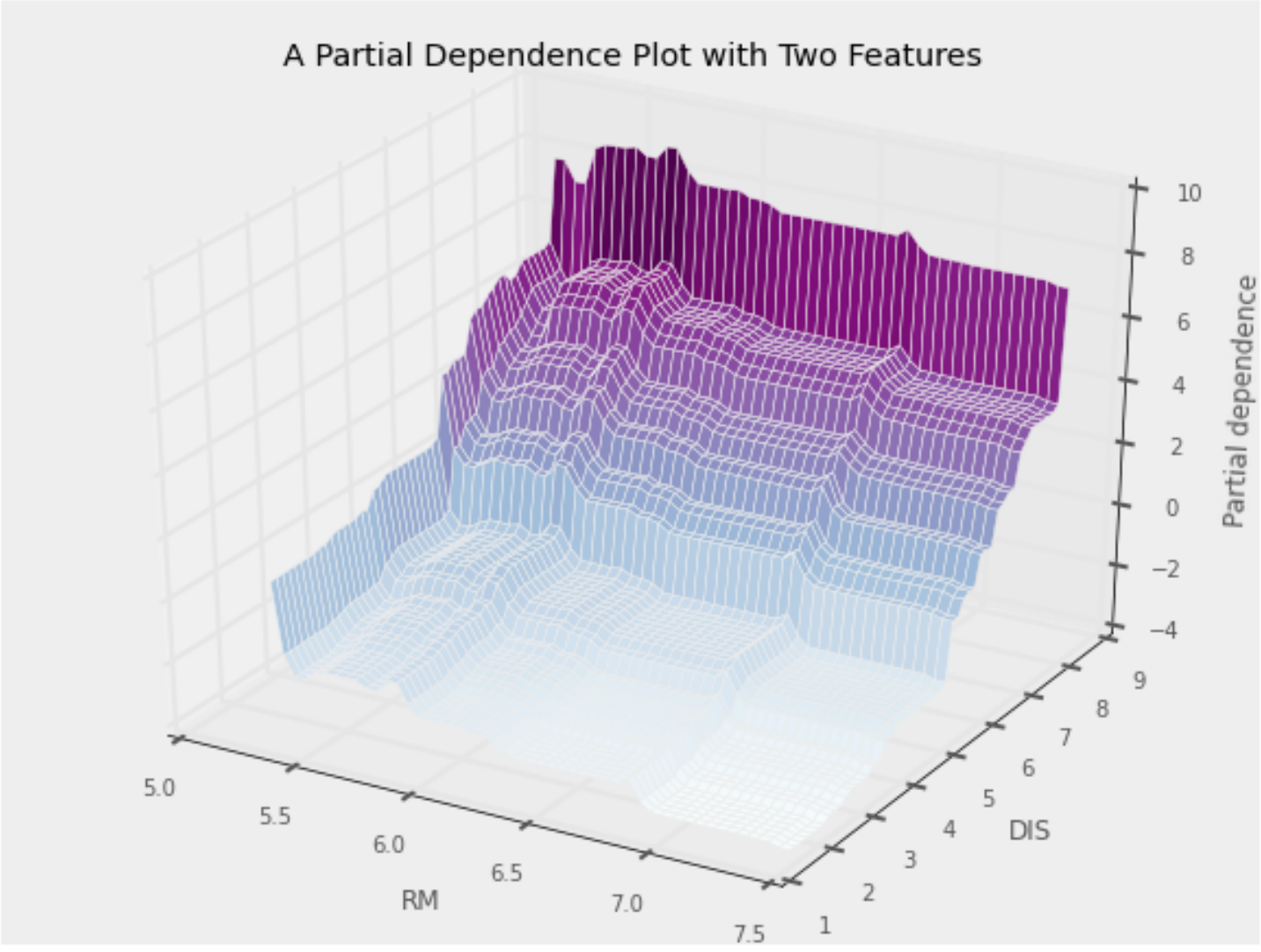
In symbols:

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

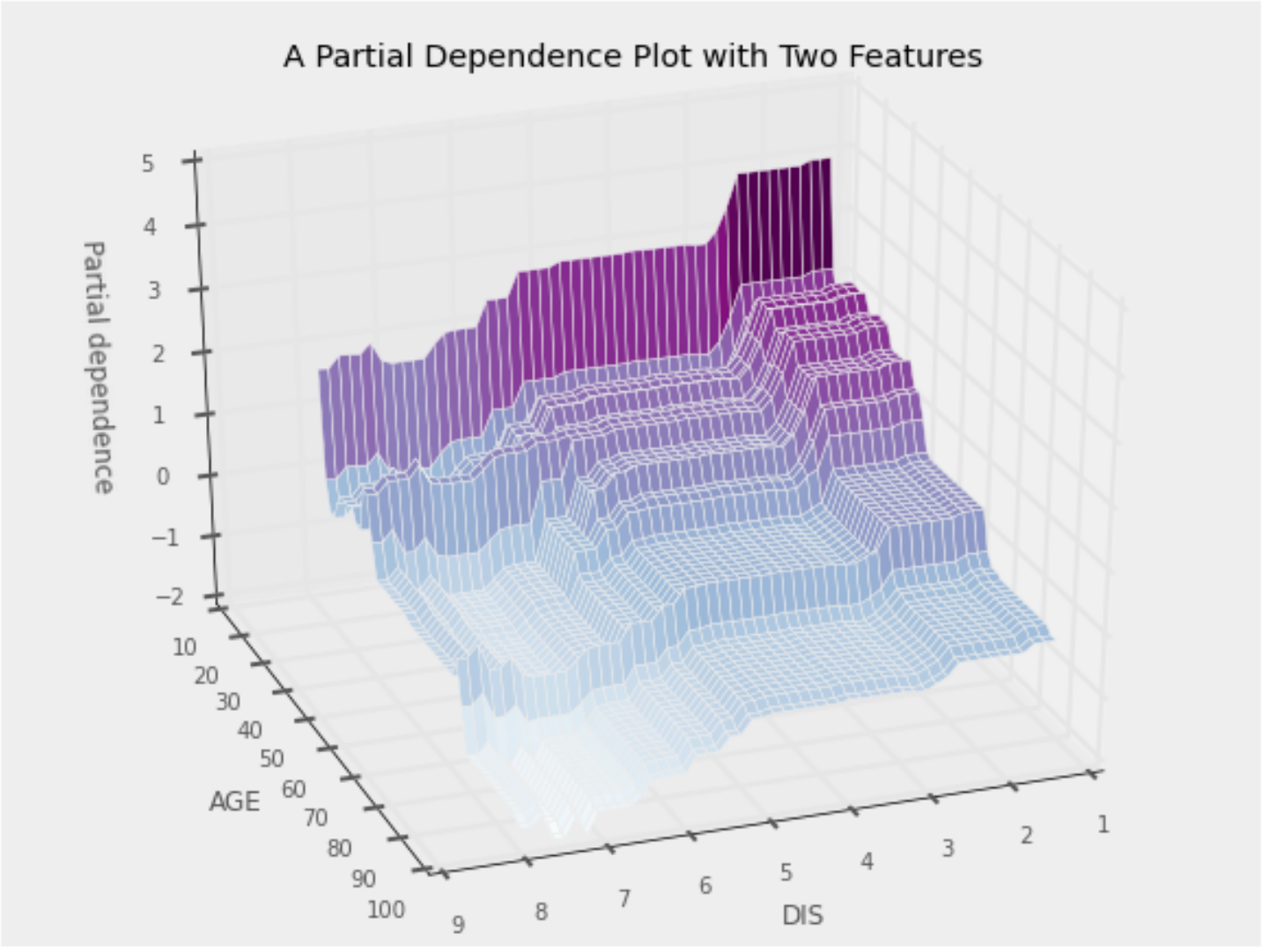
By varying the values of two predictors, we can draw partial dependence plots in higher dimensions



Question: Does it look like any trees split on *both* RM and DIS?



