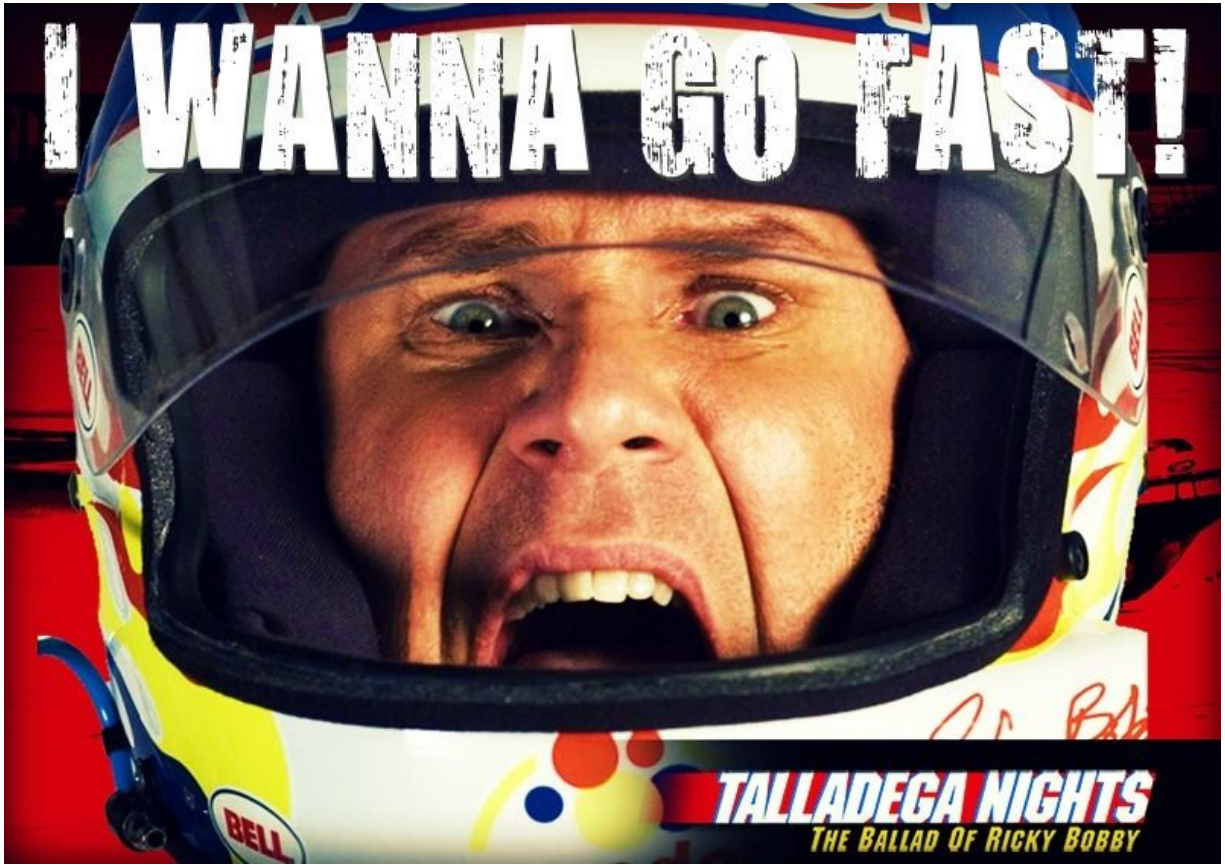
Module 3: Boosted Algorithms

December 18, 2018

FLATIRON SCHOOL


I WANNA GO FAST!
TALLADEGA NIGHTS
THE BALLAD OF RICKY BOBBY
BELL

Lesson Overview

