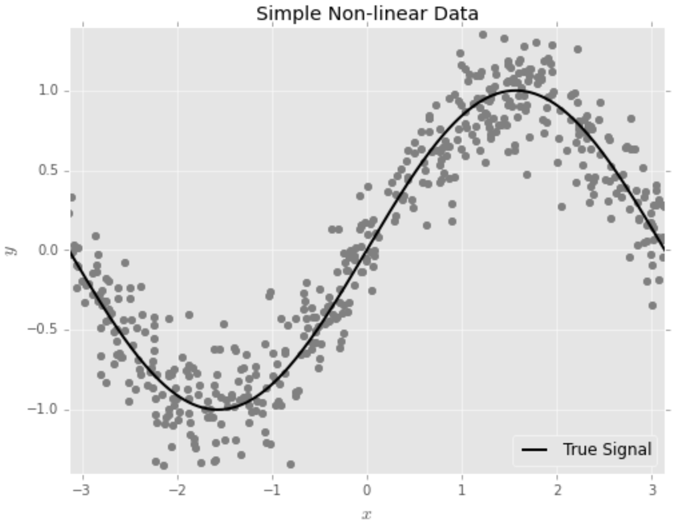
Boosting

Matthew Drury

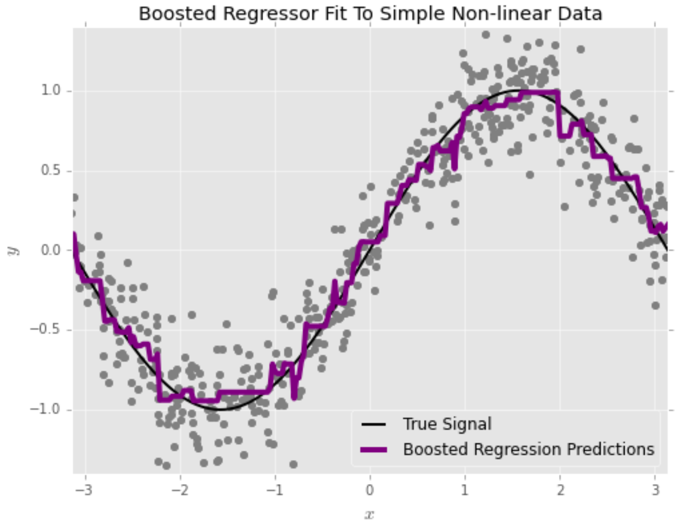
August 17, 2016

Boosting encompasses an entire family highly successful learning algorithms.

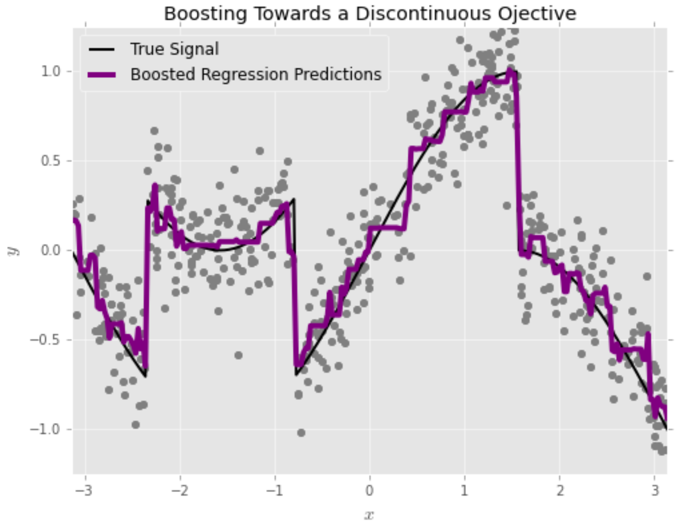
Boosting can adapt itself effortlessly to very non-linear objectives



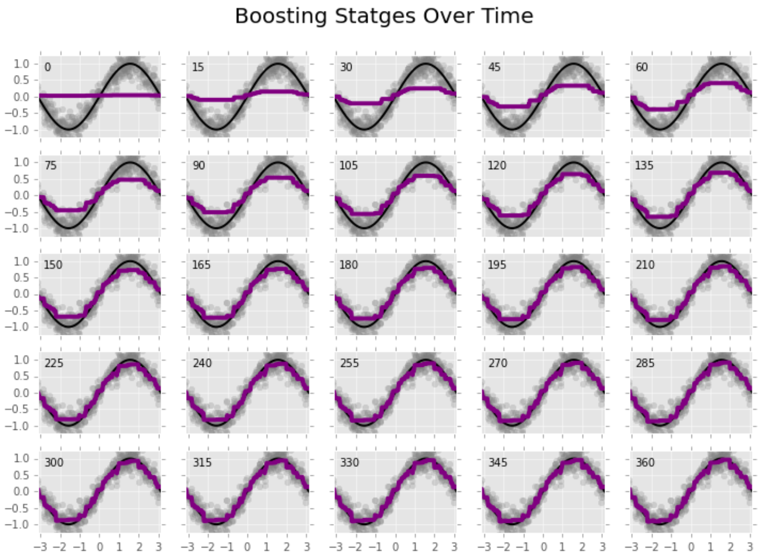
Boosting can adapt itself effortlessly to very non-linear objectives



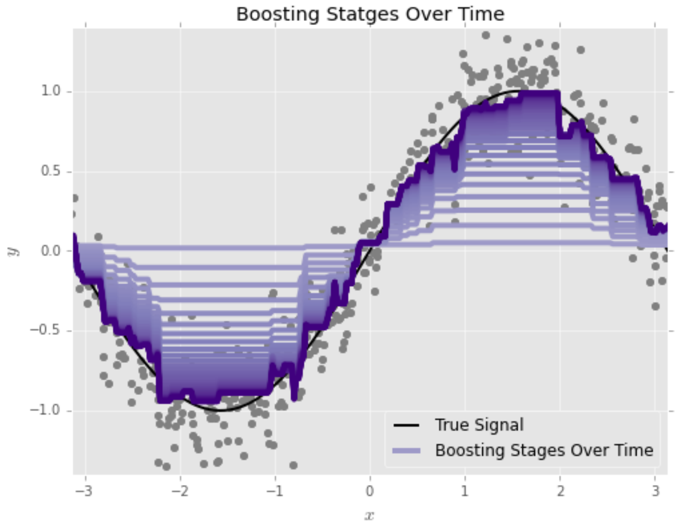
Boosting can adapt itself effortlessly to very non-linear objectives



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



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.

Compared to:

Linear Models:

Lowers variance by using a regularization penalty.

Lowers bias by including more predictors.

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.

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.

Outline of the Lesson

Agenda:

- Introduction
- Boosting To The Residuals
- Practical Gradient Boosted Regression
- Interpreting Gradient Boosted Regression
- Boosting for Classification.
- Drawbacks of Boosting

Objectives:

- Understand the conceptual foundation of Boosting
- Understand the algorithm's hyperparameters, and how to tune them.
- Understand some basic strategies for interpreting a booster.
- Understand when boosting is appropriate, and when it is not.
- Be aware of the possibility of creating your own loss function.

Boosting To Residuals

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}.

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

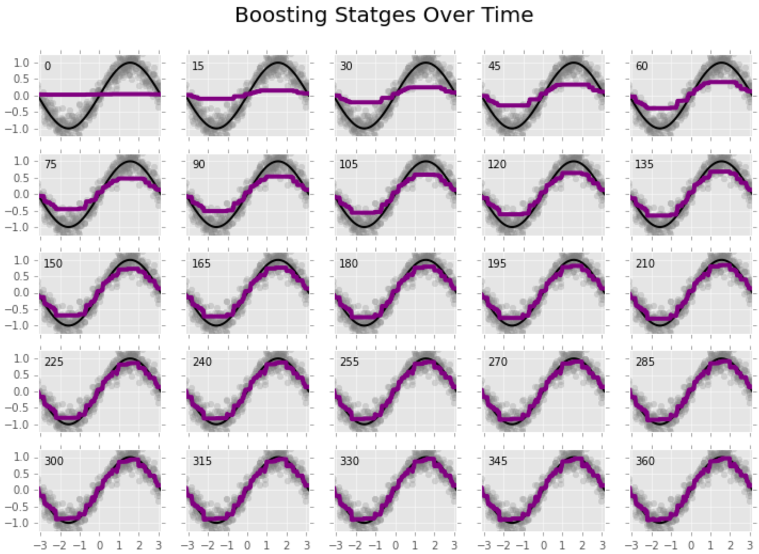
$$y_i \approx f(x_i) \text{ for all } i$$

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

$$\sum_i (y_i - f(x))^2$$

is small.

As we saw in the introduction, we would like to build up f in stages, making small adjustments as we go along.



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

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

The process of building up the model looks like

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

Where Should We Start?

We want to make the simplest possible choice for our first stage

$$f_0(x) = \text{constant}$$

But what constant?

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

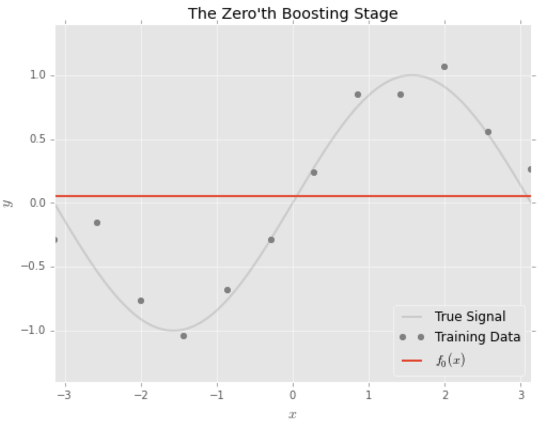
$$\sum_i (y_i - \text{constant})^2$$

Question: What is the correct choice here?

