Module 3: Boosted Algorithms

Data Science Immersive

FLATIRON SCHOOL

Lesson Overview

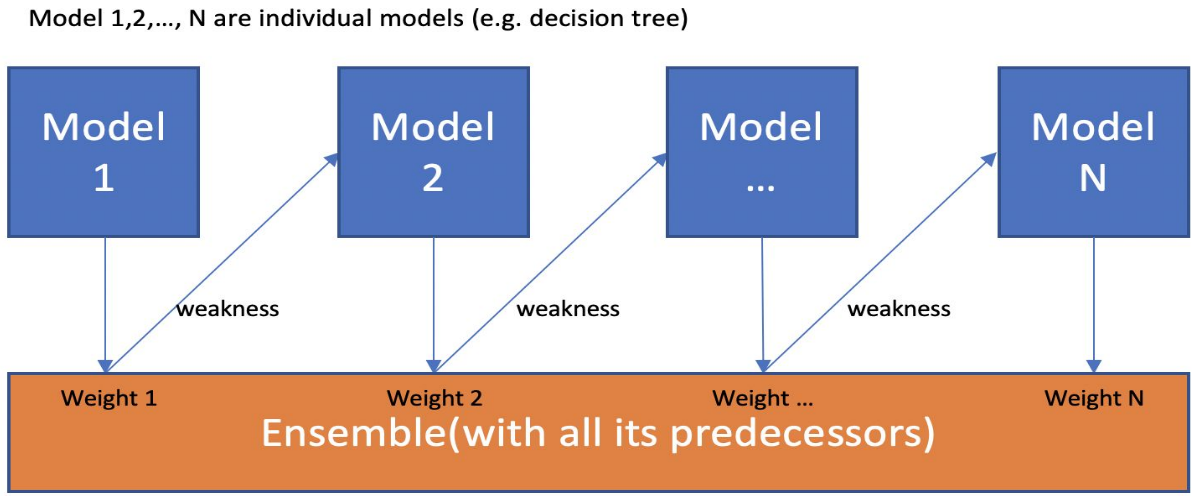
Aim: SWBAT differentiate between bagging and boosting ensemble methods, as well as explain how a two specific boosting methods (AdaBoost and Gradient Boosting) differ.

Agenda:

- Review Ensemble Methods
- Learn about the AdaBoost method
- Learn about the Gradient Boosting

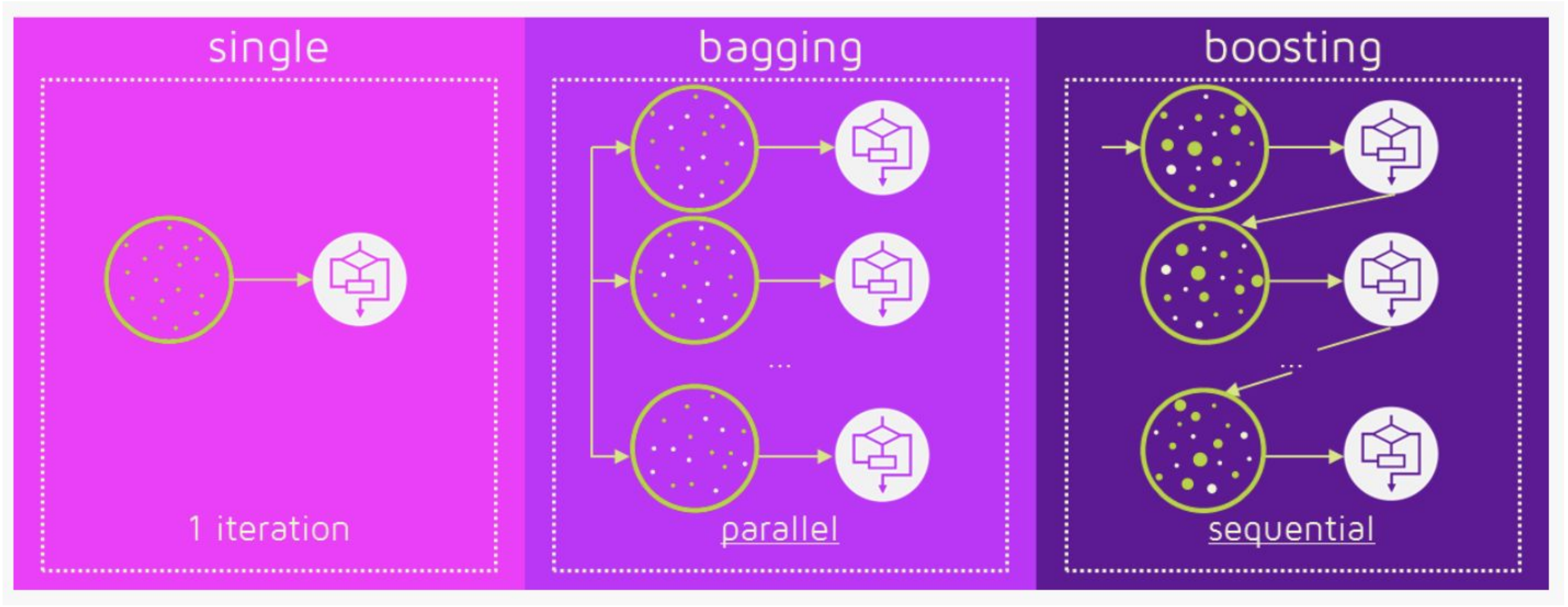
Boosting

Boosting (originally called hypothesis boosting) refers to any ensemble method that train predictors sequentially, each trying to correct its predecessor in some fashion.


Model 1,2,…, N are individual models (e.g. decision tree)
Model 1
Model 2
Model …
Model N
weakness
weakness
weakness
Weight 1
Weight 2
Weight …
Weight N
Ensemble(with all its predecessors)

There are many boosting methods available, but by far the most popular are AdaBoost(short for Adaptive Boosting) and Gradient Boosting.