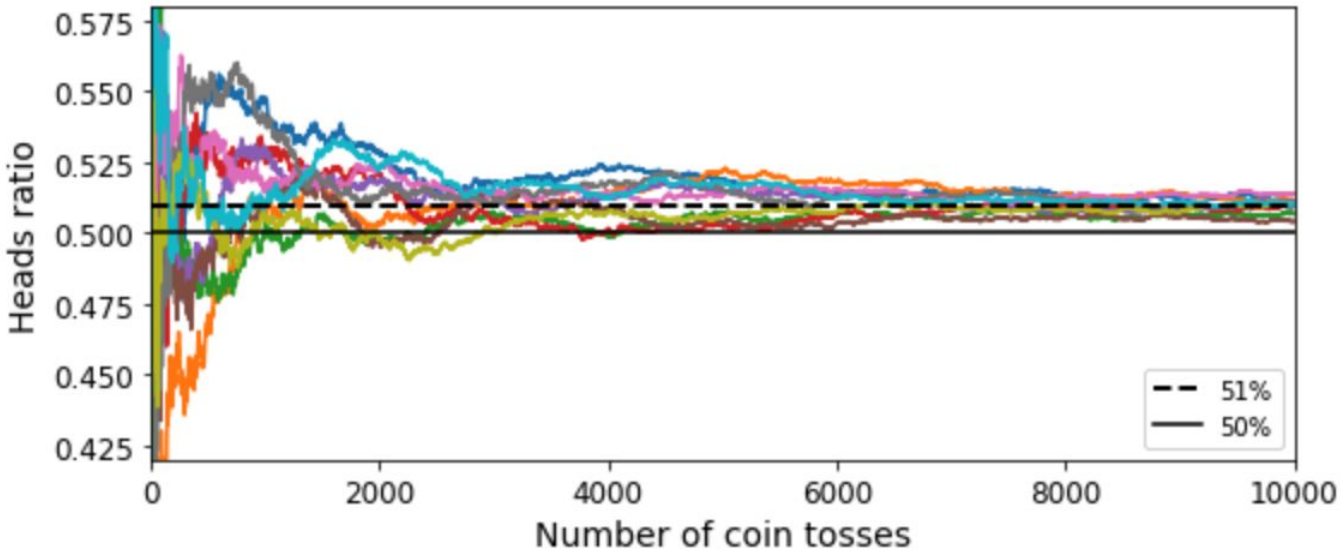
Aim: SWBAT differentiate between bagging and boosting ensemble methods, as well as explain how a two specific boosting methods (AdaBoost and Gradient Boosting) differ.

Agenda:

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

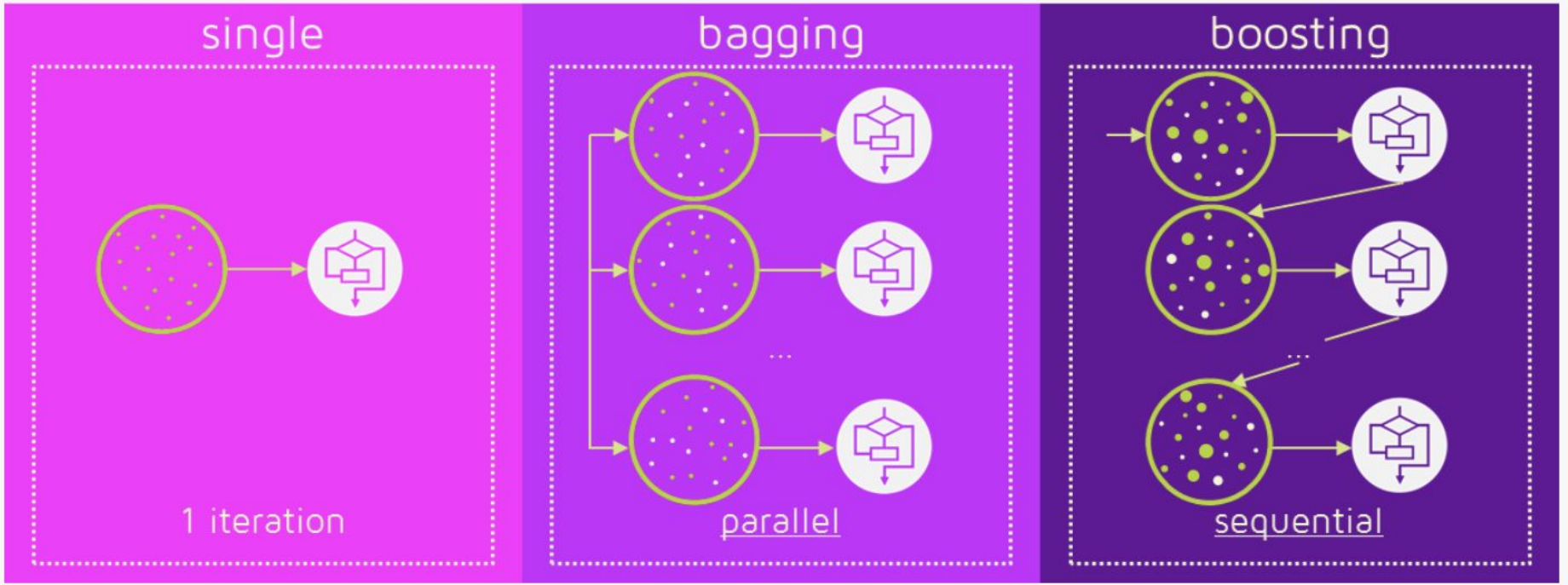
Why Ensemble Methods

Shower Thoughts: Which would you rather use to solve a problem, a bunch of simple models that aren't that accurate or one complex model that is very accurate?

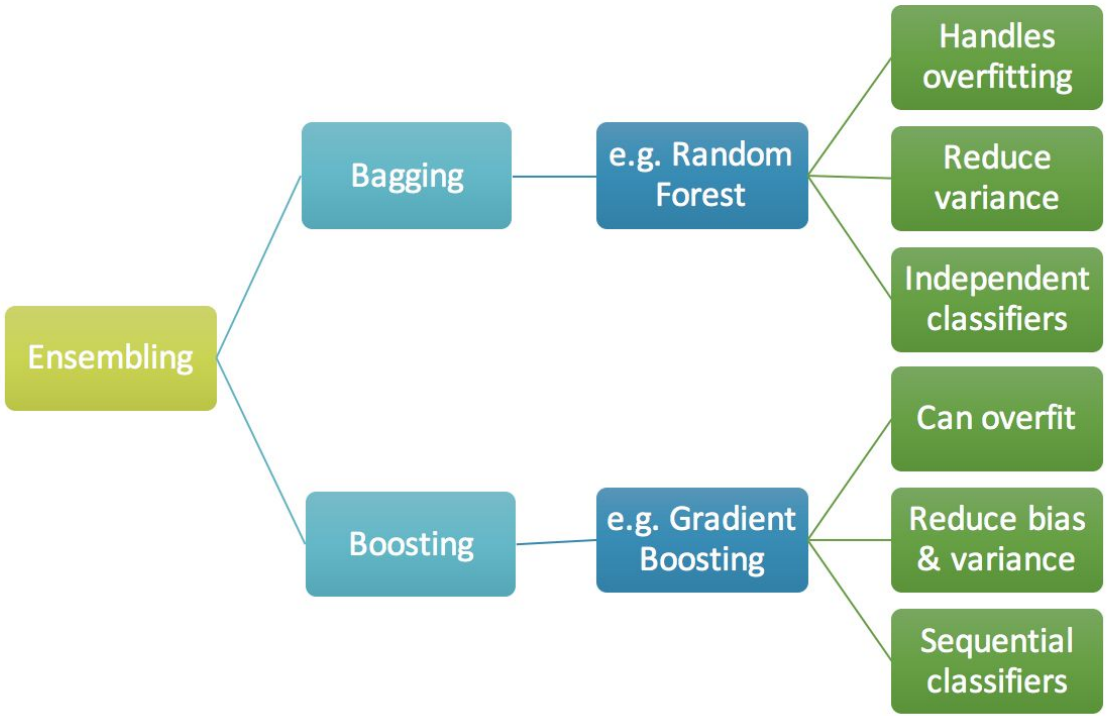


You have a biased coin that has a 51% chance of coming up heads and a 49% chance of coming up tails. If you toss it 1,000 times the probability of having a majority of heads is 75%. If you increase your tosses to 10,000, that probability rise to 97%