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

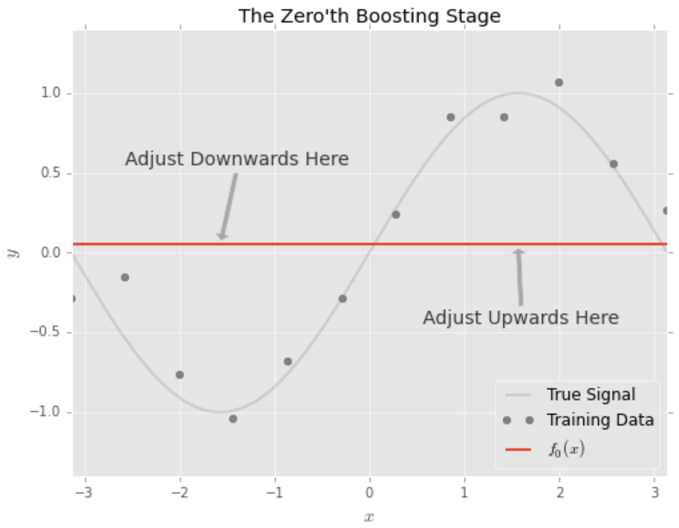
How To Update

We are now in the situation illustrated below

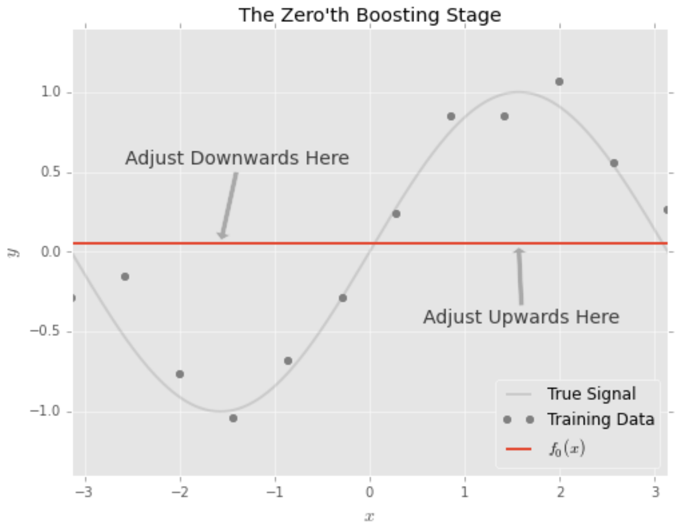


Our next task is to update our simple f_0 to $S_1(x) = f_0(x) + f_1(x)$.

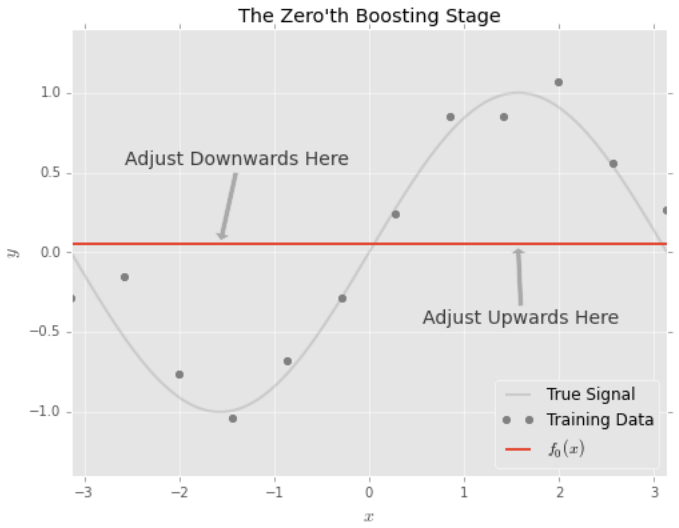
Intuitively, it's obvious what we should do



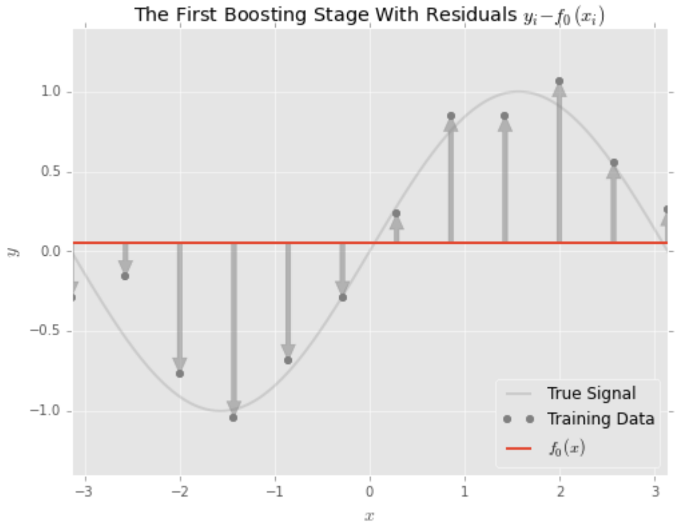
But how can we use *the data* to tell us this? We won't always have the luxury of such simple pictures.



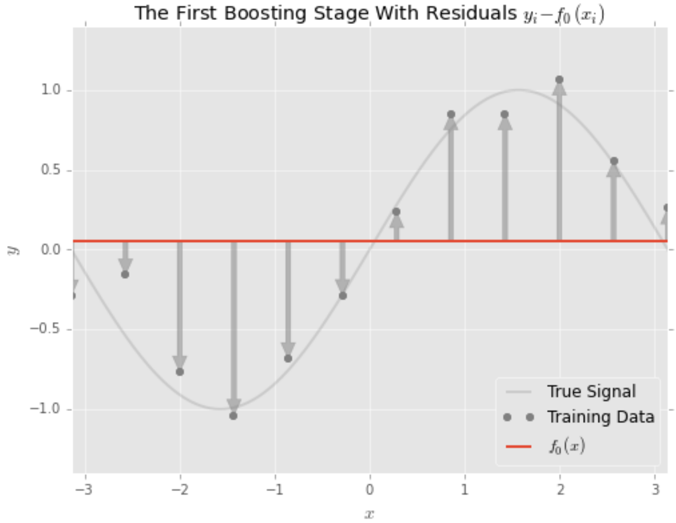
Take a moment to think: how is the data is telling us what to do?



The *residuals*, $y_i - f_0(x_i)$ answer this question

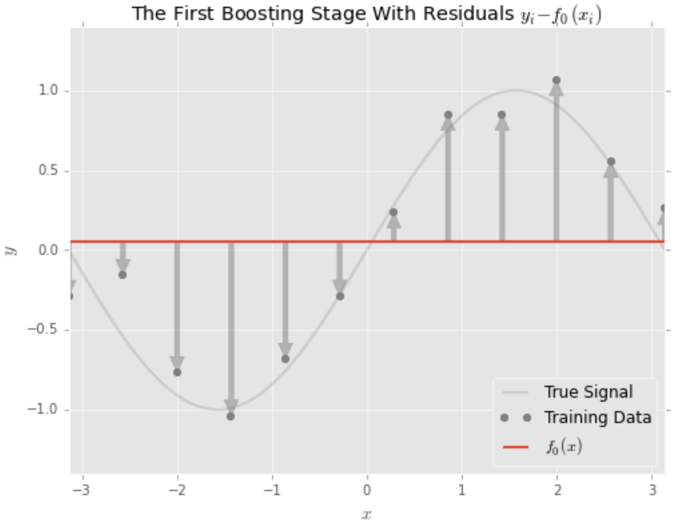


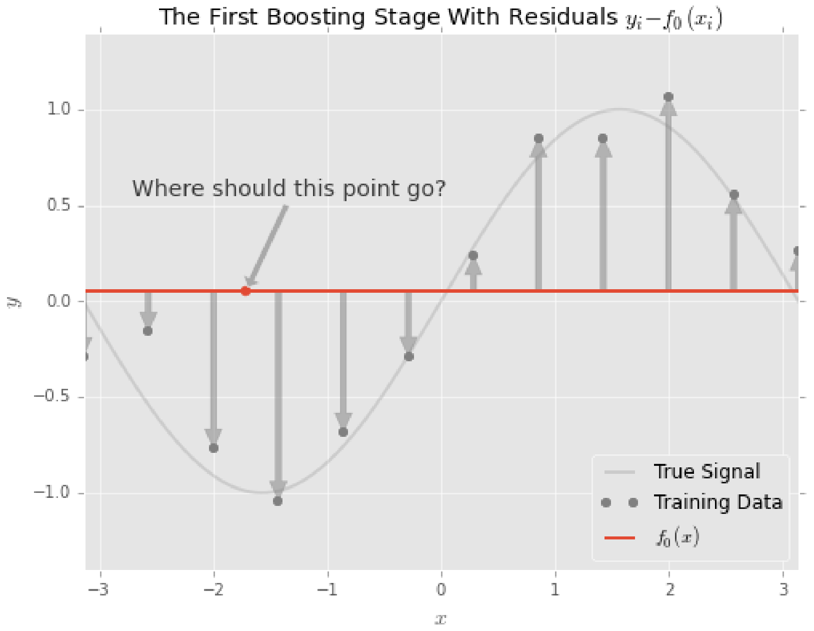
We should adjust $f_0(x)$ in the direction of the residuals!