Question: What about AGE and DIS?



Other Gradient Boosting Algorithms

The last great feature of Gradient Boosting is that it generalizes easily to other loss functions.

We will discuss two generalizations:

- **Gradient Boosted Logistic Regression**: Minimizes the binomial deviance (logistic log likelihood) loss function.
- **AdaBoost**: Minimizes a custom classification loss.

It is important to say: *There are many more possibilities*!

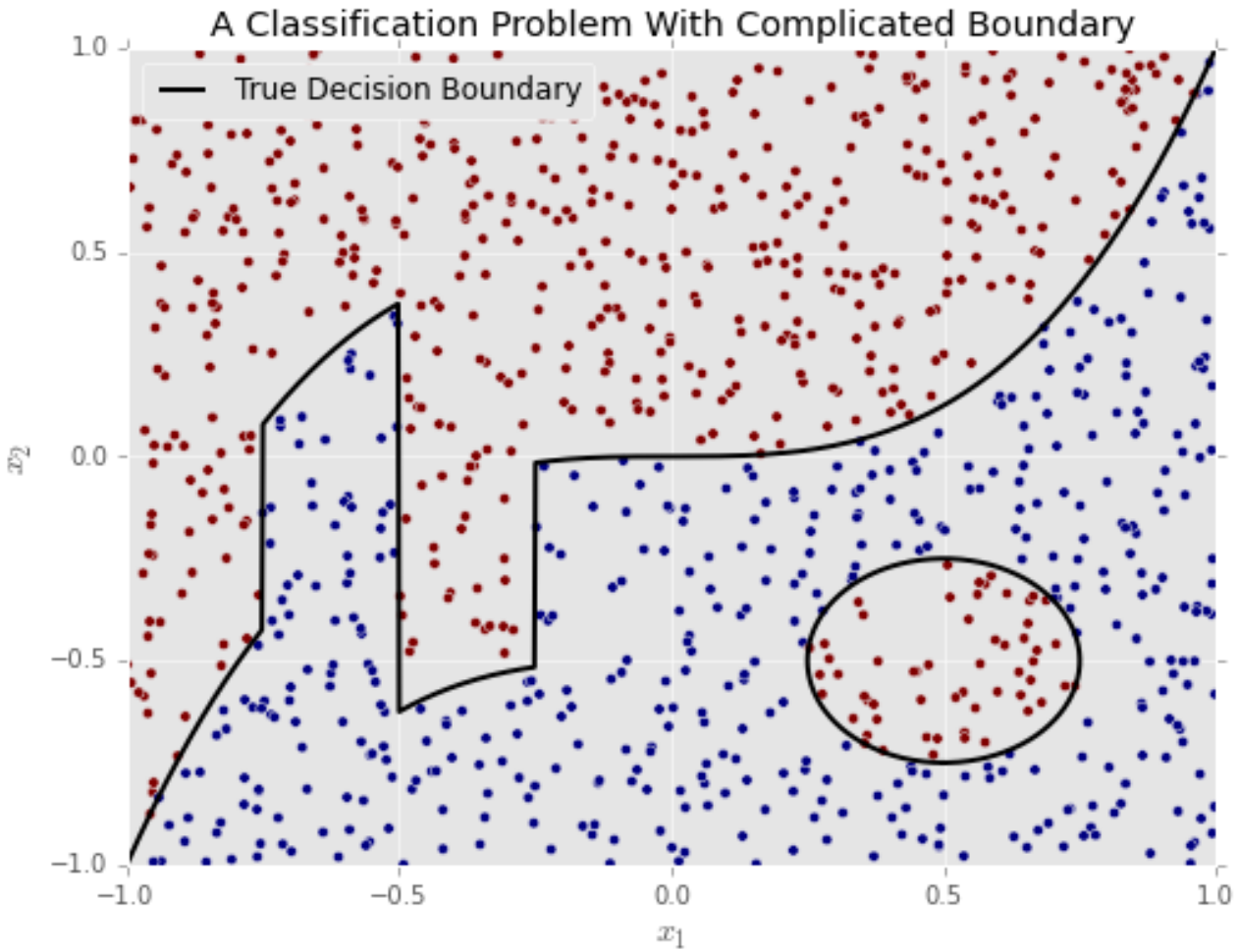
Gradient Boosted Logistic Regression

We want to generalize our boosting algorithm to also solve *classification* problems.