Bagging vs. Boosting



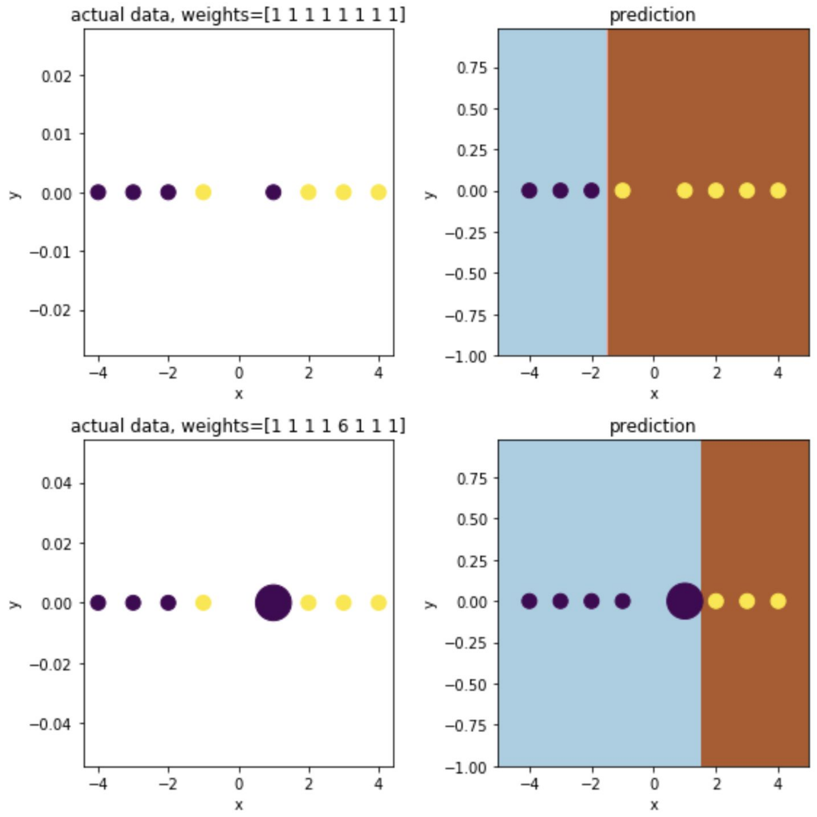
Boosting Models

- Boosting **is a generic algorithm rather than a specific model**. Boosting needs you to specify a weak model (e.g. regression, shallow decision trees, etc) and then improves it.
- This allowed different loss functions to be used, expanding the technique beyond binary classification problems to support regression, multi-class classification and more.
- **Boosting** is a statistical framework where the objective is to minimize the loss of the model by adding weak learners using a **gradient descent** like procedure.
- This class of algorithms were described as a **stage-wise additive** model. This is because one new weak learner is added at a time and existing weak learners in the model are frozen and left unchanged.

AdaBoost - Adaptive Boosting

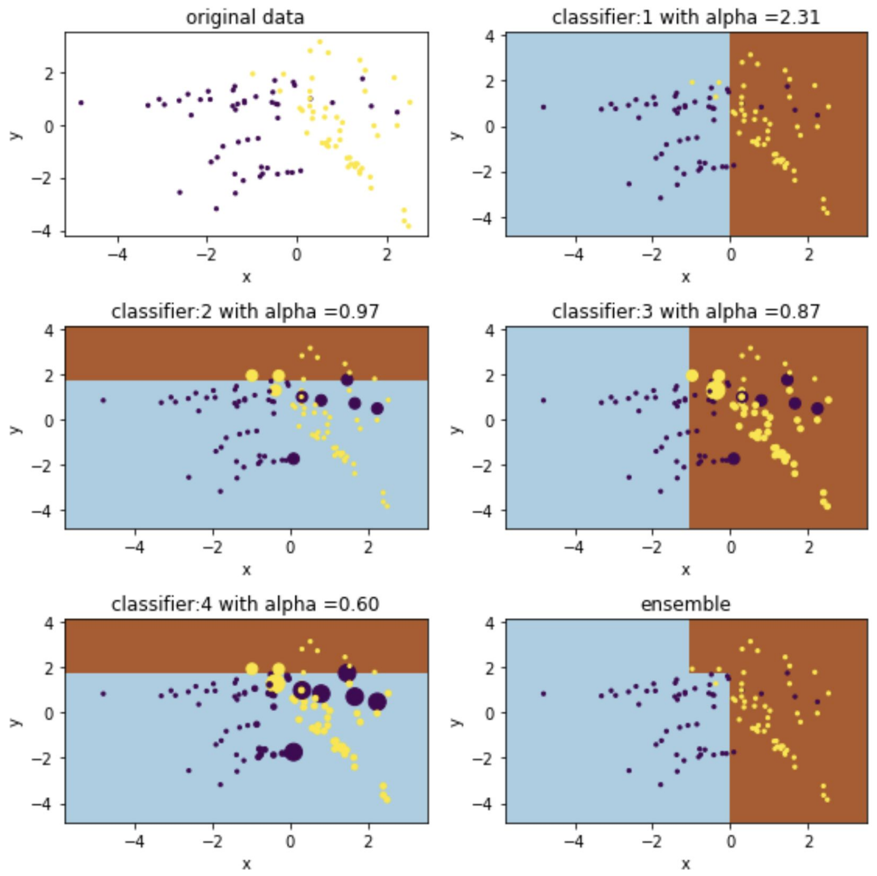
AdaBoost is a specific Boosting algorithm developed for classification problems (also called discrete AdaBoost). The weakness is identified by the weak estimator’s error rate.

In each iteration, AdaBoost identifies miss-classified data points, increasing their weights (and decrease the weights of correct points, in a sense) so that the next classifier will pay extra attention to get them right.



AdaBoost

AdaBoost trains a sequence of models with augmented sample weights, generating ‘confidence’ coefficients Alpha for individual classifiers based on errors. Low errors leads to large Alpha, which means higher importance in the voting.



Weighting the Classifiers

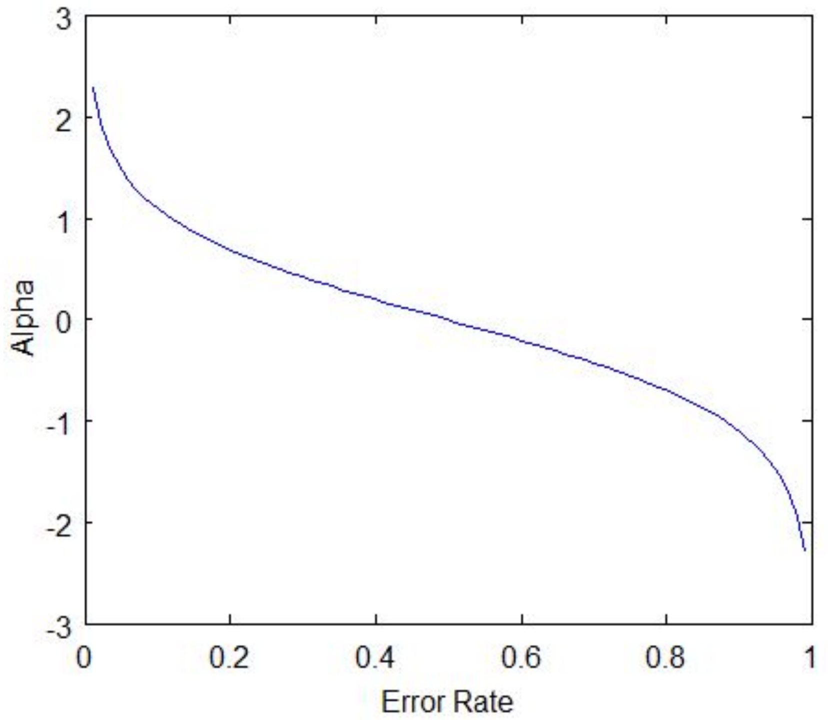
After each classifier is trained, a weight is assigned to the classifier on accuracy. More accurate classifier is assigned higher weight so that it will have more impact in final outcome.