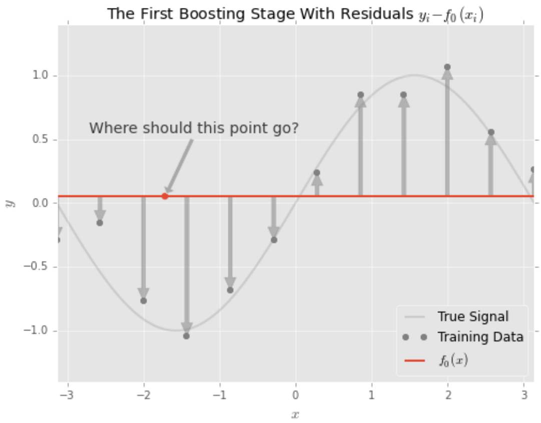
We should adjust $f_0(x)$ in the direction of the residuals!

But... there is a huge issue with this...




The First Boosting Stage With Residuals $y_i - f_0(x_i)$
Where should this point go?
True Signal
Training Data
$f_0(x)$
y
x
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3

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*.

Solution:

Solution:

Fit a model to the residuals!

Solution:

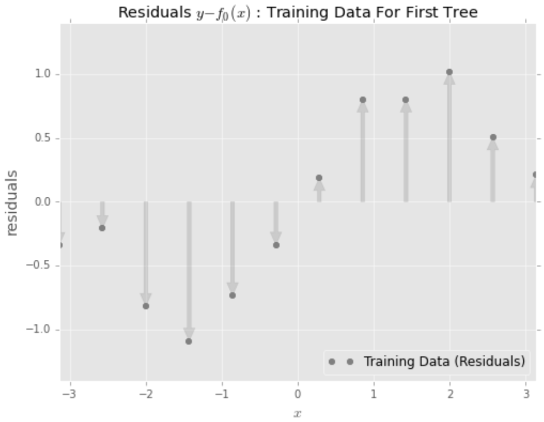
Fit a model to the residuals!

The **predictions from this model** both:

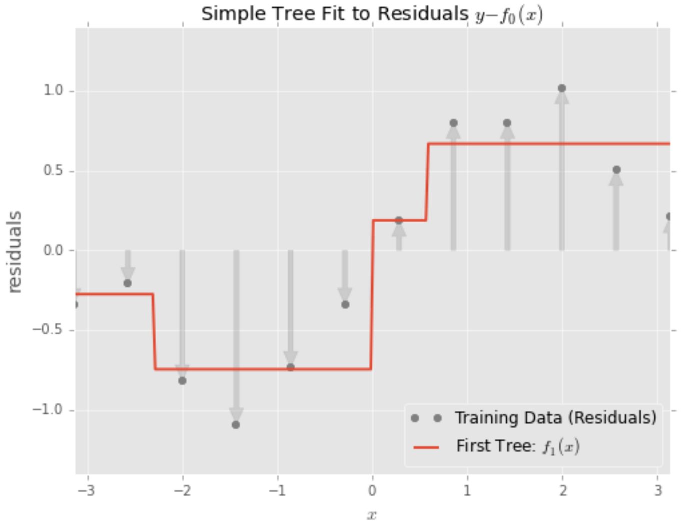
- Approximate the residuals at the places we *have* data.
- Are defined everywhere.

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*: $y_{\text{new}} = y_i - f_0(x_i)$.

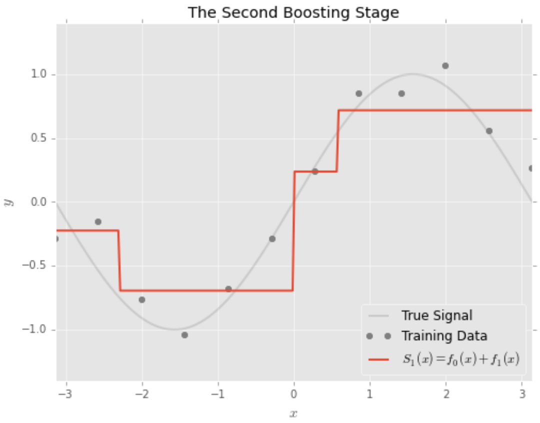


Here is a very simple model fit to this *working data set*



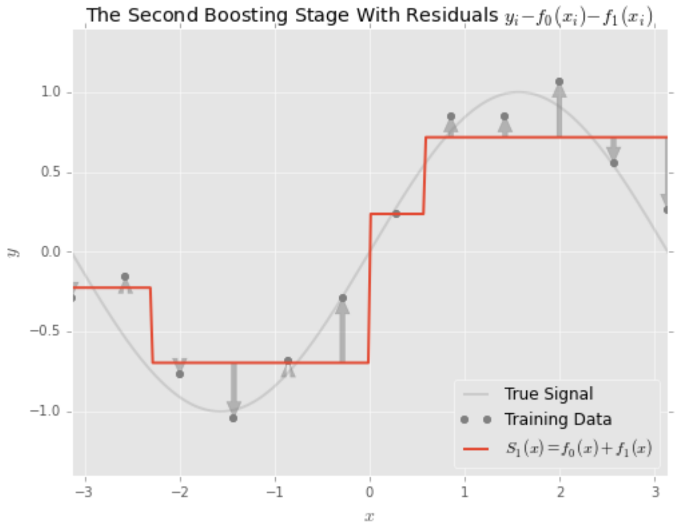
Now we can update the model.

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

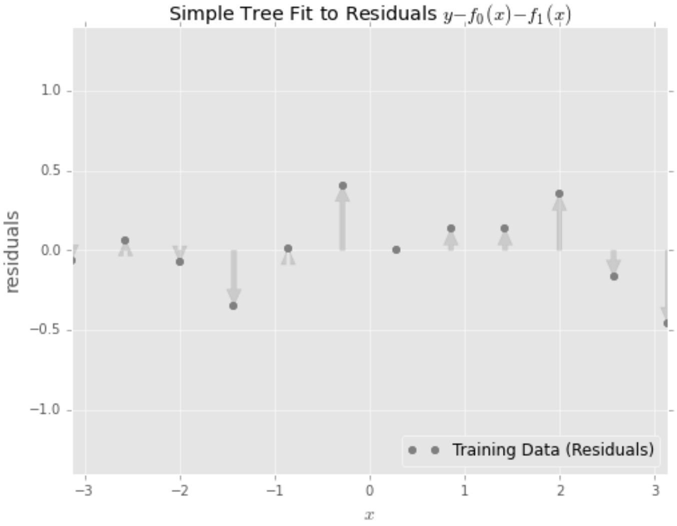


Let’s go one more step, so that the idea is clear.

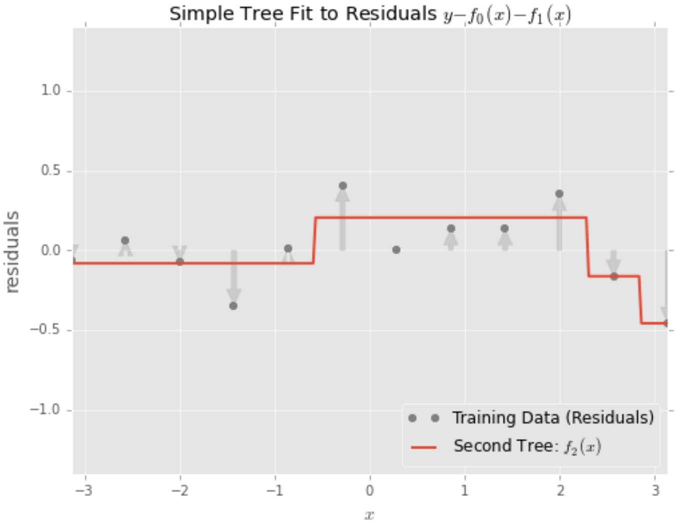
Calculate the residuals of the current model...



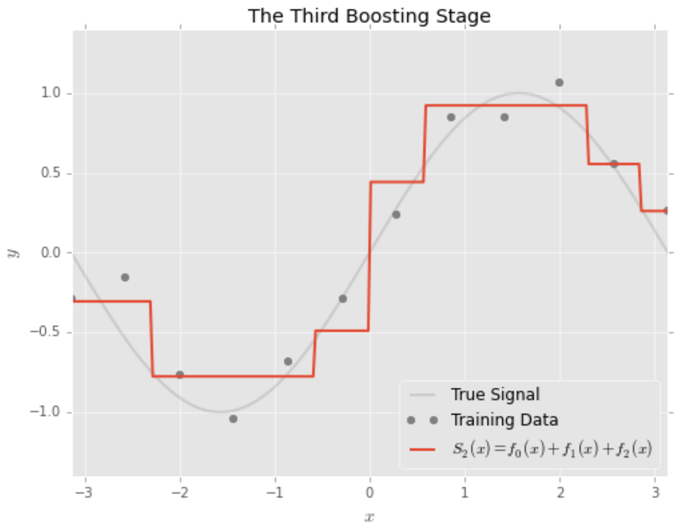
Create a training data set with the residuals as the response...