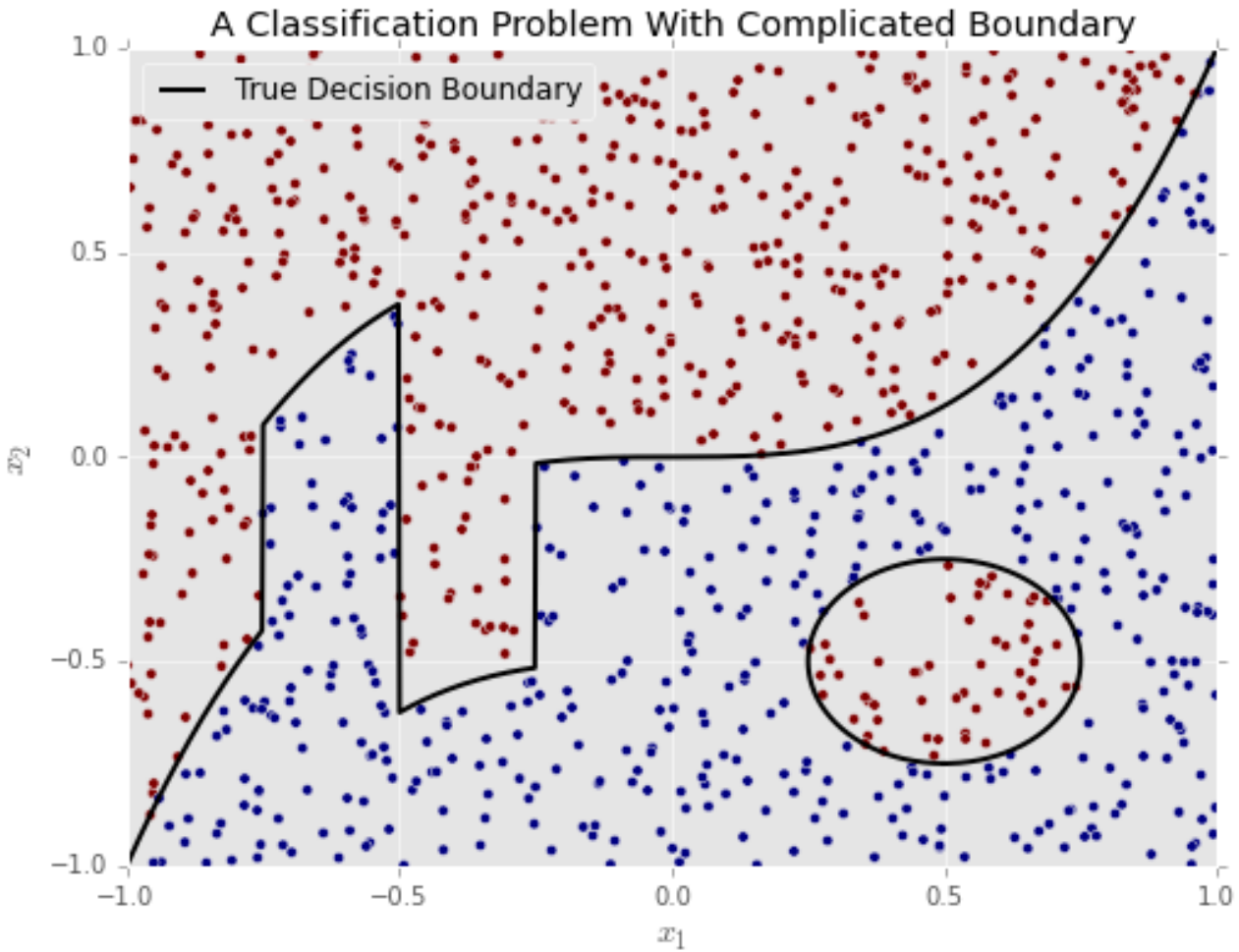
$$y \in \{0, 1\}$$

We want to estimate $f(x) = Pr(y = 1 \mid x)$.

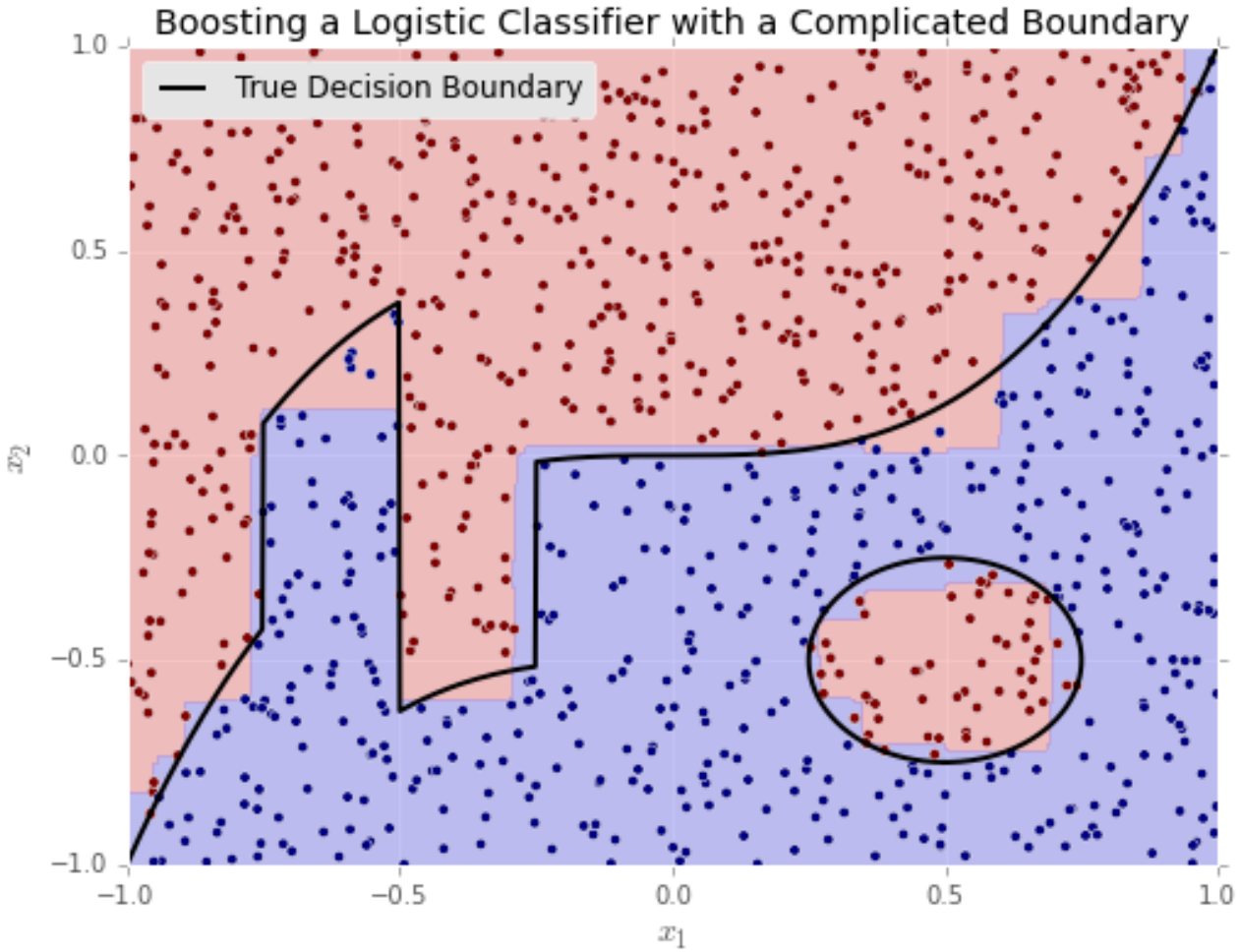
For example, here is a complex classification problem



How would you solve this with standard logistic regression?



Gradient boosted logistic regression makes short work of it.



Recall our friend logistic regression.

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

Where $\nu(\beta, x_i) = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_M x_{iM}$ (called the *linear predictor*).

This can be solved with gradient descent.

Once we have solved for $\hat{\beta}$, we can make predictions using

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

The predictions can interpreted as *the conditional probability that* $y = 1$, *given the values of* x.

The function we are minimizing in logistic regression is called the *logistic loss*:

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

Gradient Boosted Logistic Regression:

Replace the linear predictor in logistic regression

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

With a sum of small regression trees

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

To fit a Gradient Boosted Logistic Regression, replace the least squares loss function

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

With the logistic loss

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

And then use the same gradient boosting technique.

Note: There are some subtleties. I've included the details in an appendix.

Gradient boosted logistic regression is implemented in sklearn as

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

The options to `GradientBoostingClassifier` are the same as those to `GradientBoostingRegressor`

```
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,
                           ...)
```

And everything we said before generalizes.

To make predictions use `predict_proba`

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

The `predict` method returns *class labels* (by comparing the probability to 0.5), making it much less useful.