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Equation for the final classifier

$$\alpha_t = \frac{1}{2} ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

Calculation of weight for each individual classifier



Classification Models

The first classifier is trained using a ***random subset*** of overall training set...

Misclassified item is assigned higher weight so that it appears in the training subset of next classifier with higher probability.

$$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

Data Preparation for AdaBoost

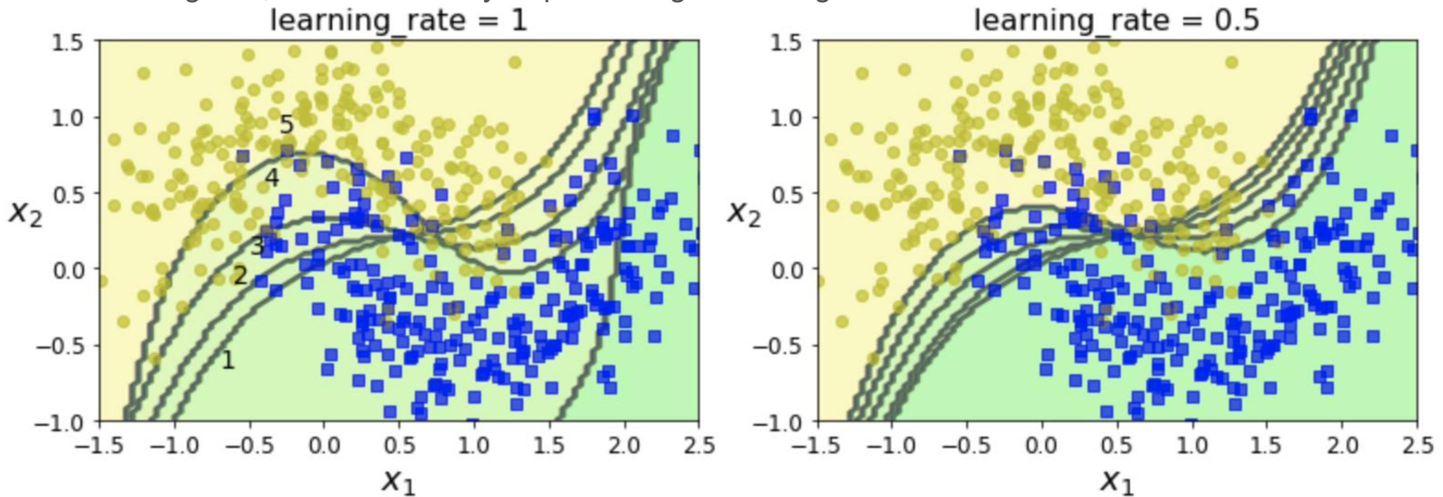
This section lists some heuristics for best preparing your data for AdaBoost.

- **Quality Data**: Because the ensemble method continues to attempt to correct misclassifications in the training data, you need to be careful that the training data is of a high-quality.
- **Outliers**: Outliers will force the ensemble down the rabbit hole of working hard to correct for cases that are unrealistic. These could be removed from the training dataset.
- **Noisy Data**: Noisy data, specifically noise in the output variable can be problematic. If possible, attempt to isolate and clean these from your training dataset.

Implementing AdaBoost

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```

AdaBoost also supports a learning rate that controls the contribution of each model to the ensemble prediction. More trees may require a smaller learning rate; fewer trees may require a larger learning rate.



AdaBoost Summary

AdaBoost sequentially trains classifiers.

Tries to improve classifier by looking at the misclassified instances.

The algorithm weights misclassified instance more so they are more likely to be included in the training data subset.

Each classifier is weighted based on their accuracy and then all are aggregated to create the final classifier.

Doesn’t perform well on very noisy data or data with outliers.

AdaBoost can use any type of classification model, not just a decision tree.

Gradient Boosting

Gradient boosting approaches the problem a bit differently. Instead of adjusting weights of data points, Gradient boosting focuses on the difference between the prediction and the ground truth.

Gradient boosting requires a differential loss function and works for both regression and classifications. I'll use a simple Least Square as the loss function (for regression). The algorithm for classifications shares the same idea, but the math is slightly more complicated.

Steps in Gradient Boosting

1. Fit a simple linear regression or decision tree on data **[call x as input and y as output]**

2. Calculate error residuals. **[e1= y - y_predicted1]**

3. Fit a new model on error residuals as target variable with same input variables **[call it e1_predicted]**

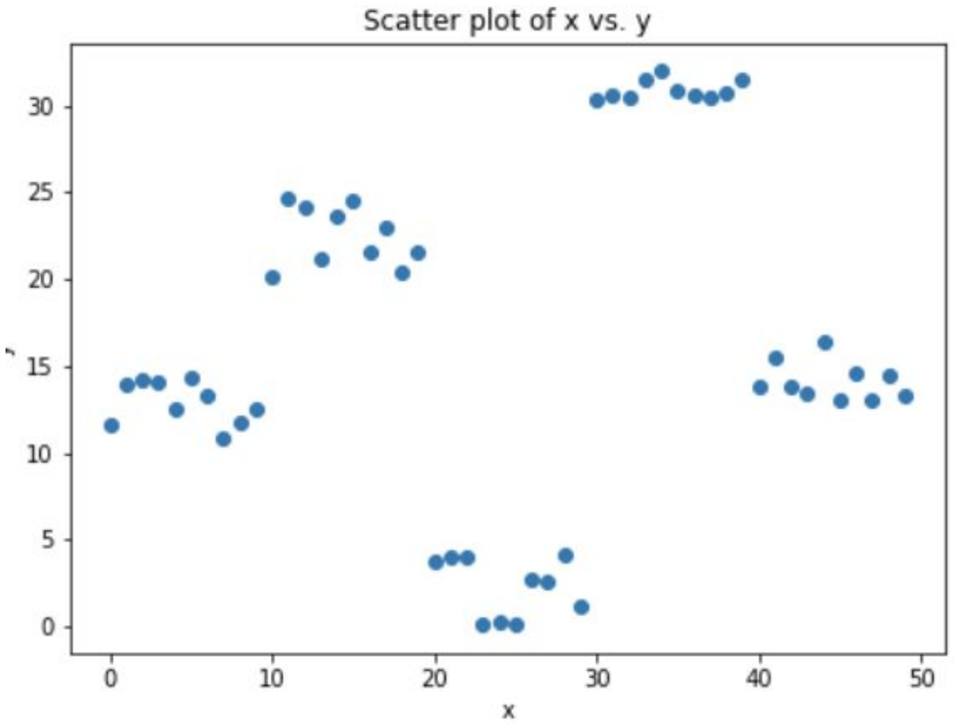
4. Add the predicted residuals to the previous predictions

[y_predicted2 = y_predicted1 + e1_predicted]