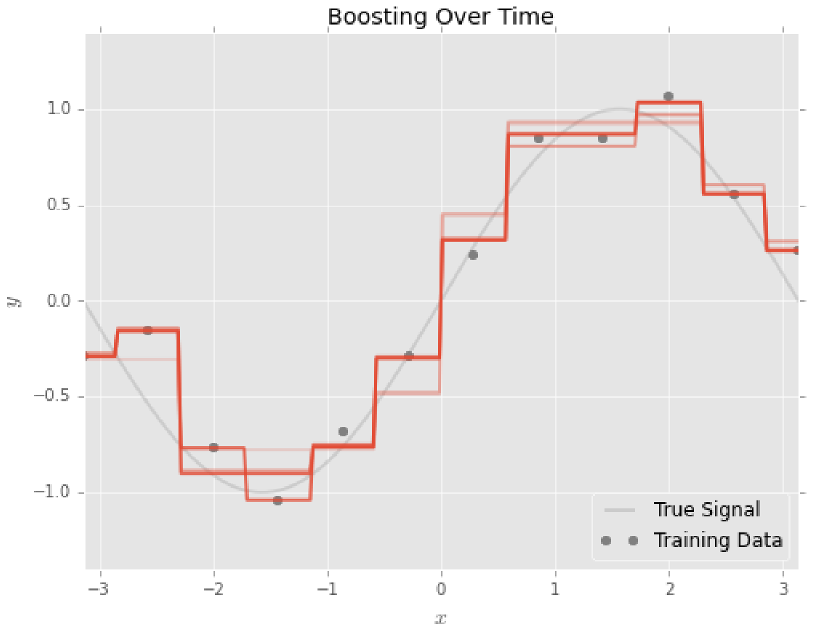


Create a training data set with the residuals as the response...



Update the model!




Boosting Over Time
1.0
0.5
0.0
−0.5
−1.0
y
−3
−2
−1
0
1
2
3
x
True Signal
Training Data

But, What Happened to Growing Slowly Over Time?

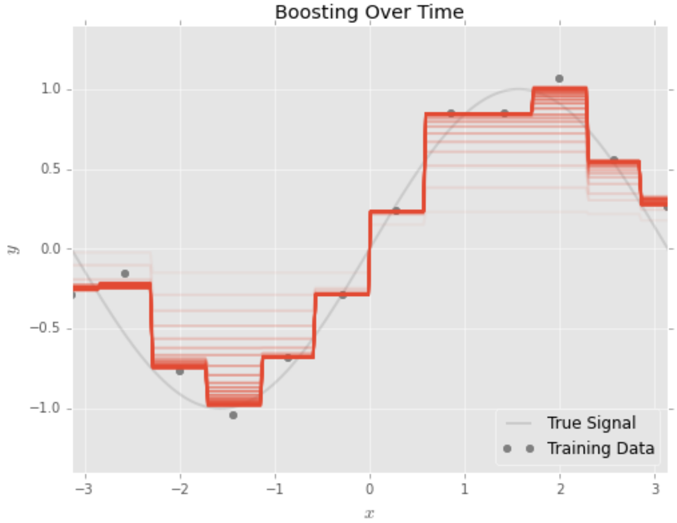
Instead of adding in the entirety of the residual fitted tree during an update

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

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

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

$\lambda = 0.01$



Boosting Over Time
1.0
0.5
0.0
−0.5
−1.0
y
−3
−2
−1
0
1
2
3
x
True Signal
Training Data

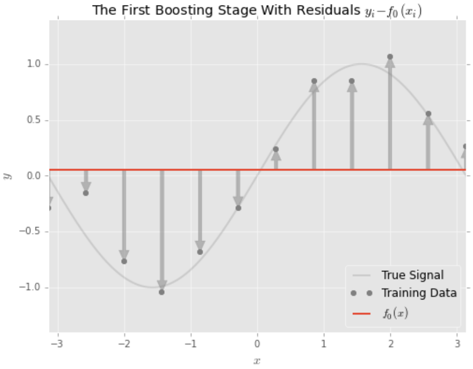
Gradient Boosting to Minimize Sum of Squared Errors

Inputs: A data set $\{x_i, y_i\}$, and a learning rate λ.
Returns: A function f such that $f(x_i) \approx y_i$.

- Initialize $S_0(x) = f_0(x) = \frac{1}{N}\sum_i y_i$.
- Iterate (parameter k) until satisfied:
    - Create the working data set $W_k = \{x_i, y_i - S_{k-1}(x_i)\}$.
    - Fit a decision tree to W_k, minimizing least squares. Call this tree f_k.
    - Set $S_{k+1}(x) = S_k(x) + \lambda f_k(x)$.
- Return $f_{\max}(x) = f_0(x) + f_1(x) + f_2(X) + \cdots + f_{\max}(x)$.

Why is it Called Gradient Boosting?

Remember when we were in this situation:



We want to use the residuals to update the model

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

Removing the details of the specific situation, this looks like

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

$$\Downarrow$$

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

Question: Does this look... familiar?

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 L.
Outputs: A point x_{optim} that minimizes L.

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

What happens when we apply gradient descent to minimize the squared error loss?

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

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

So gradient descent for least squares looks like...

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

That is, **least squares gradient descent iteratively moves in the direction of the residual**.

This is why the algorithm is called **Gradient Boosted Regression**.

We "boost" the current estimate by adding an *approximation to the gradient of the squared error loss function*.

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

```python
for preds in model.staged_predict(X_new):
    # Do something interesting, used heavily in the plots to
      follow.
```

`GradientBoostingRegressor` has many knobs to turn.

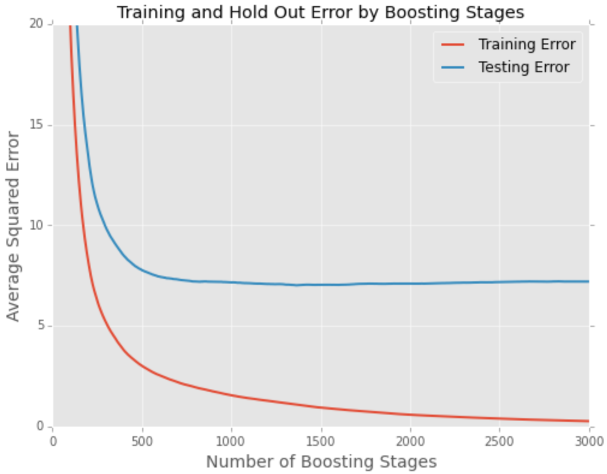
```
GradientBoostingRegressor(loss='ls',
                          n_estimators=100,
                          learning_rate=0.1,
                          max_depth=3,
                          subsample=1.0,
                          min_samples_split=2,
                          min_samples_leaf=1,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

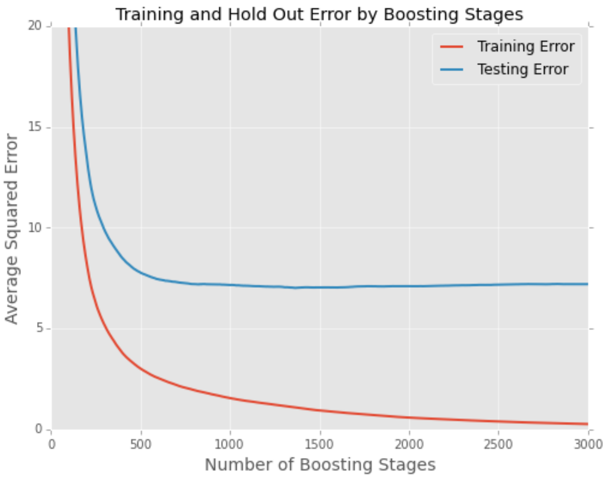
The most important options to `GradientBoostedRegressor` are

- `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

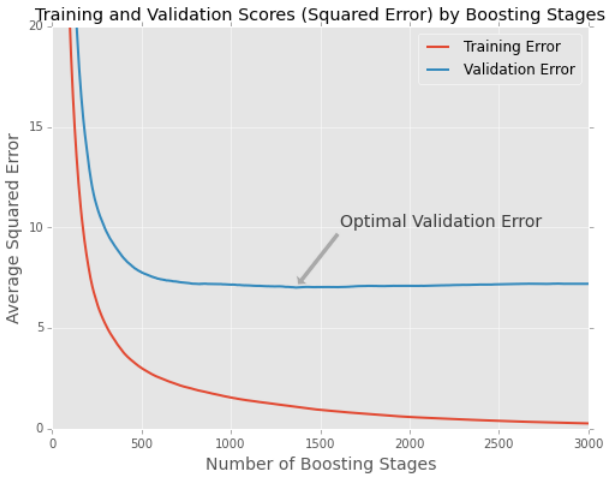
As more and more trees are added to the model the training set error will be driven down monotonically, but the same is not true for the testing error





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

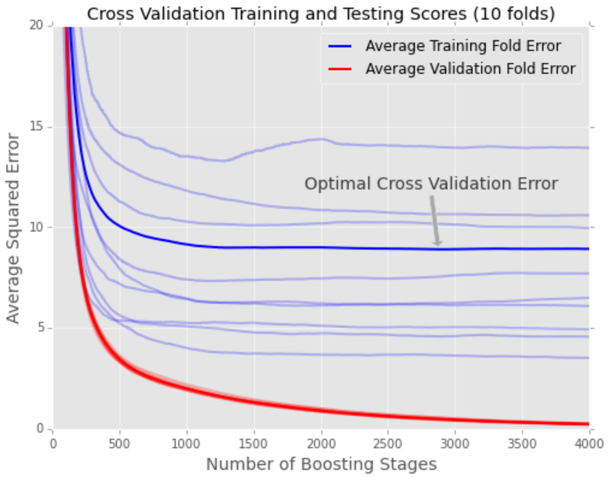
```python
def get_optimal_n_estimators(model, X_new, y_new):
    validation_loss = np.zeros(
        model.get_params('n_estimators'))
    for i, preds in enumerate(model.staged_predict(X_new)):
        validation_loss[i] = model.loss_(preds, y_new)
    optimal_tree = np.argmin(validation_loss)
    optimal_loss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