Bagging vs. Boosting



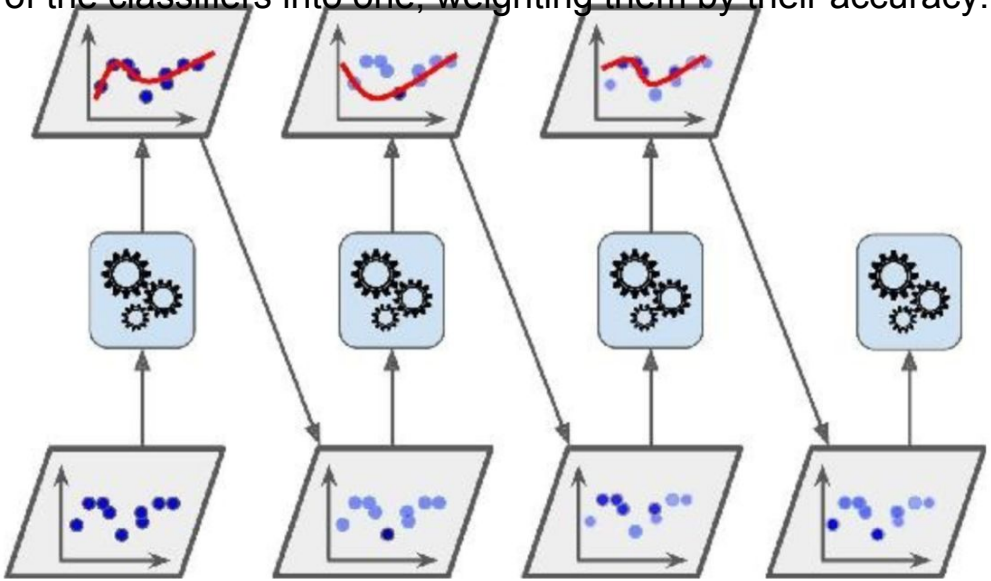
Bagging vs. Boosting



AdaBoost - Adaptive Boosting

AdaBoost is a ensemble method used to solve classification problems.

1. Fit a base classifier like a Decision tree.
2. Use that classifier to make predictions on the training set.
3. Increase the relative weight of the of the instances that were misclassified
4. Train another classifier using the updated weights.
5. Aggregate all of the classifiers into one, weighting them by their accuracy.



Weighting the Classifiers

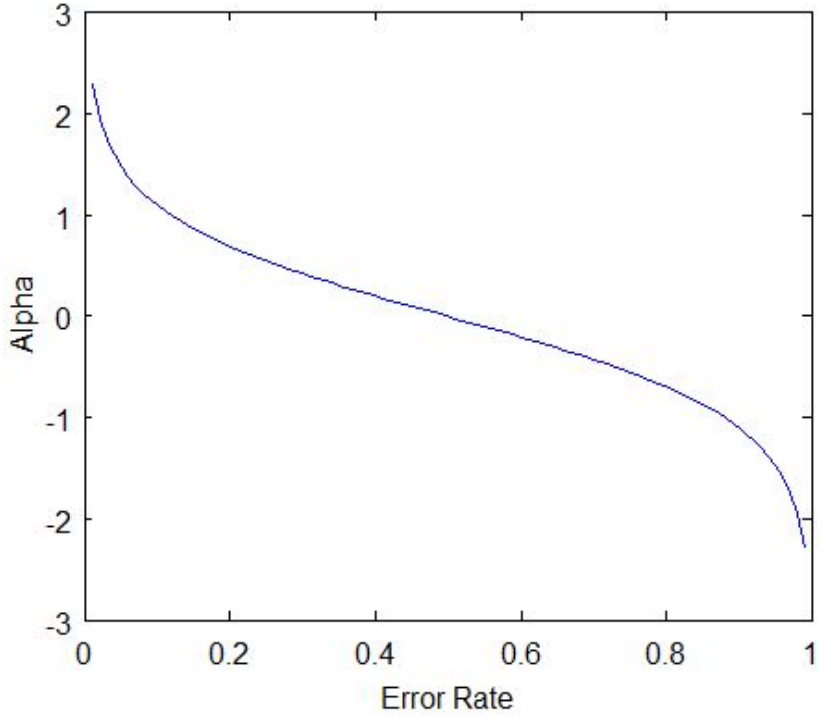
After each classifier is trained, a weight is assigned to the classifier on accuracy. More accurate classifier is assigned higher weight so that it will have more impact in final outcome.