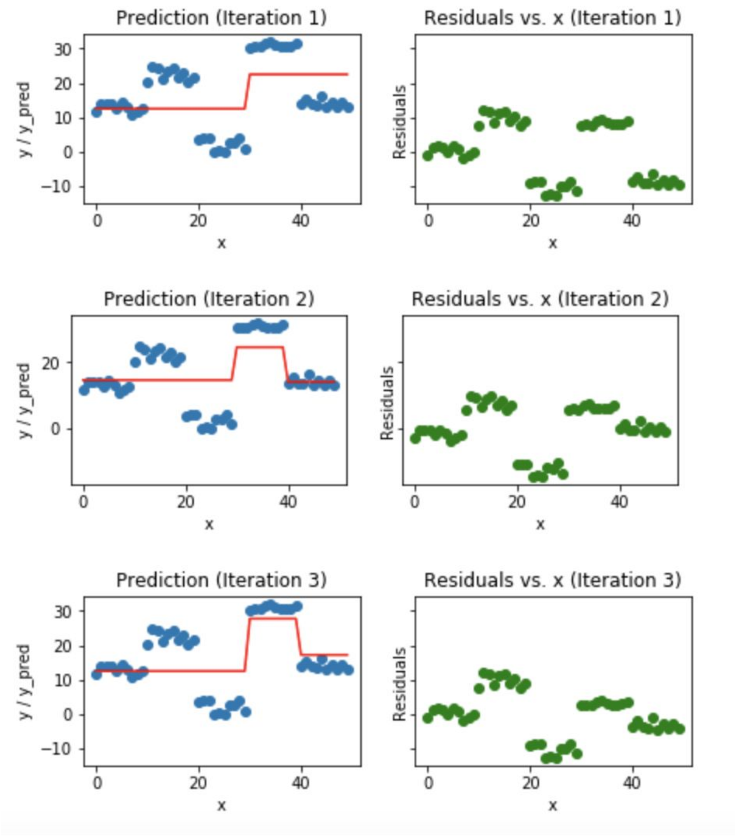
5. Fit another model on residuals that is still left. i.e. **[e2 = y - y_predicted2]**

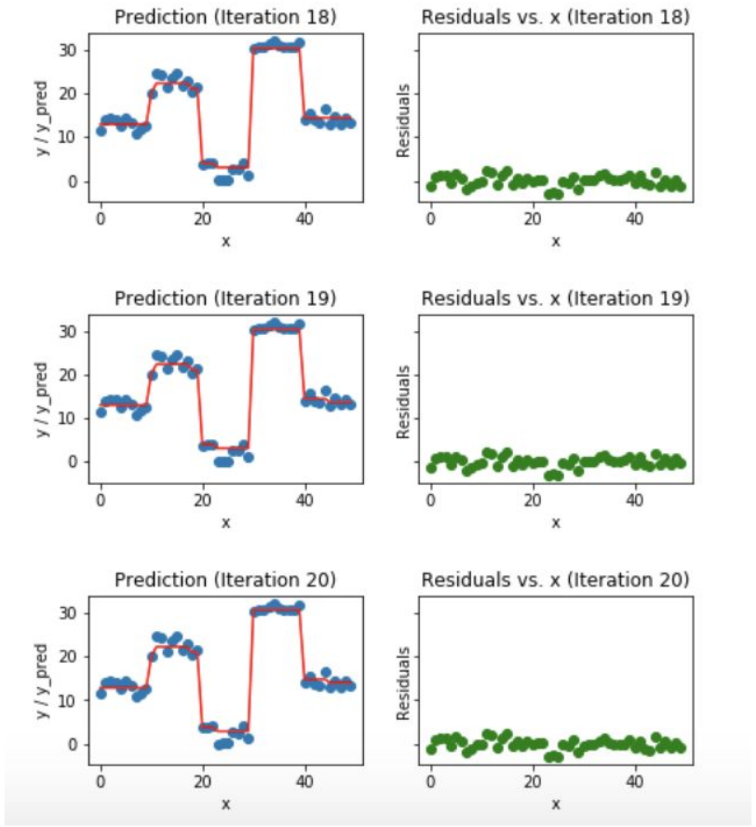
Repeat steps 2 to 5 until it starts overfitting or the sum of residuals become constant.

Overfitting can be controlled by consistently checking accuracy on validation data.


Scatter plot of x vs. y
0
5
10
15
20
25
30
0
10
20
30
40
50
x
y

Gradient Boosting over Iterations





Manual Implementation of Gradient Boosting

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Now train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

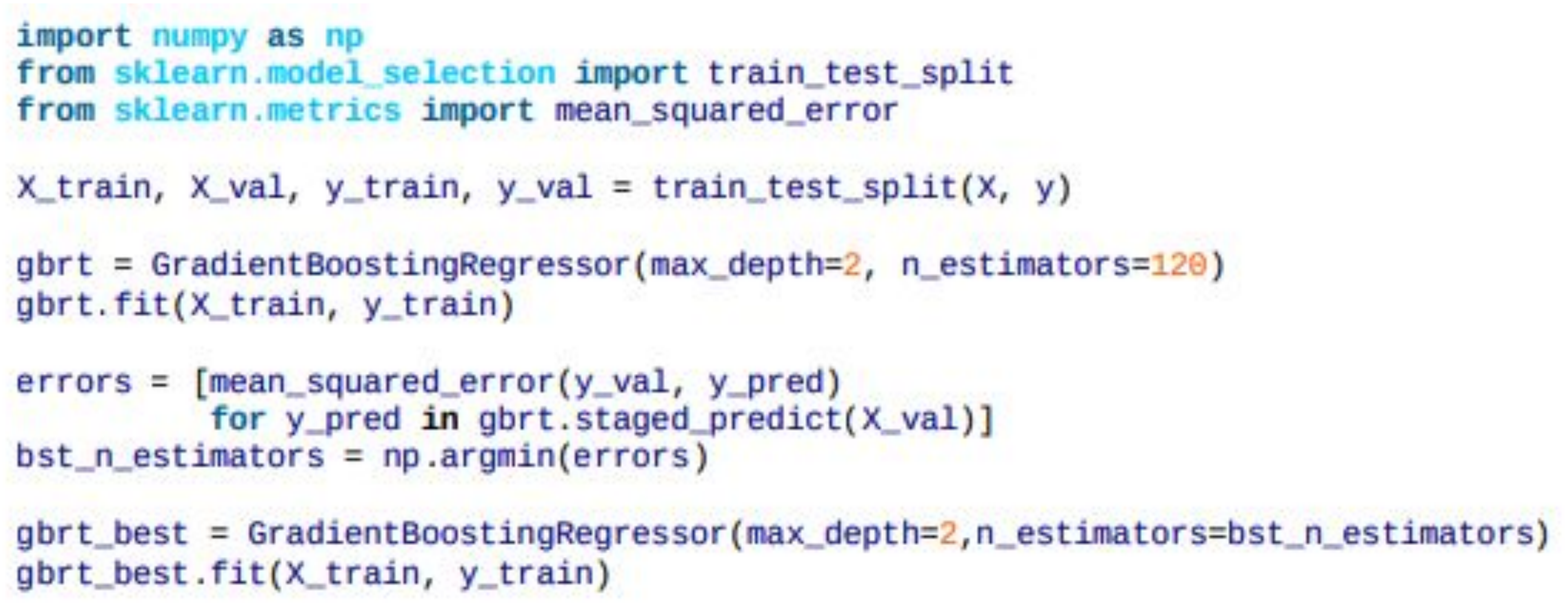
Then we train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