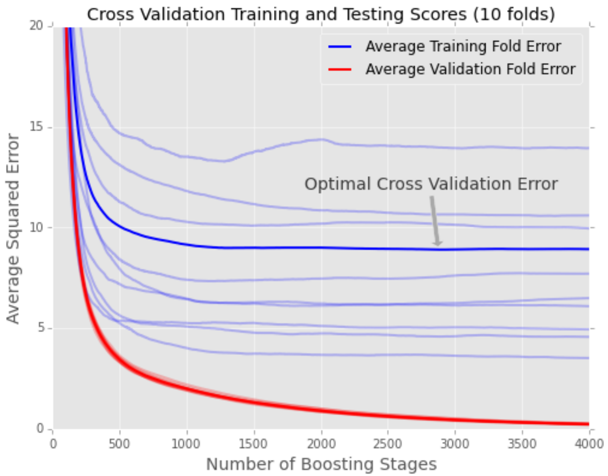
One the optimal number of estimators is known, it's useful to truncate the model

```
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,
      :]
    new_model.n_estimators = n_estimators
    return new_model
```

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.

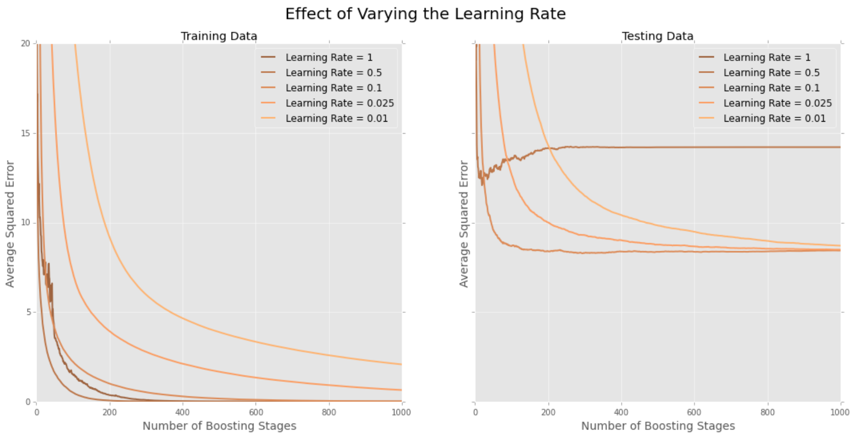
Tuning the Learning Rate

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

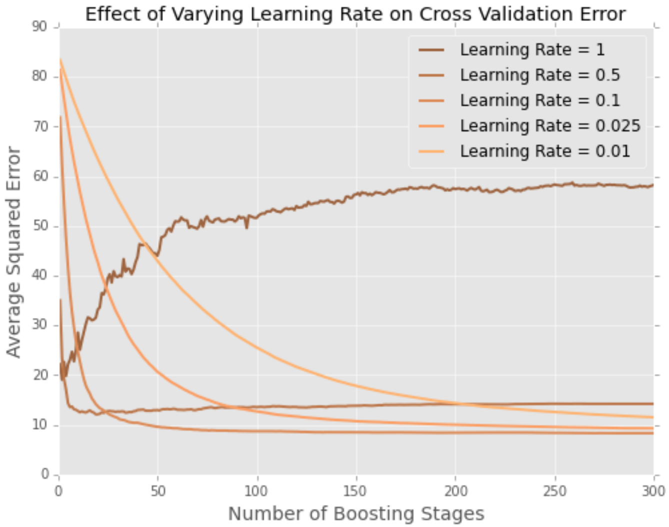
A large learning rate will cause the model to fit hard to the training data, which creates a *high variance* situation.

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

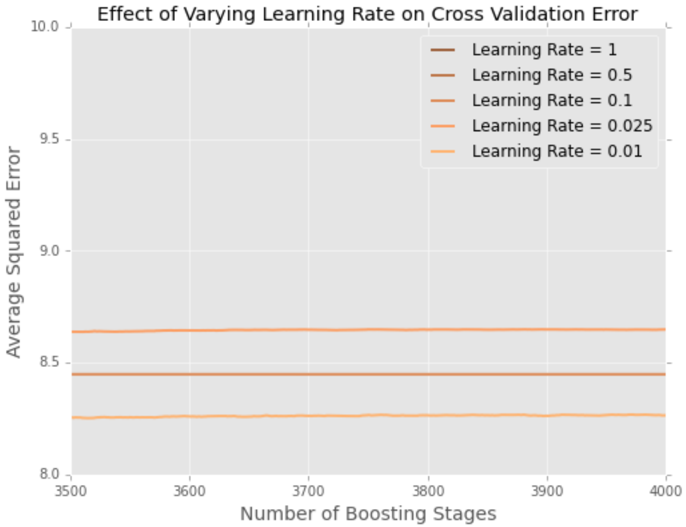
Decreasing the learning rate reduces how fast the booster drives down the training error rate



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.

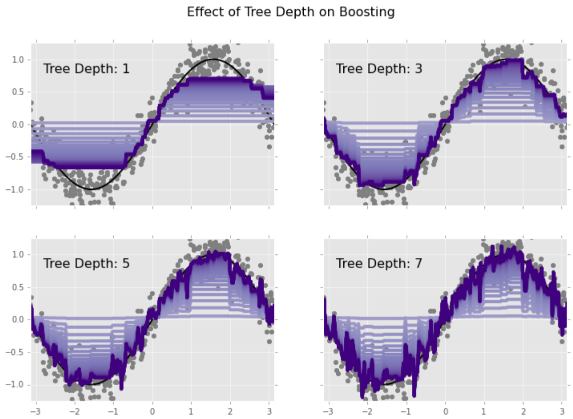
General Advice:

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

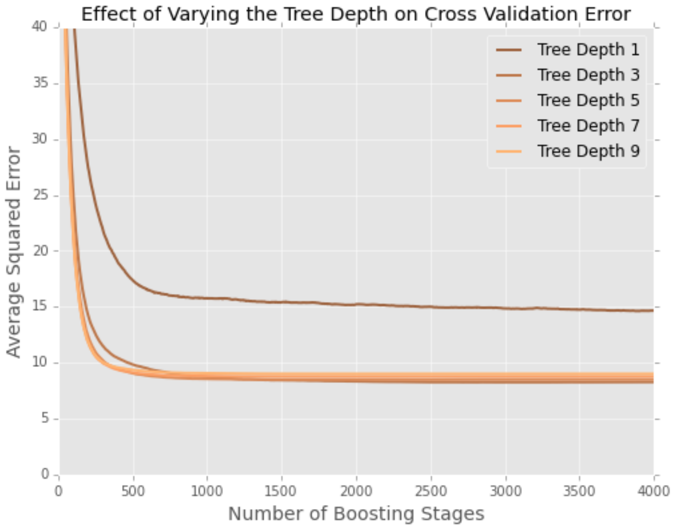
A larger tree depth allows the model to capture deeper interactions between the predictors, resulting in lower bias.

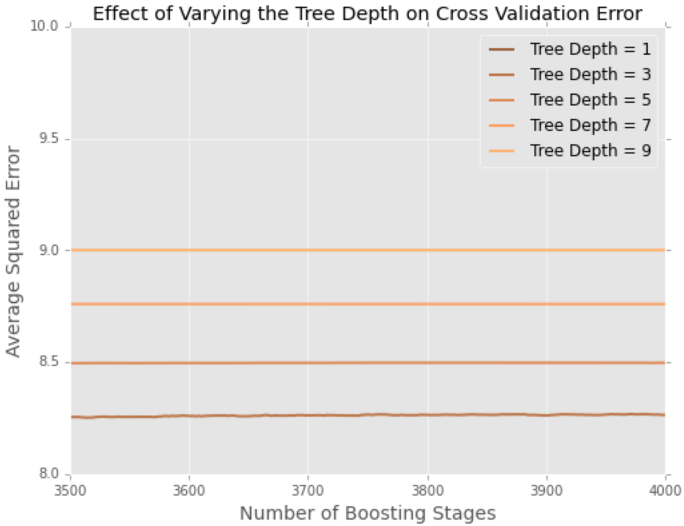


A deeper tree depth also

- 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

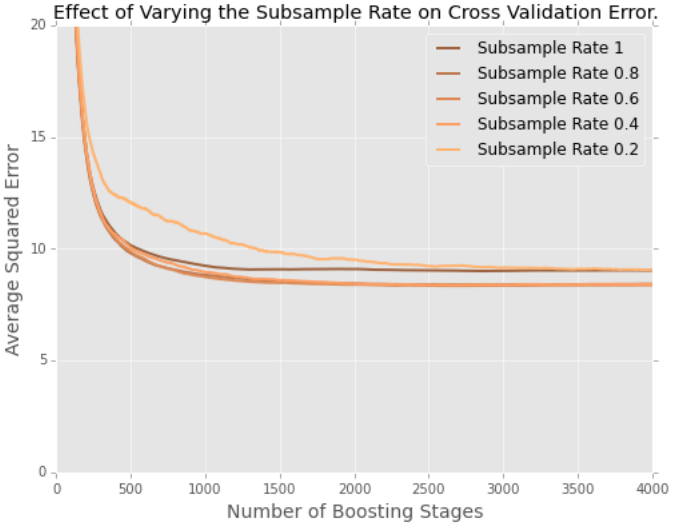
Strategy for Tree Depth:
Tune with a grid search and cross validation.