Gradient Boosted AdaBoost

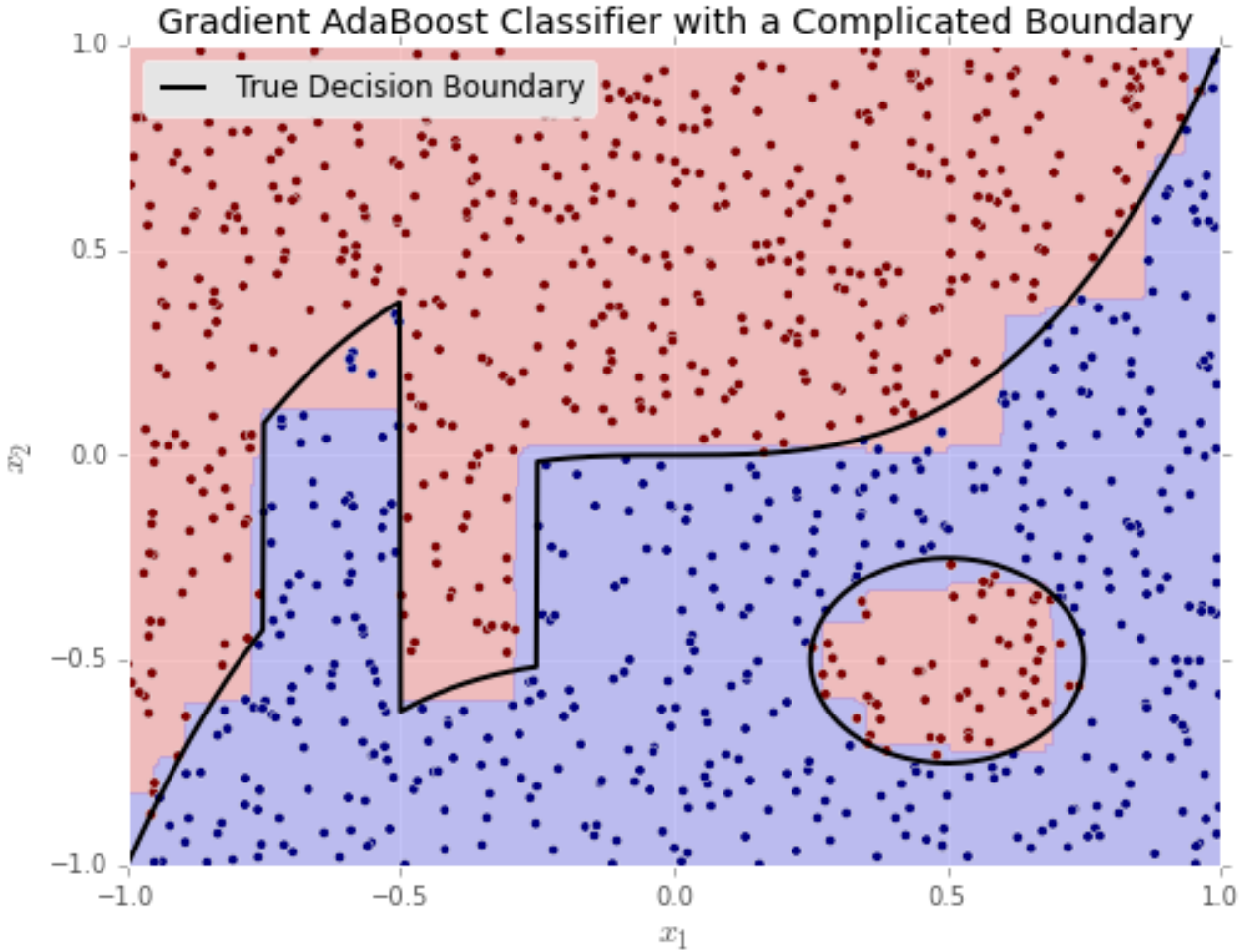
By using the 'exponential' loss, we get the Adaboost algorithm:

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

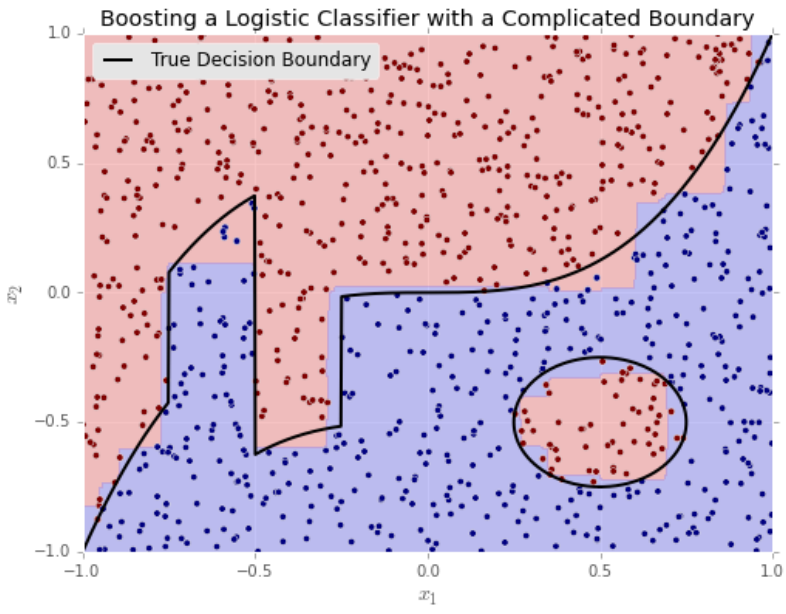
The Adaboost algorithm uses labels $y \in \{-1, 1\}$ and minimizes the loss function

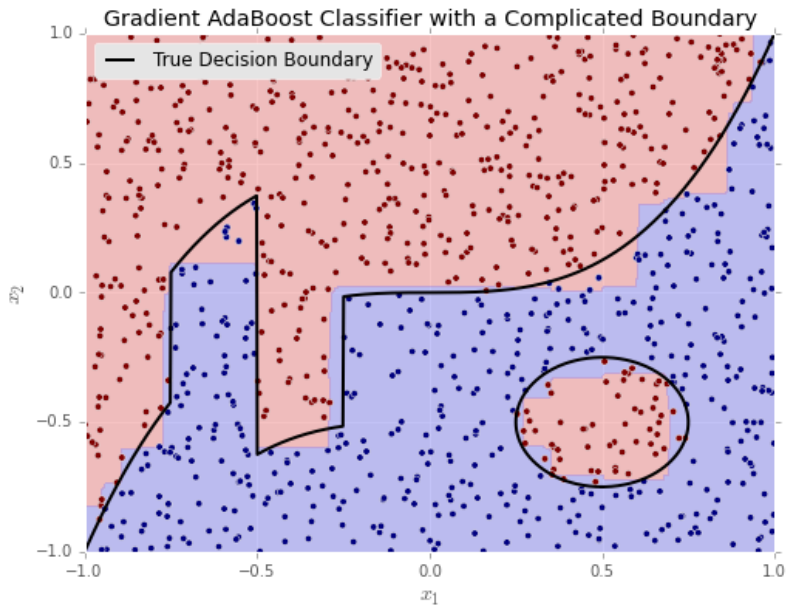
$$L(f, y) = \exp(yf)$$

The performance of Adaboost is generally comparable to that of gradient boosted logistic regression



In fact, the classification boundaries are *the same*





On the other hand, logistic boosting generally returns better calibrated probabilities, and is less sensitive to outliers.