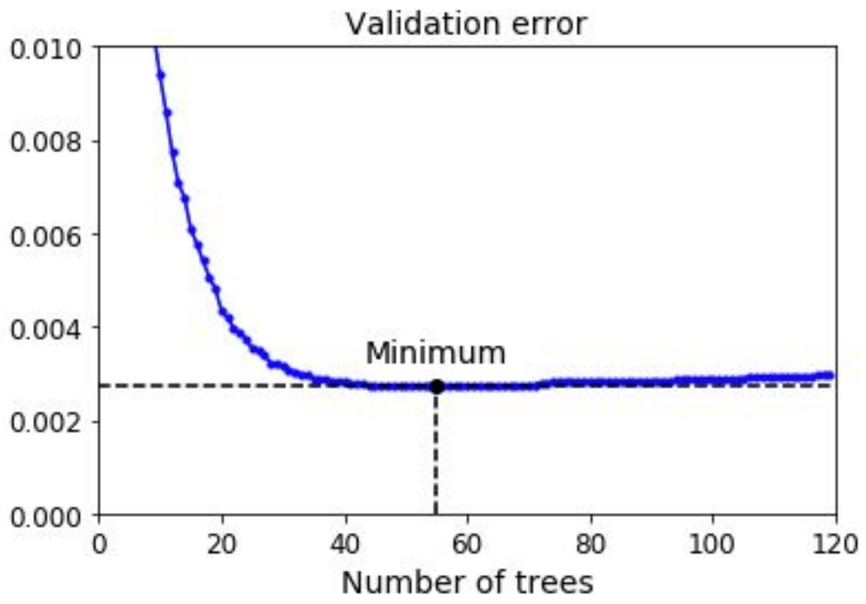
```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

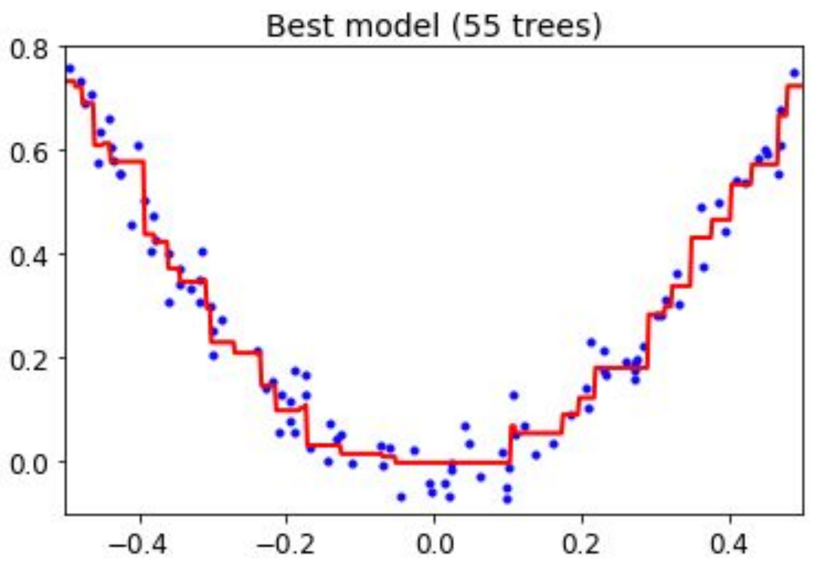
Code Implementation



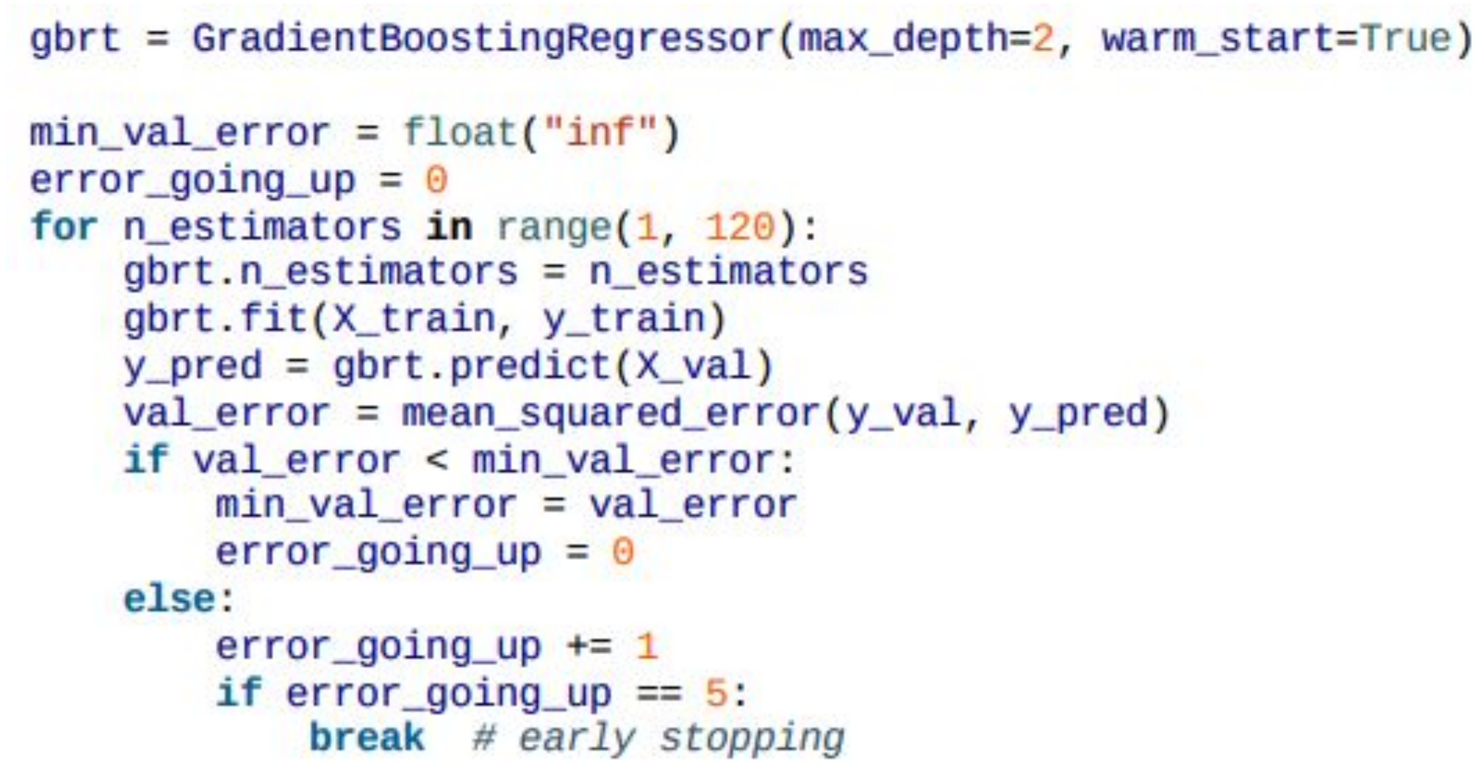
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors)

gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

Validation Error over Iterations





Early Stopping to prevent overfitting