Tuning the Subsample Rate

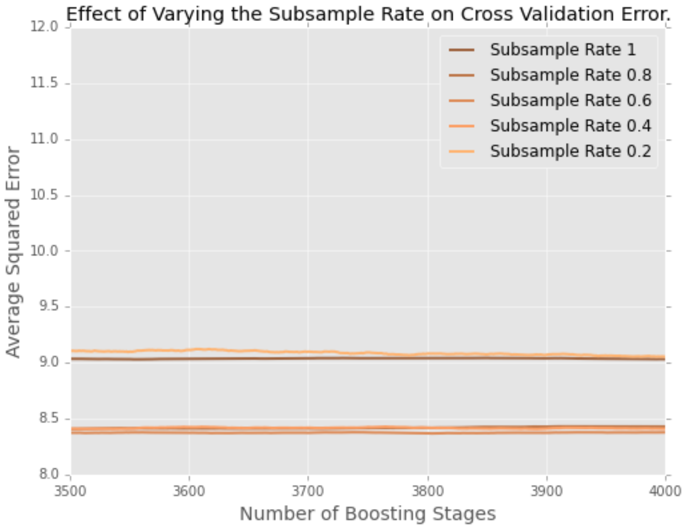
The `subsample` parameter allows one to train each tree on a subsample of the training data.

This is similar to bagging in the random forest algorithm, 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.

If you have a massive amount of data and want the model to fit more quickly, decrease this value.

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*!

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*.

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.

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

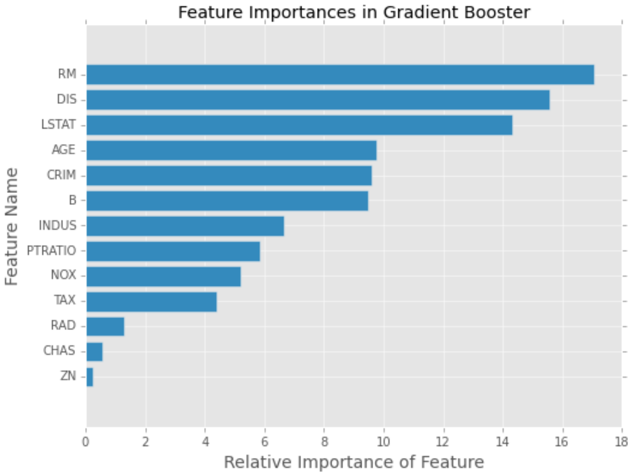
The concept here is the same as in random forest.

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:

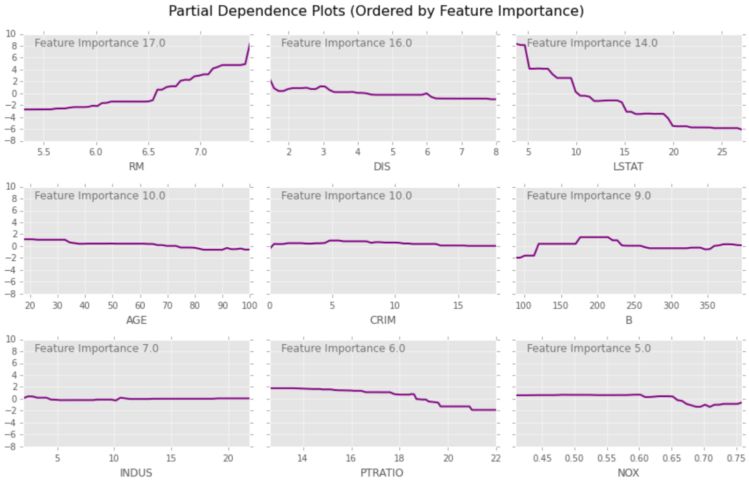
- Make sure your model only includes trees *up to the optimal point*. Otherwise you'll allocate importance to overfitting.

Comments:

- Dominant features should be treated with suspicion. They can often be a sign of data leakage.

Partial Dependence Plots

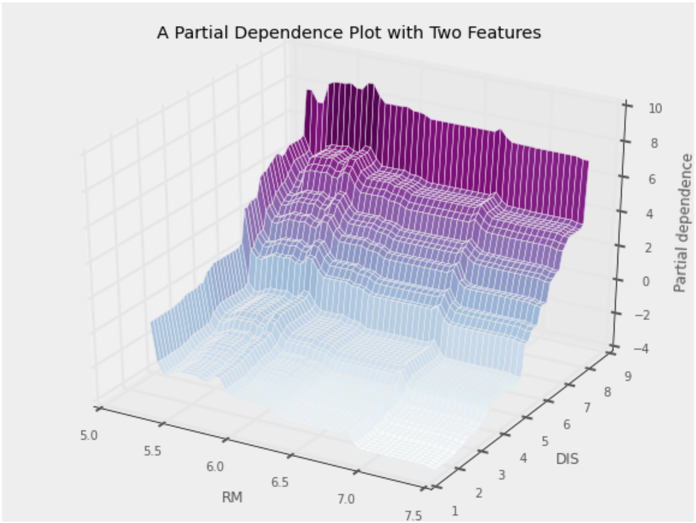
Visualizations of the effect of a single predictor, averaging out the effects of all the rest



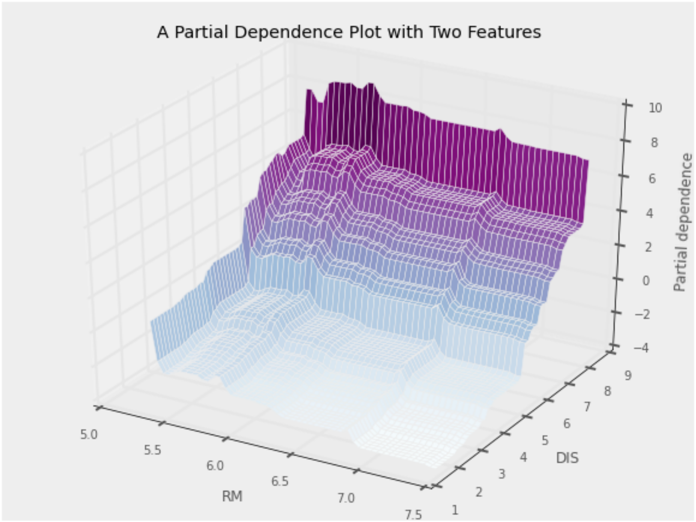
In symbols:

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

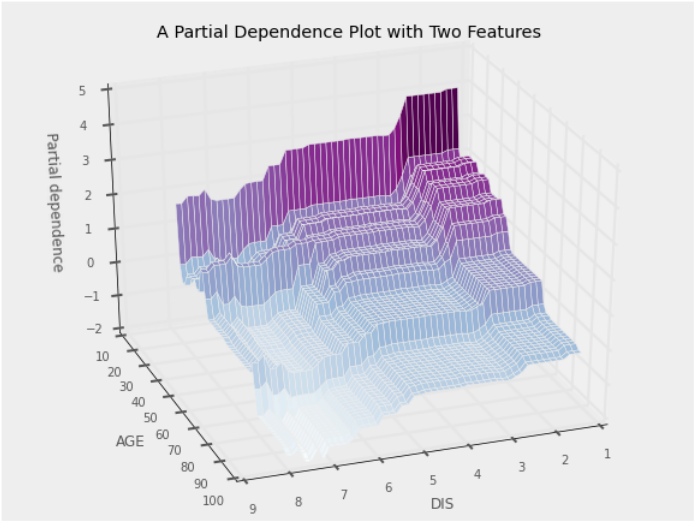
By varying the values of two predictors, we can draw partial dependence plots in higher dimensions



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*!