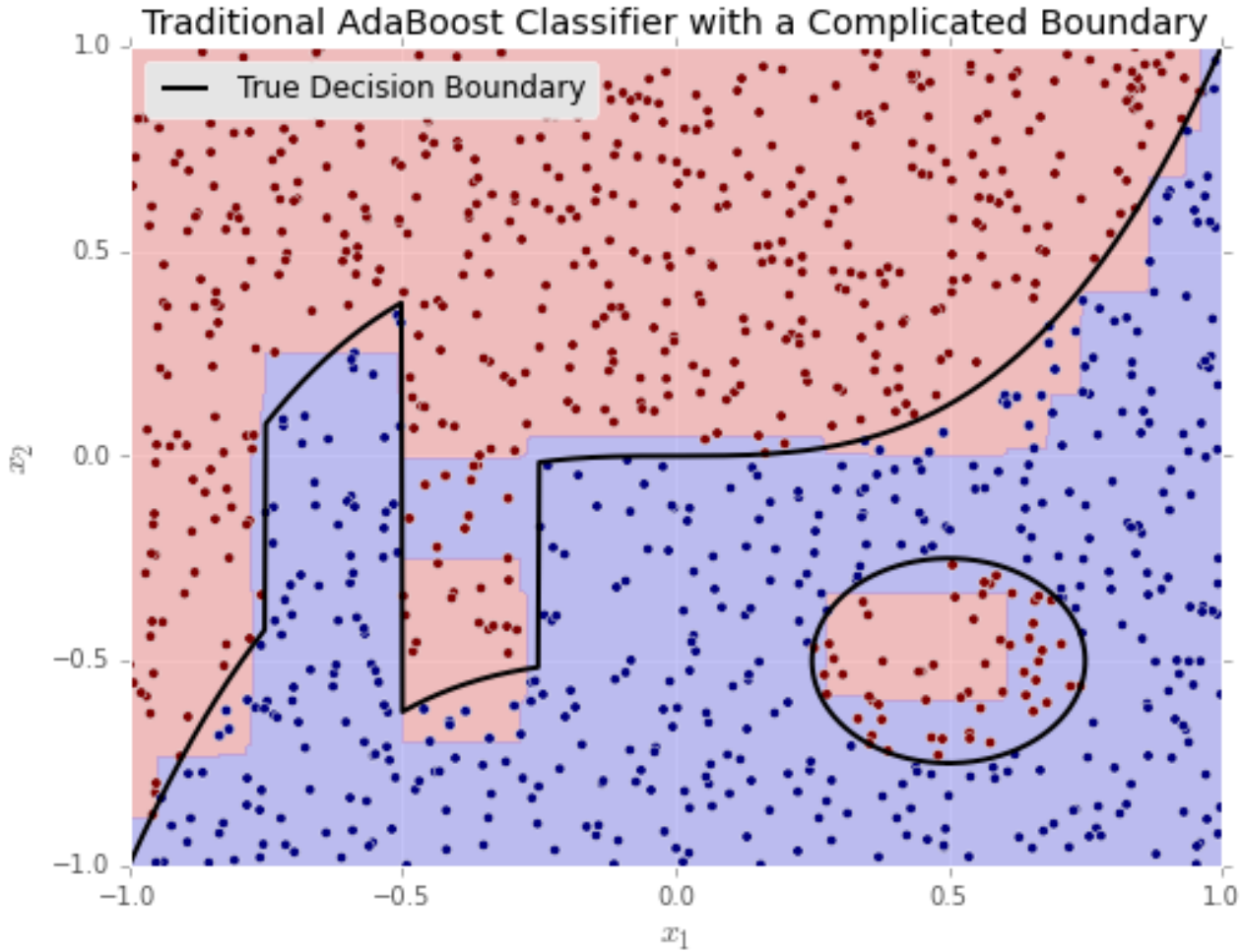
The implementation of Adaboost as a gradient booster is a somewhat recent development.

Originally (before the invention of gradient boosting), Adaboost was accomplished by a complicated sample re-weigting scheme.

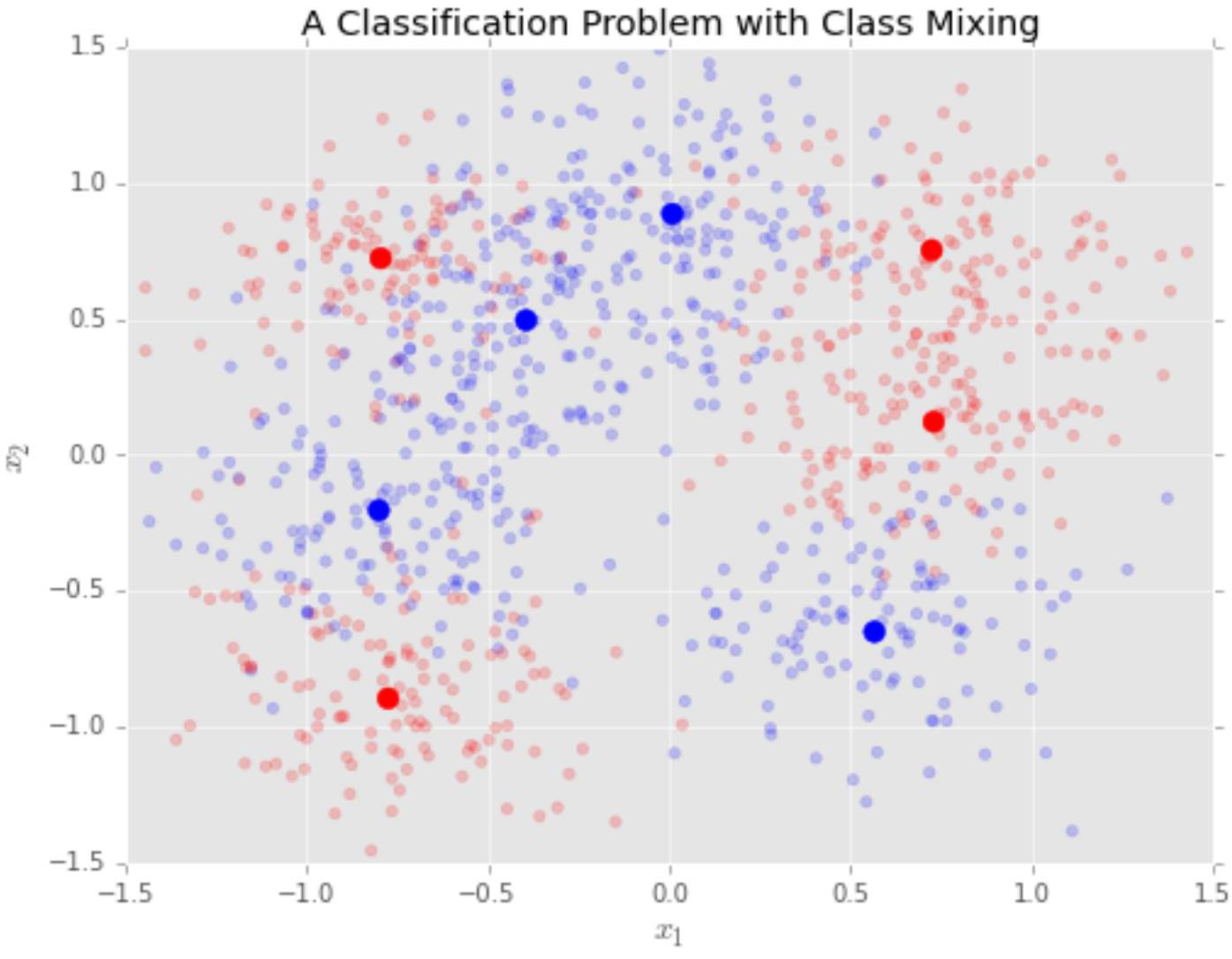
The traditional Adaboost is included in sklearn as `AdaBoostClassifier`