```python
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping
```

Shrinkage - Learning Rate

The contribution of each tree to this sum can be weighted to slow down the learning by the algorithm. This weighting is called a **shrinkage** or a **learning rate**.

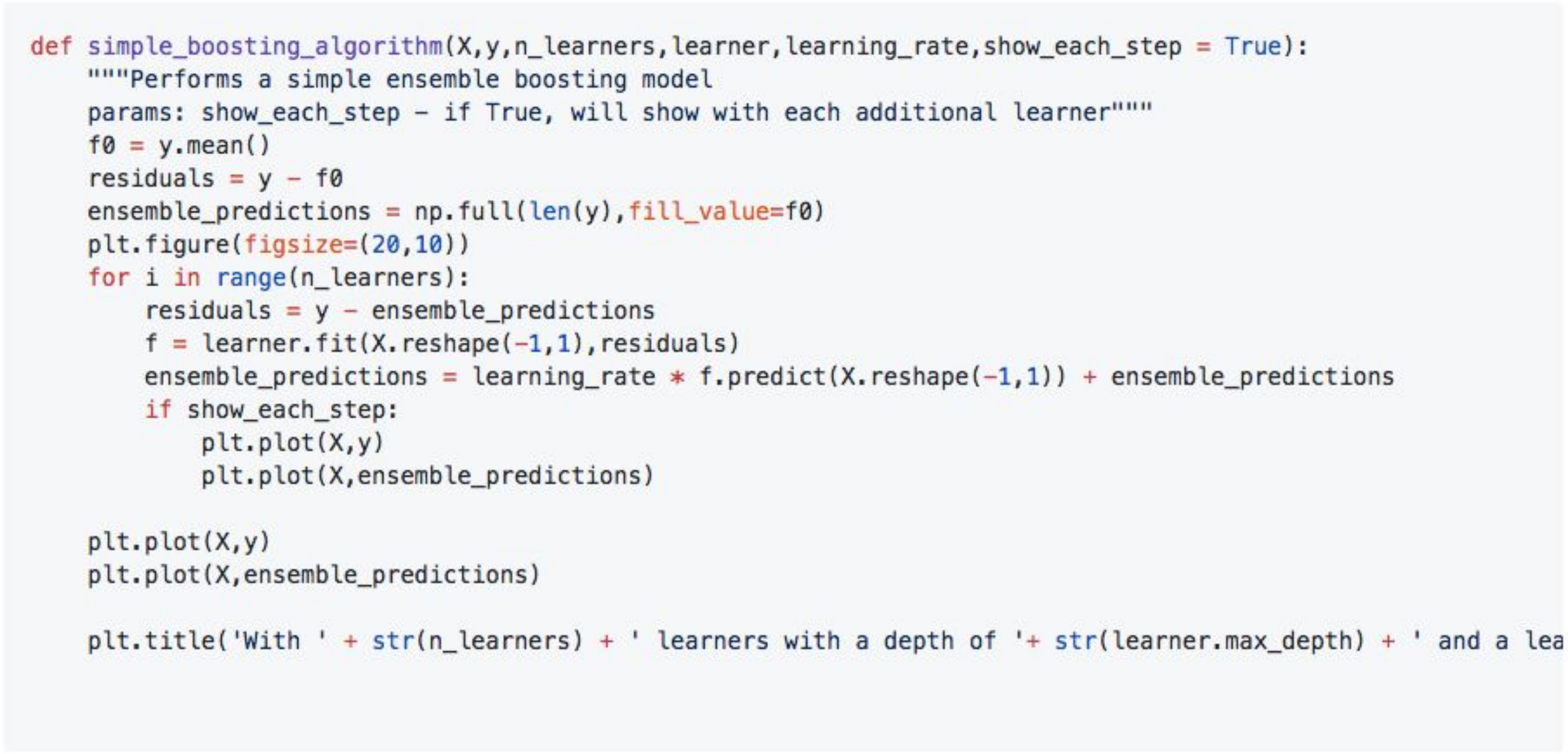
For each gradient step, the step magnitude is multiplied by a factor between 0 and 1

Shrinkage causes sample-predictions to slowly converge toward observed values.

As this slow convergence occurs, samples that get closer to their target end up being grouped together into larger and larger leaves (due to fixed tree size parameters), resulting in a natural regularization effect.

Shrinkage Visualized

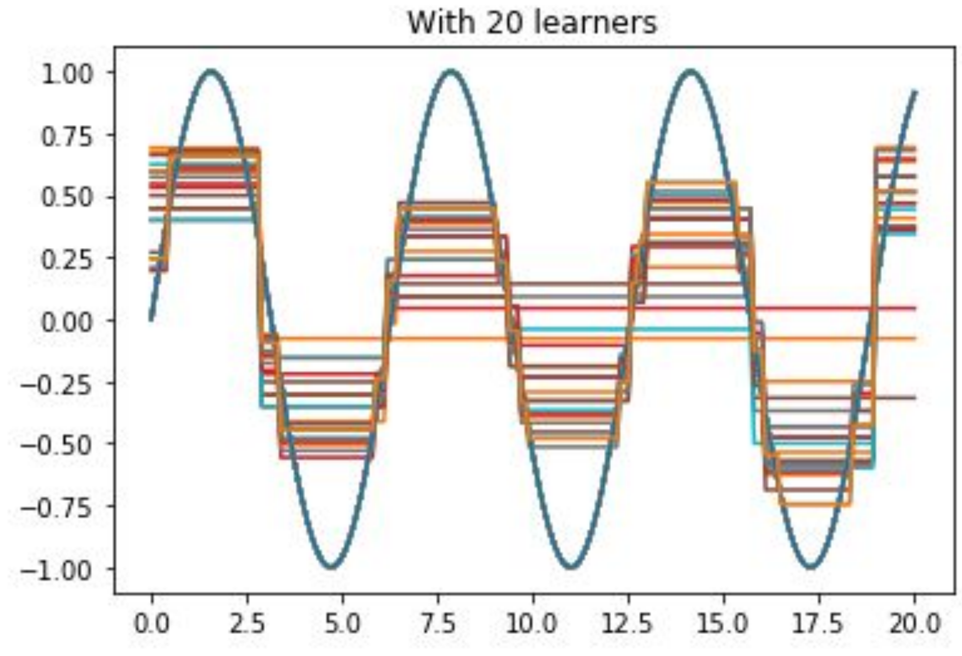
https://github.com/fpolchow/boosting_notes