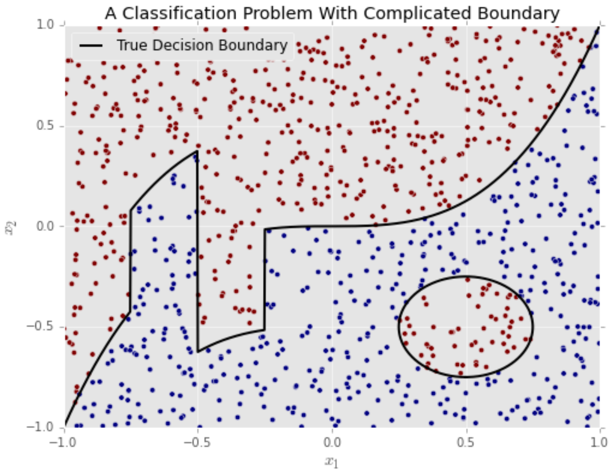
Gradient Boosted Logistic Regression

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

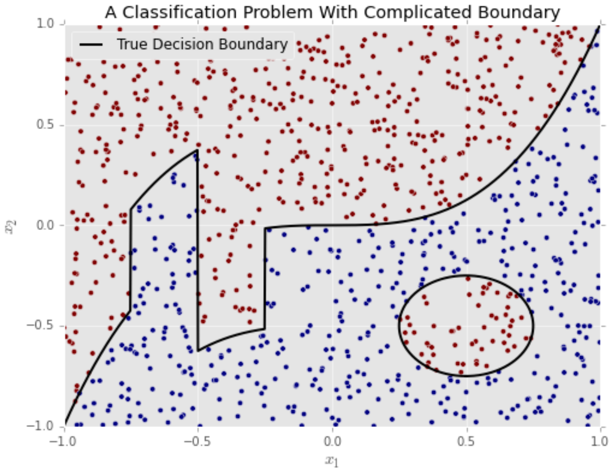
$$y \in \{0, 1\}$$

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

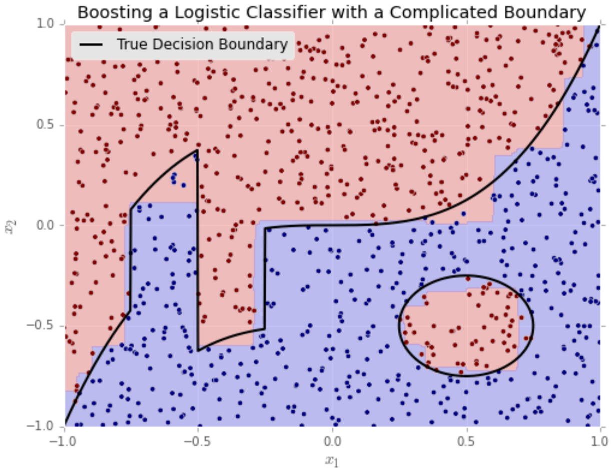
For example, here is a complex classification problem



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 $\nu(\beta, x_i) = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_M x_{iM}$.

The quantity ν is called the *linear predictor*.

In logistic regression it represents the *log odds* of the outcome.

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

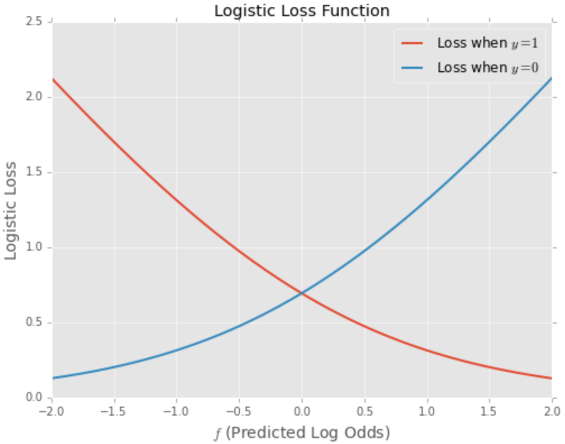
$$p(x) = \frac{1}{1 + e^{-(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_M x_M)}}$$

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

$$p(x) = Pr(y = 1 \mid x)$$

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

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



Logistic regression can be solved with gradient descent.

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

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

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.

The gradient of the logistic loss is

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

So we can interpret this method as "boosting to the residual probabilities".

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.

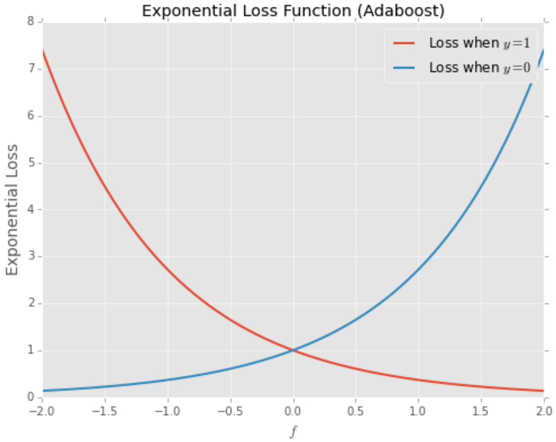
Gradient Boosted AdaBoost

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

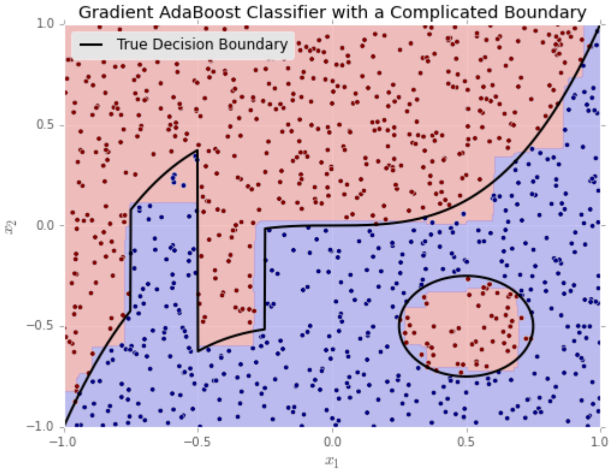
```
model = GradientBoostingClassifier(loss='exponential')
model.fit(X, y)
```

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

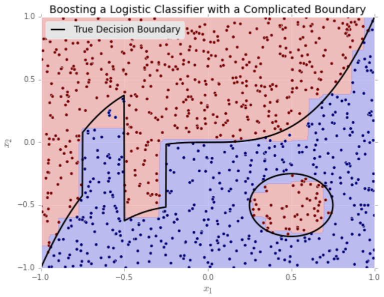
$$L(f, y) = \exp(-yf)$$

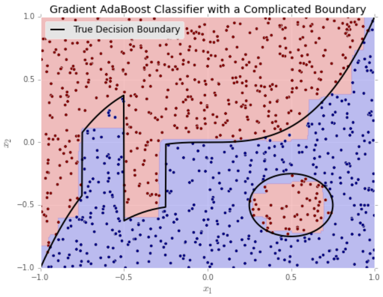


The performance of Adaboost is generally comparable to that of gradient boosted logistic regression

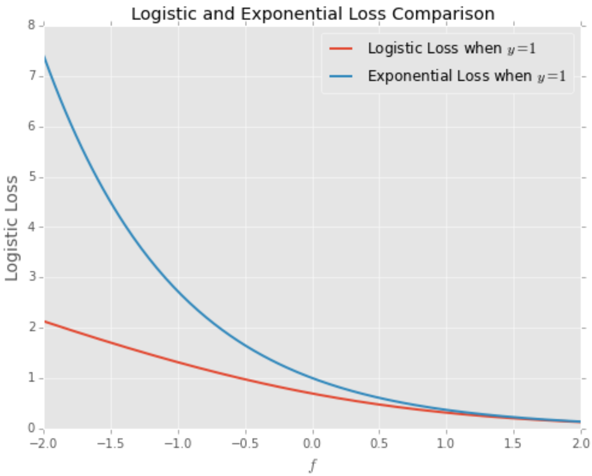


In fact, the classification boundaries are *the same*





Gradient boosted logistic regression is less sensitive to outliers than Adaboost. Can you see why?



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

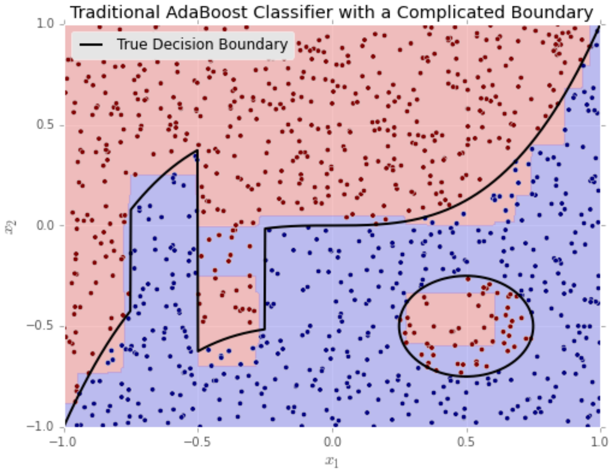
Originally (before the invention of gradient boosting), Adaboost was accomplished by a complicated sample re-weigting scheme.

The traditional Adaboost is included in sklearn as `AdaBoostClassifier`

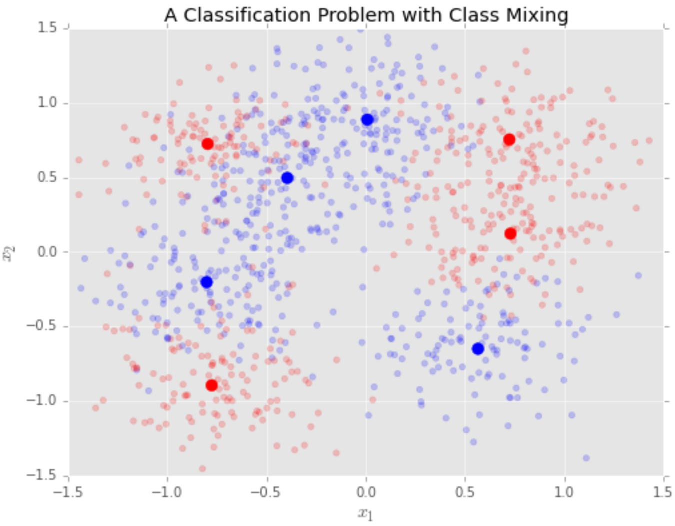
```
model = AdaBoostClassifier()
model.fit(X, y)
```

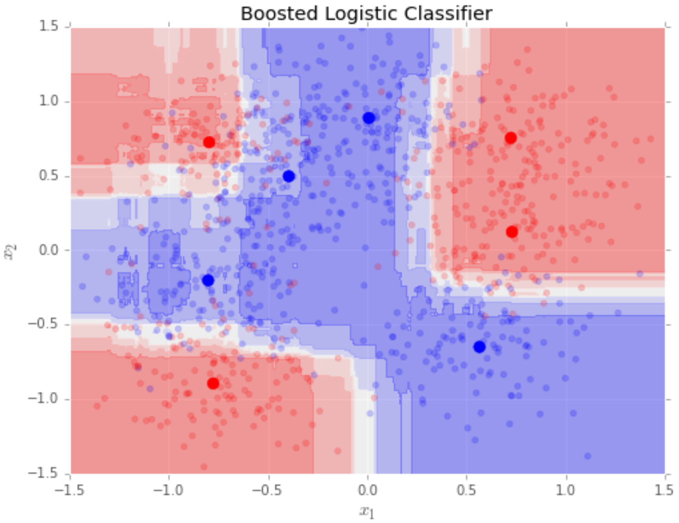
Traditional Adaboost always uses trees of depth 1 (often playfully called "stumps", so there is no `max_depth` argument.