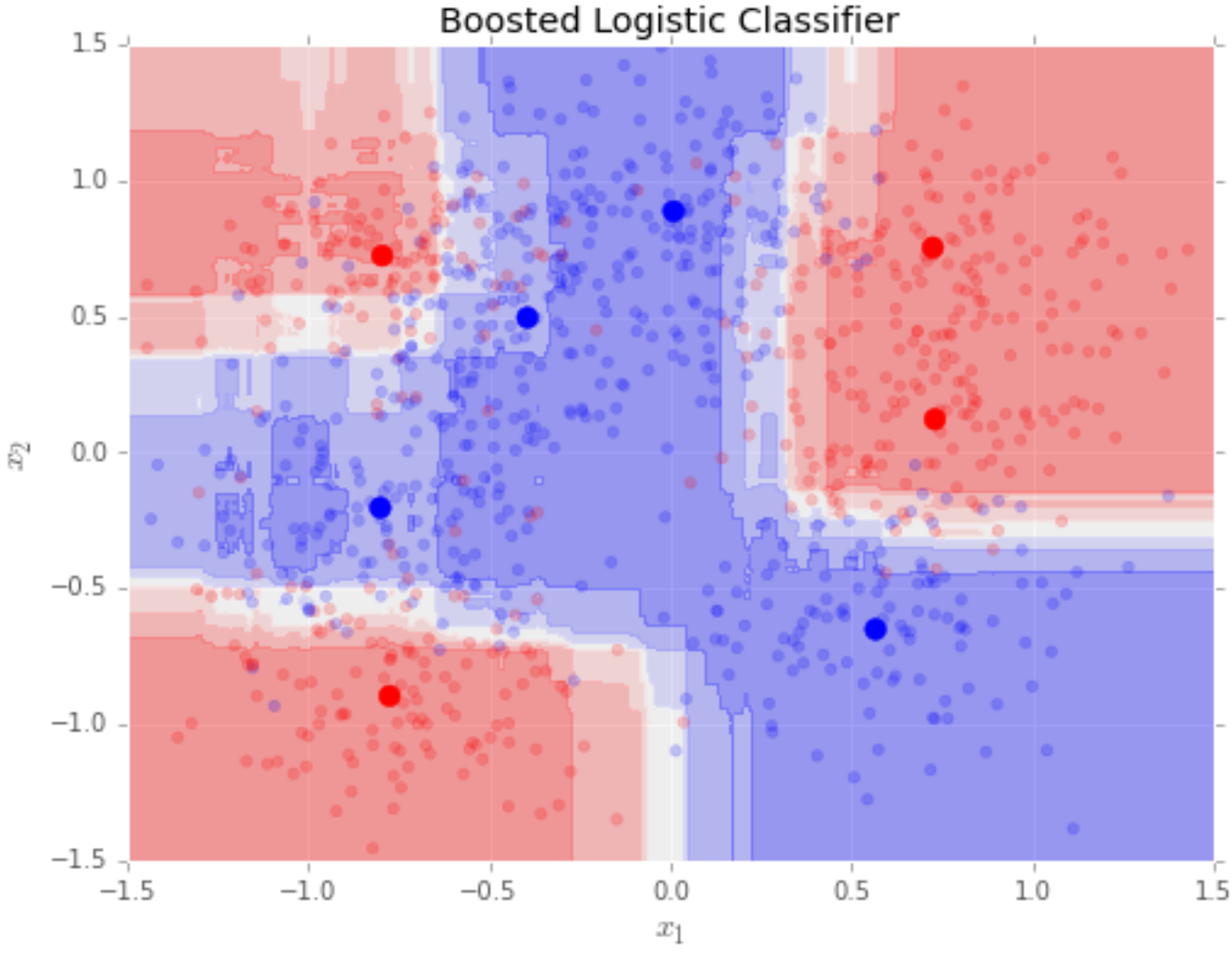
```
model = AdaBoostClassifier(loss='exponential')
model.fit(X, y)
```

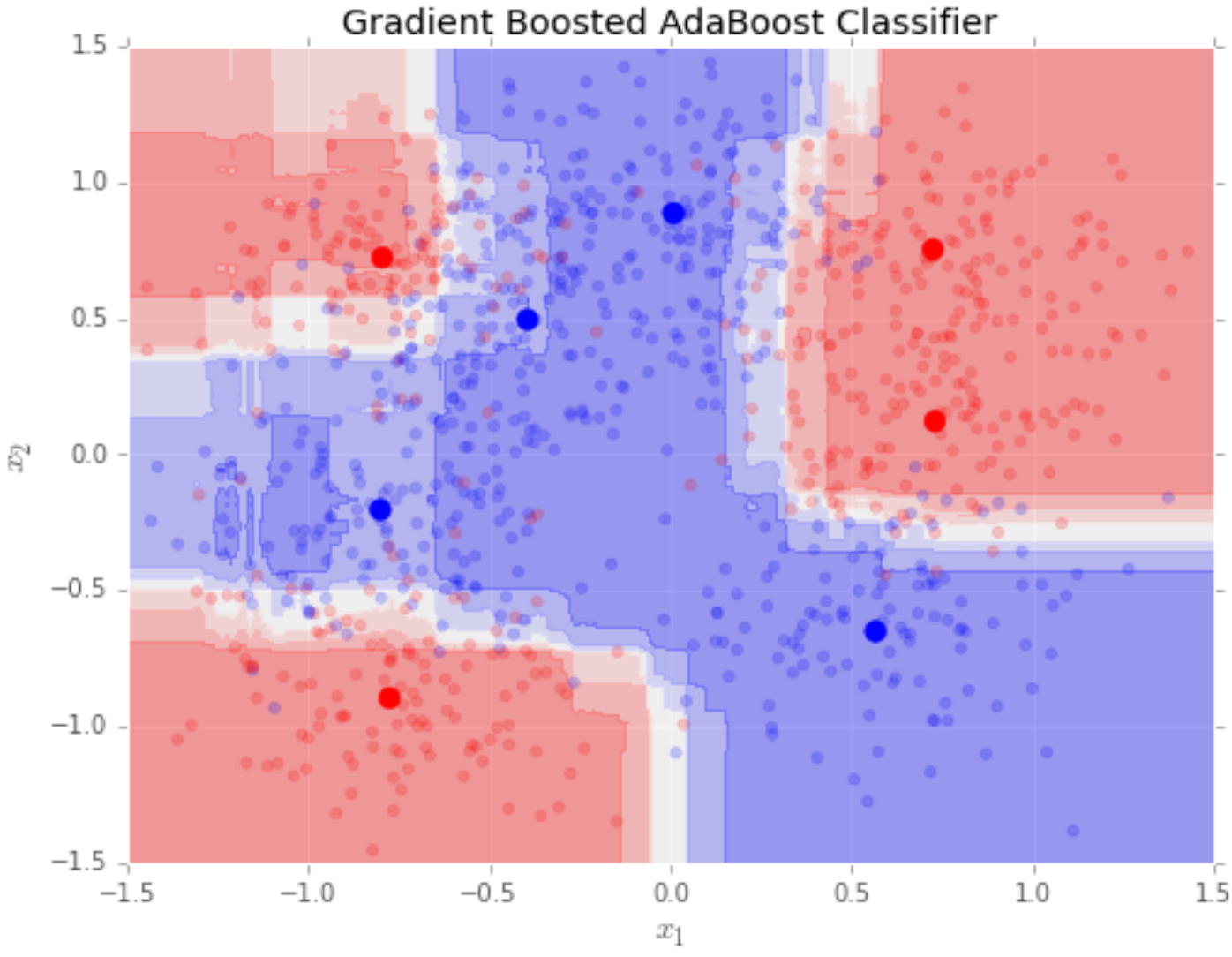
Traditional Adaboost is outclassed by gradient boosting, but retains historical and mathematical significance