```
def simple_boosting_algorithm(X,y,n_learners,learner,learning_rate,show_each_step = True):
    """Performs a simple ensemble boosting model
    params: show_each_step - if True, will show with each additional learner"""
    f0 = y.mean()
    residuals = y - f0
    ensemble_predictions = np.full(len(y),fill_value=f0)
    plt.figure(figsize=(20,10))
    for i in range(n_learners):
        residuals = y - ensemble_predictions
        f = learner.fit(X.reshape(-1,1),residuals)
        ensemble_predictions = learning_rate * f.predict(X.reshape(-1,1)) + ensemble_predictions
        if show_each_step:
            plt.plot(X,y)
            plt.plot(X,ensemble_predictions)

    plt.plot(X,y)
    plt.plot(X,ensemble_predictions)

    plt.title('With ' + str(n_learners) + ' learners with a depth of '+ str(learner.max_depth) + ' and a lea
```

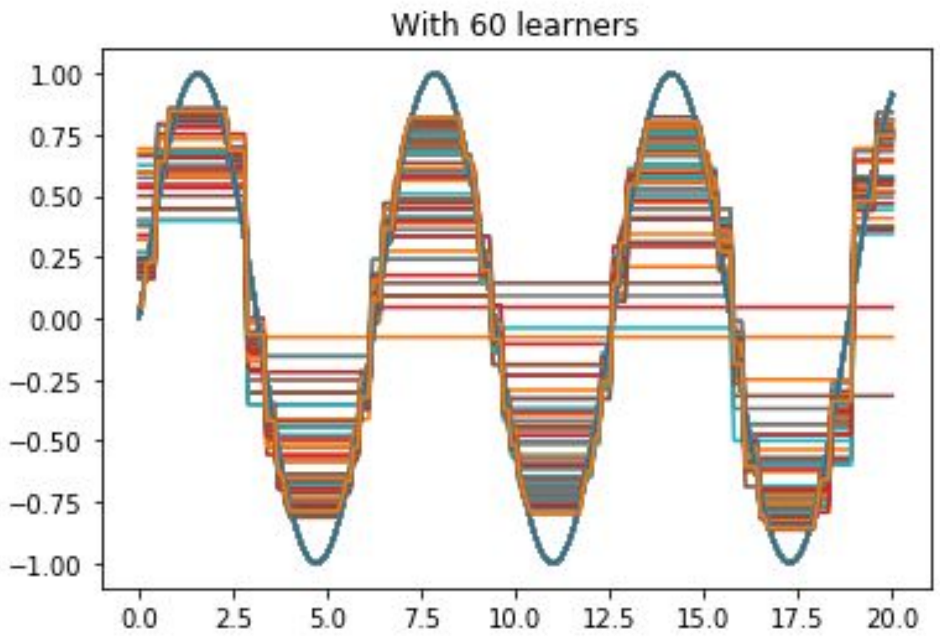
Shrinkage - Slow Convergence

```
simple_boosting_algorithm(X,np.sin(X),20,tree.DecisionTreeRegressor(max_depth=1),0.001)
```



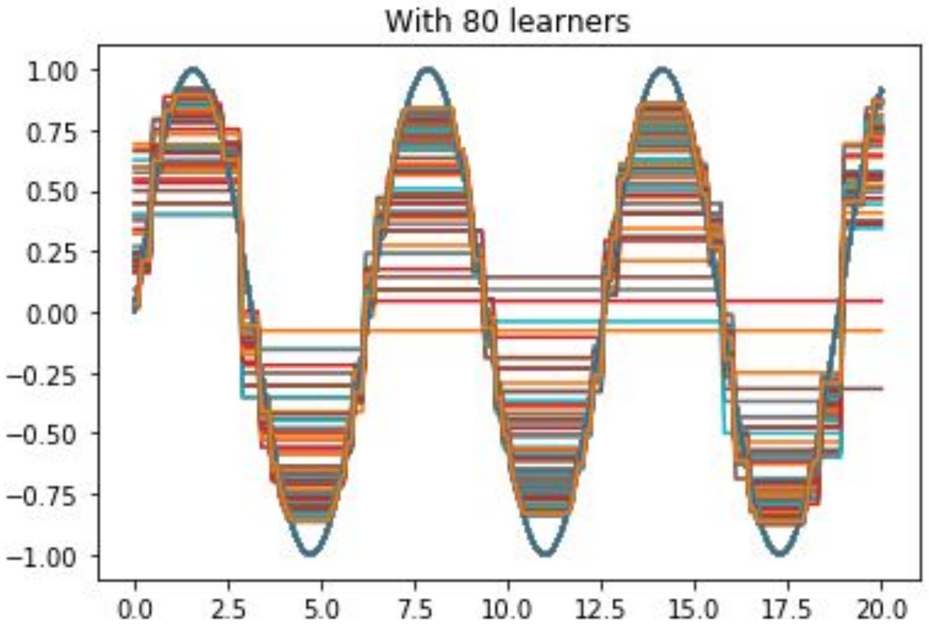
Shrinkage - Slow Convergence

```
simple_boosting_algorithm(X,np.sin(X),60,tree.DecisionTreeRegressor(max_depth=1),0.001)
```



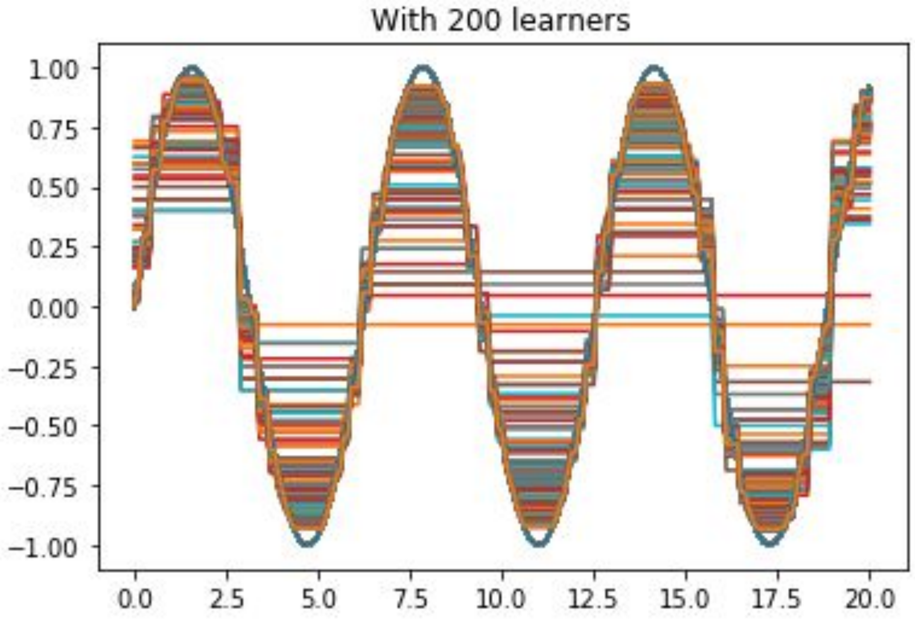
Shrinkage - Slow Convergence