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

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

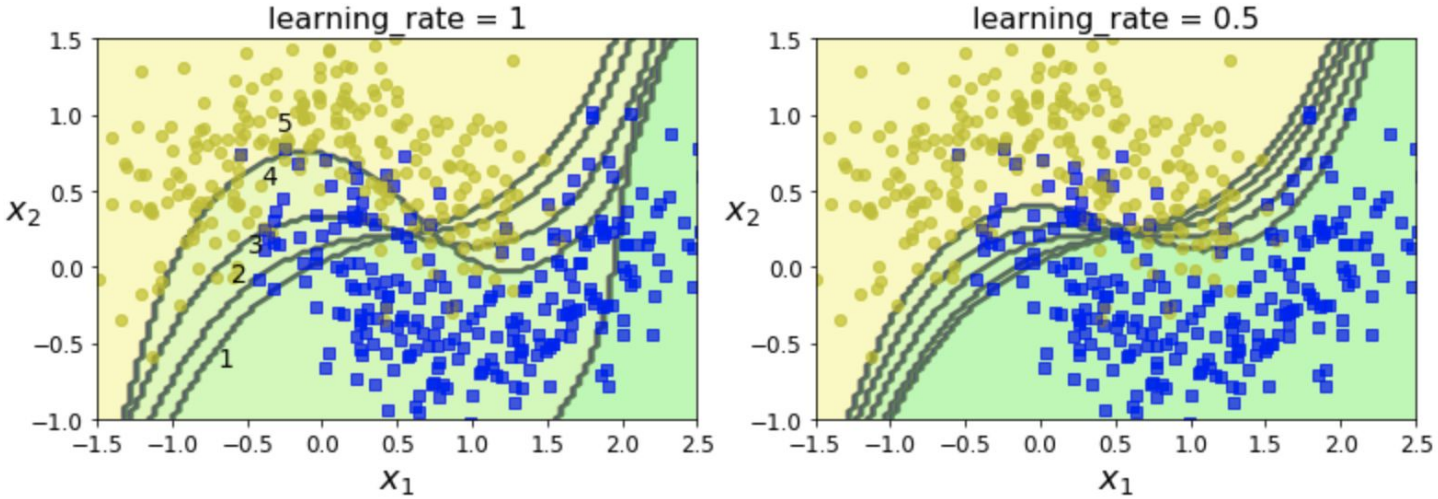
This section lists some heuristics for best preparing your data for AdaBoost.