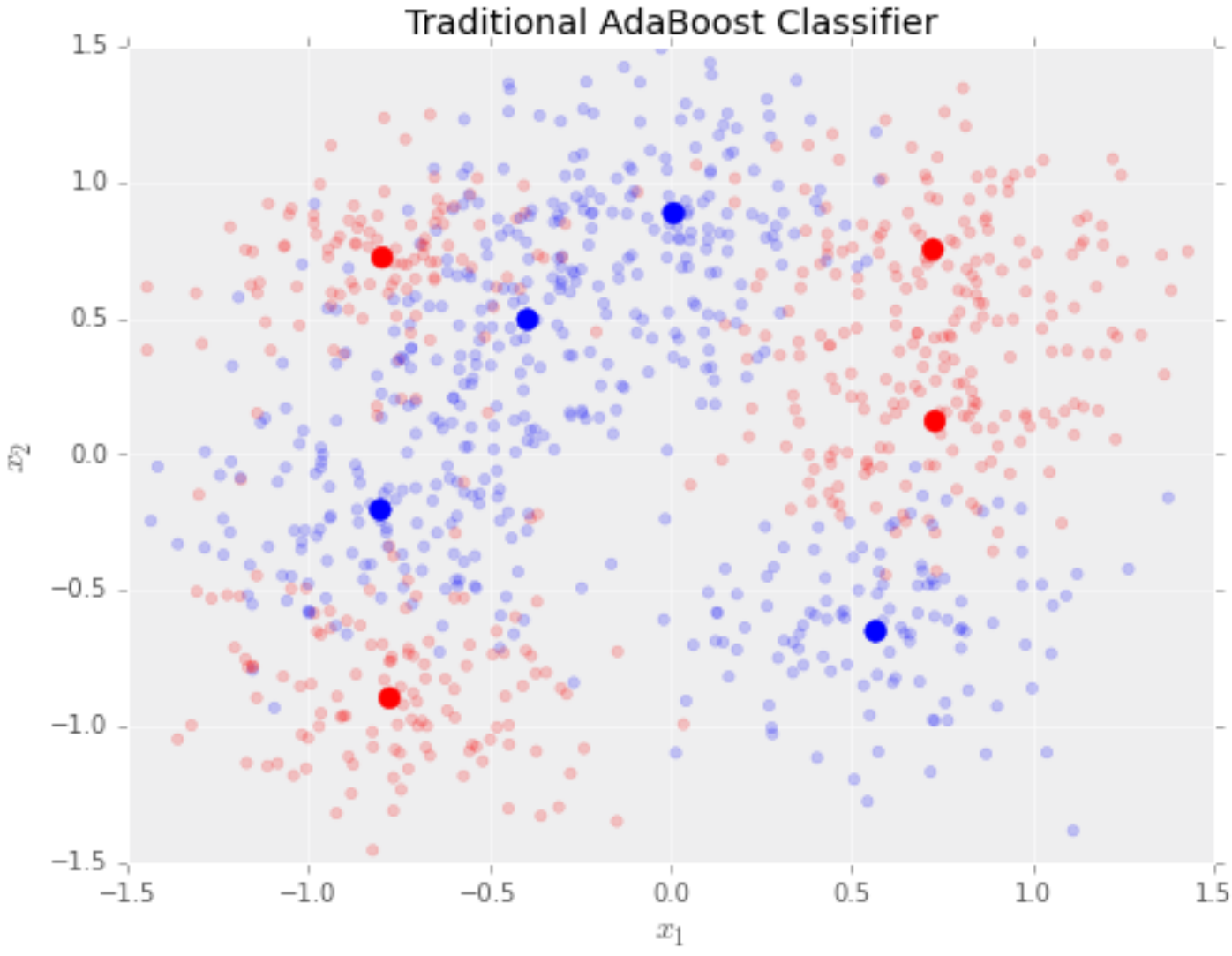


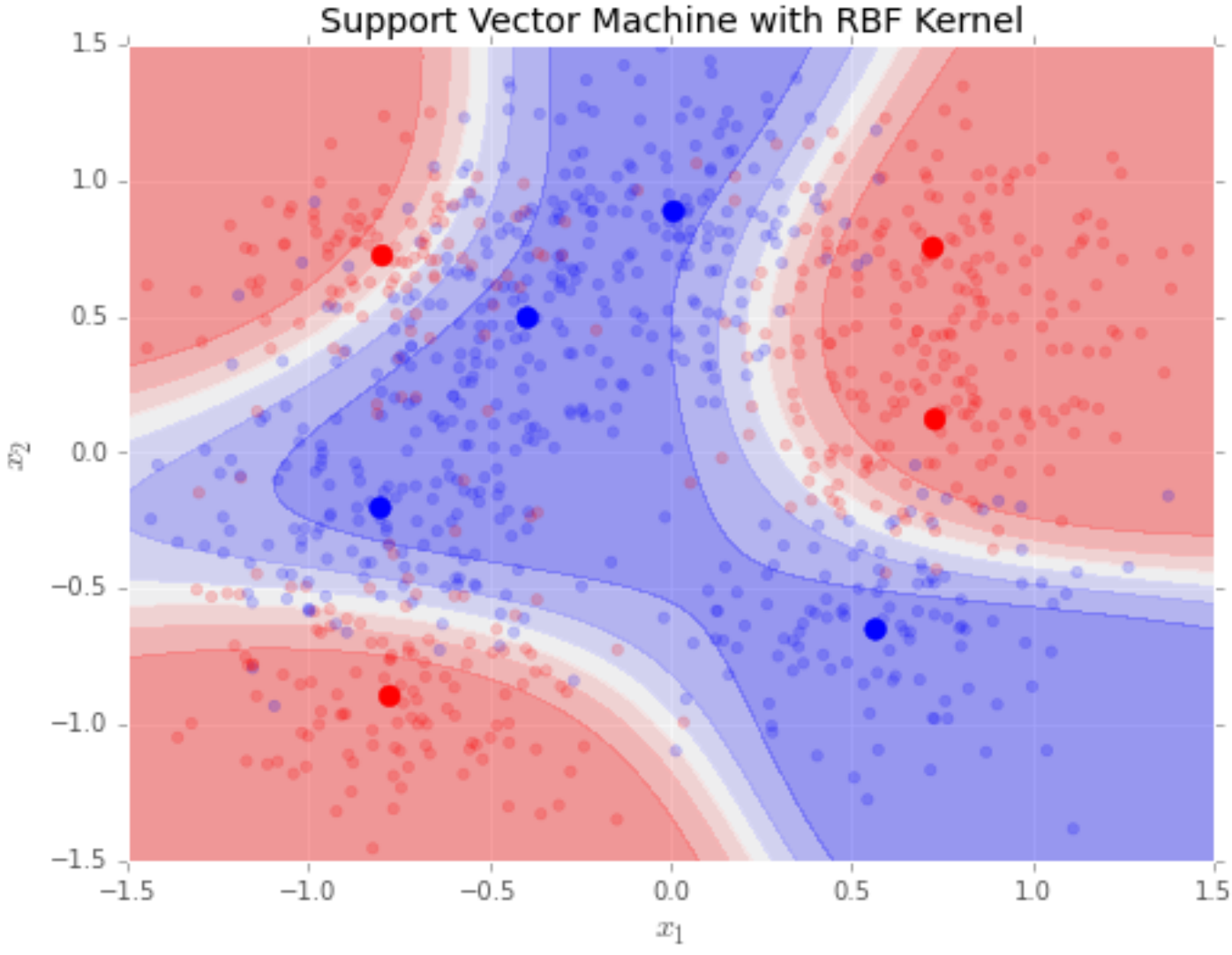
A Final Comparison of Classification Algorithms

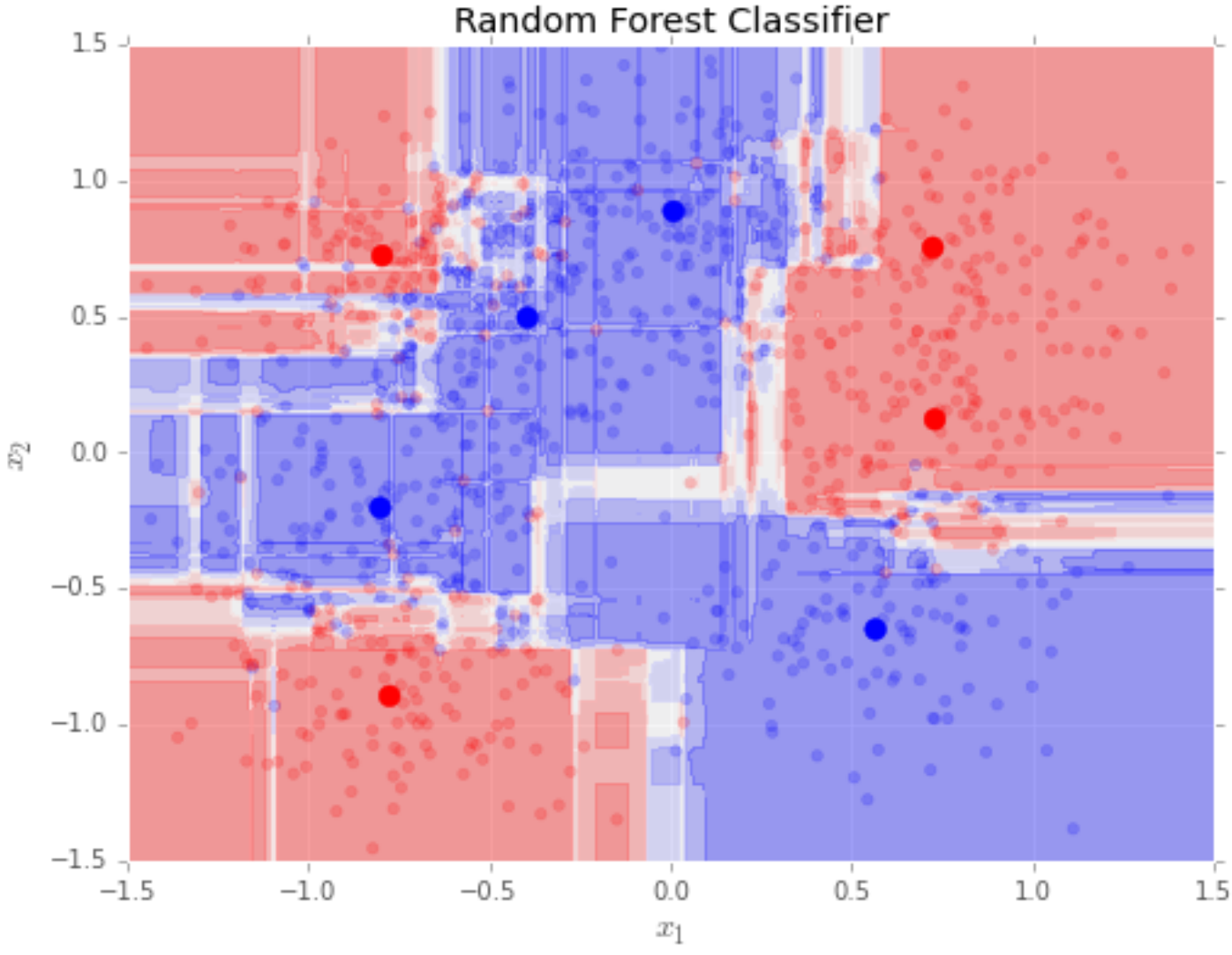



Boosted Logistic Classifier
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
x_2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x_1


Gradient Boosted AdaBoost Classifier
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
x_2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x_1


Traditional AdaBoost Classifier
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
x_2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x_1


Support Vector Machine with RBF Kernel
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
x_2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x_1


Random Forest Classifier
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
x_2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x_1

Final Words About Boosting

Gradient Boosting is the best off-the-shelf learning algorithm available today.

It effortlessly produces accurate models.

Nonetheless, it has drawbacks.

Drawbacks of Gradient Boosting

- Boosting creates very complex models. It can be difficult to extract intuitive, conceptual, or inferential information from them.

Drawbacks of Gradient Boosting

- Boosting is difficult to explain (maybe you just learned this through experience). It can be hard to convince business leader to accept such a black box model.

Drawbacks of Gradient Boosting

- Boosted models can be difficult to implement in production environments due to their complexity.

Drawbacks of Gradient Boosting

- The sequential nature of the standard boosting algorithm makes it very difficult to parallelize (compared to, for example, random forest). Recently, there has been great progress (xgboost).