```
simple_boosting_algorithm(X,np.sin(X),80,tree.DecisionTreeRegressor(max_depth=1),0.001)
```

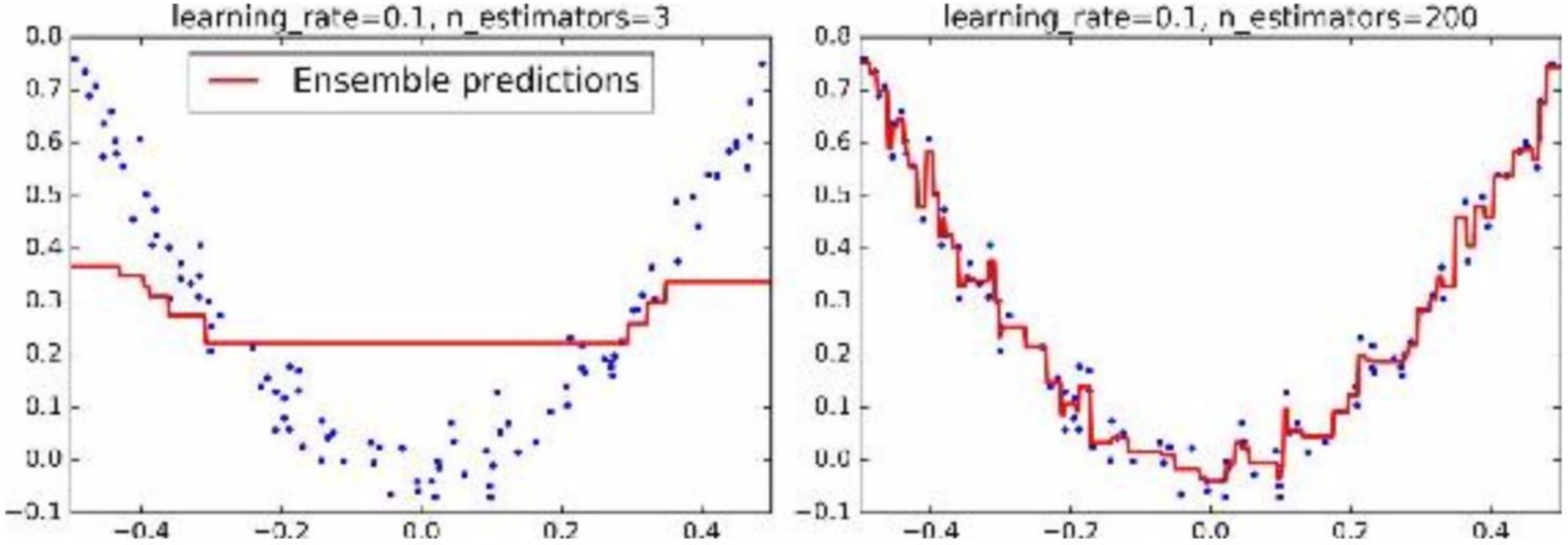


Shrinkage - Slow Convergence

```
simple_boosting_algorithm(X,np.sin(X),200,tree.DecisionTreeRegressor(max_depth=1),0.001)
```



The Effect of Shrinkage



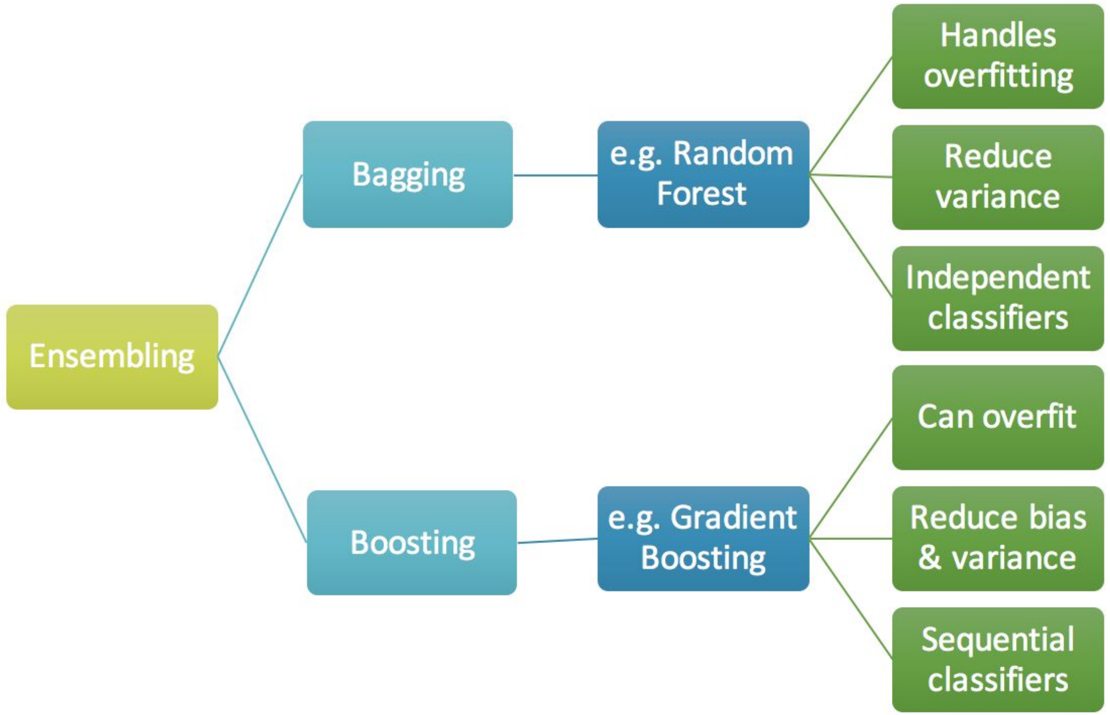
XGBoost

Both xgboost and gbm follows the principle of gradient boosting. There are however, the difference in modeling details. Specifically, xgboost used a more regularized model formalization to control over-fitting, which gives it better performance.

The name xgboost, though, actually refers to the engineering goal to push the limit of computations resources for boosted tree algorithms. Which is the reason why many people use xgboost.

- Has its own library -- not part of SKLearn
- Harder, better, faster, stronger
- ANY cost function, and more parameters
- Must be twice-differentiable because it uses something called a “Hessian” in the gradient descent algorithm. This is like another weight in addition to the learning rate, just like Shrinkage
- Built-in regularization (L2 by default) based on node weights.

Bagging vs. Boosting



Great Resources

Boosting Models:
https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/

AdaBoost: http://mccormickml.com/2013/12/13/adaboost-tutorial/

Gradient Descent: https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d

Comparison of Gradient Boosting and XGBoost:
https://medium.com/@gabrieltseng/gradient-boosting-and-xgboost-c306c1bcfaf5

Any Questions?