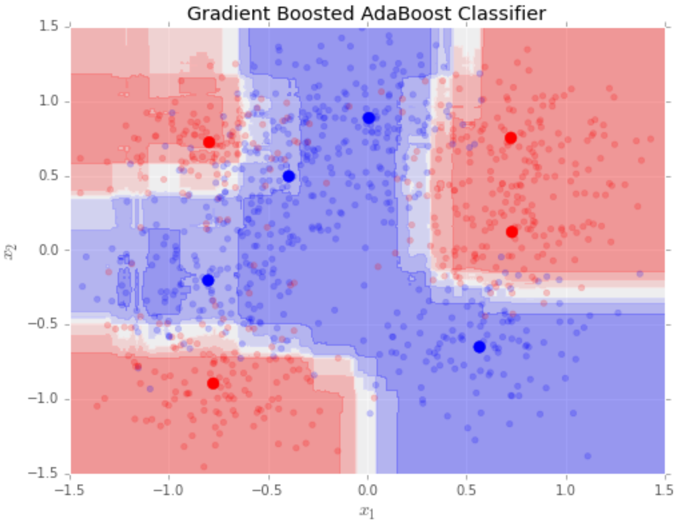
Traditional Adaboost is outclassed by gradient boosting, but retains historical and mathematical significance

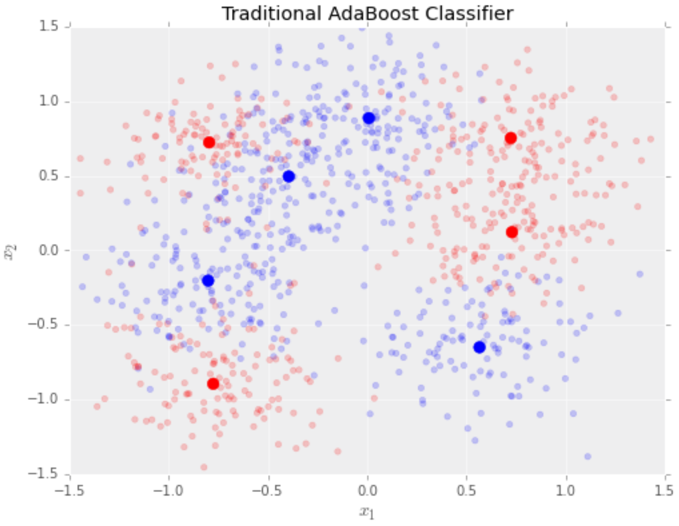


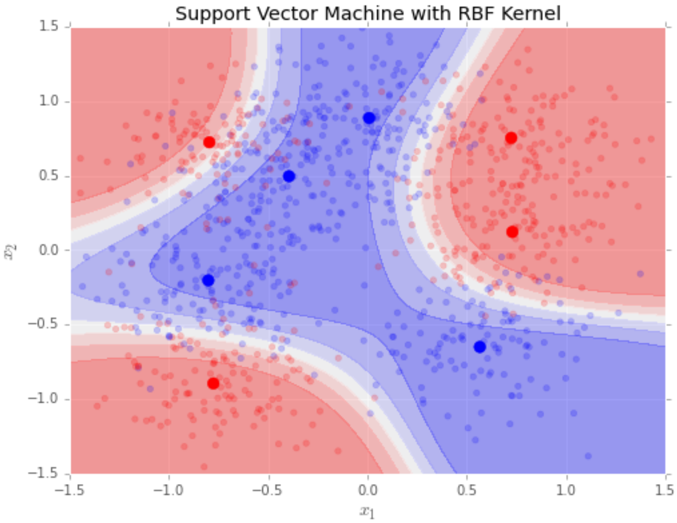
A Final Comparison of Classification Algorithms



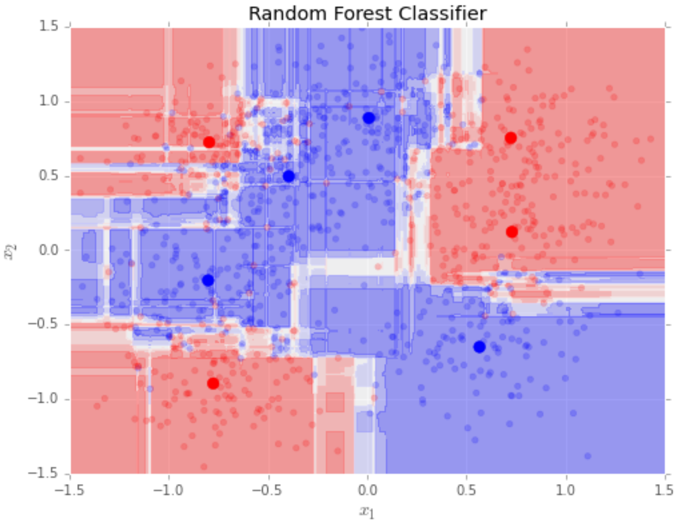

Boosted Logistic Classifier
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
x_2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x_1


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).

Q&A

- Understand the conceptual foundation of Boosting
- Understand the algorithm's hyperparameters, and how to tune them.
- Understand some basic strategies for interpreting a booster.
- Understand the drawbacks of boosting.
- Be aware of the possibility of creating your own loss function.

Appendix: Generalizing the Algorithm

Recall the boosting algorithm
Algorithm: Gradient Boosting to Minimize Sum of Squared Errors.

Inputs: A data set $\{x_i, y_i\}$.
Returns: A function f such that $f(x_i) \approx y_i$, minimizing least squared errors.

- Initialize $f_0(x) = \frac{1}{N}\sum_i y_i$.
- Iterate (parameter k) until satisfied:
  - Create the working data set $W_k = \{x_i, y_i - f_k(x_i)\}$.
  - 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 $f_{\max}(x) = f_0(x) + T_1(x) + T_2(X) + \cdots + T_{\max}(x)$.

Remember how we got here.

We are trying to minimize least squared loss $L(f, y) = \frac{1}{2}(y - f)^2$.

The gradient is $L(f, y) = f - y$.

So maybe this works...

- Initialize $f_0(x) = \frac{1}{N} \sum_i y_i$.
- Iterate (parameter k) until satisfied:
  - Create the working data set $W_k = \{x_i, -\nabla_f L(f(x_i), y)\}$.
  - 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 $f_{\max}(x) = f_0(x) + T_1(x) + T_2(X) + \cdots + T_{\max}(x)$.

Close, but not quite...

Improvement One:

We chose to initialize to the average value of y_i because it is the constant that minimized least squares.
Now that we have a different loss function, there's probably a better choice.

Improvement One:

Initialize $f_0(x)$ to the constant μ minimizing $\sum_i L(\mu, y_i)$.

Improvement Two: (Really the same improvement as one)

In our update step we added the fit tree directly
$f_{k+1}(x) = f_k(x) + T_k(x)$.

The predictions from a regression tree are the *average of of the response in the terminal nodes*.

This happened to be the minimizer of the loss function (least squares) for those sample laying in some terminal node.

Improvement Two:

Better not to directly add the value that minimizes the loss function we care about.

For each x in a terminal node R, set $f_{k+1}(x) = f_k(x) + \mu$, where μ is chosen to minimize the loss function over all those training samples that lie in R:

$$\mu = \arg\min_t \sum_{x_i \in R} L(y_i, f_k(x_i) + t)$$

Appendix: Gradient Boosted Logistic Regression Derivation

Let's follow through our derivation of boosting and see what we get.

Question One: How to initialize f_0?

For squared error minimization, we initialized to the average value of y_i.

Wait, why did we do that?

The average of y_i is the *constant* μ that minimizes the sum of squared errors.

$\frac{1}{N}\sum_i y_i$ is the constant that minimizes

$$\sum_i (y_i - \mu)^2 = \sum_i L(\mu, y_i)$$

Ok, so maybe we should do the same thing here?

$$f_0 = \arg\min_{\mu} \sum_i L(\mu, y_i) = \sum_i (y_i \mu + \log(1 + e^{\mu}))$$

Question: What is the minimizer (hint, what happens if you put *only* an intercept in logistic regression)?

The minimizer is the *sample log odds*:

$$f_0 = log\left(\frac{\frac{1}{N}\sum_i y_i}{1 - \frac{1}{N}\sum_i y_i}\right)$$

Question Two: What's the gradient?

$$\nabla_f L(f, y) = \nabla_f \left(yf + \log(1 + e^f) \right) = y + \frac{e^f}{1 + e^f}$$

That last term is familiar...

$$\frac{e^f}{1 + e^f} = p$$

Where p is the probability associated with the log-odds f.