- **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)
```

The learning rate changes the weight of the misclassified instances.



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

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 allowed different loss functions to be used, expanding the technique beyond binary classification problems to support regression, multi-class classification and more.

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. As opposed to **stepwise** approaches that readjust previously entered terms when new ones are added.

Steps in Gradient Boosting

1. Fit a simple linear regressor 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.

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:

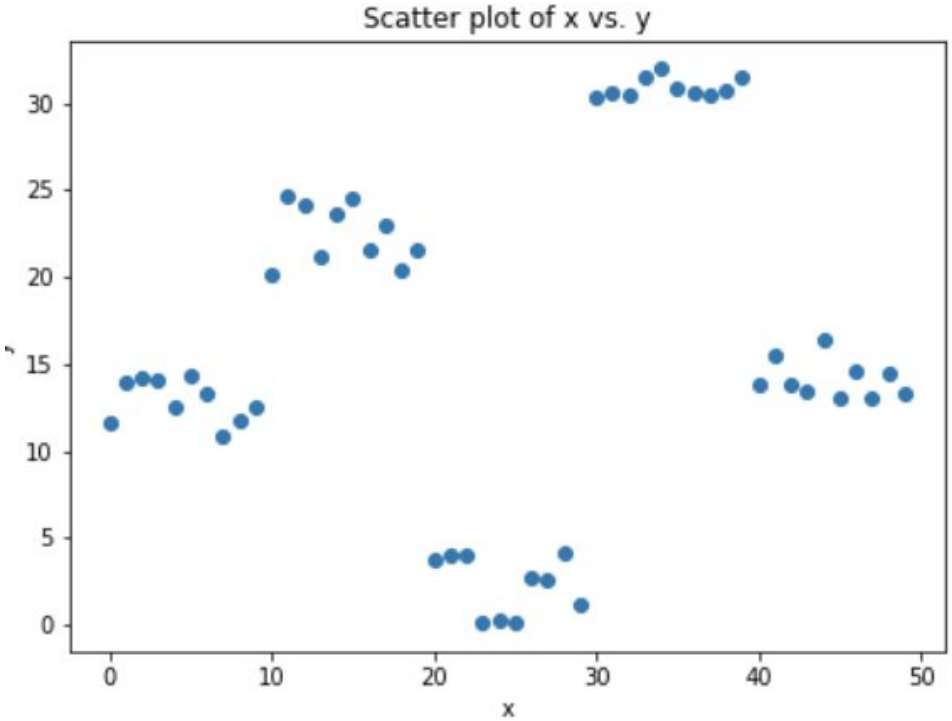
```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

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

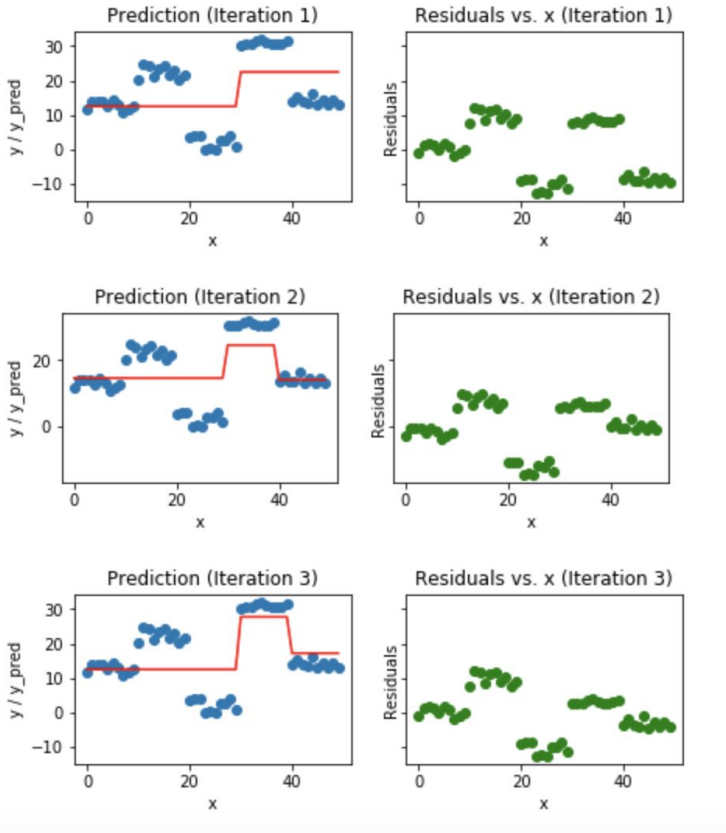
```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

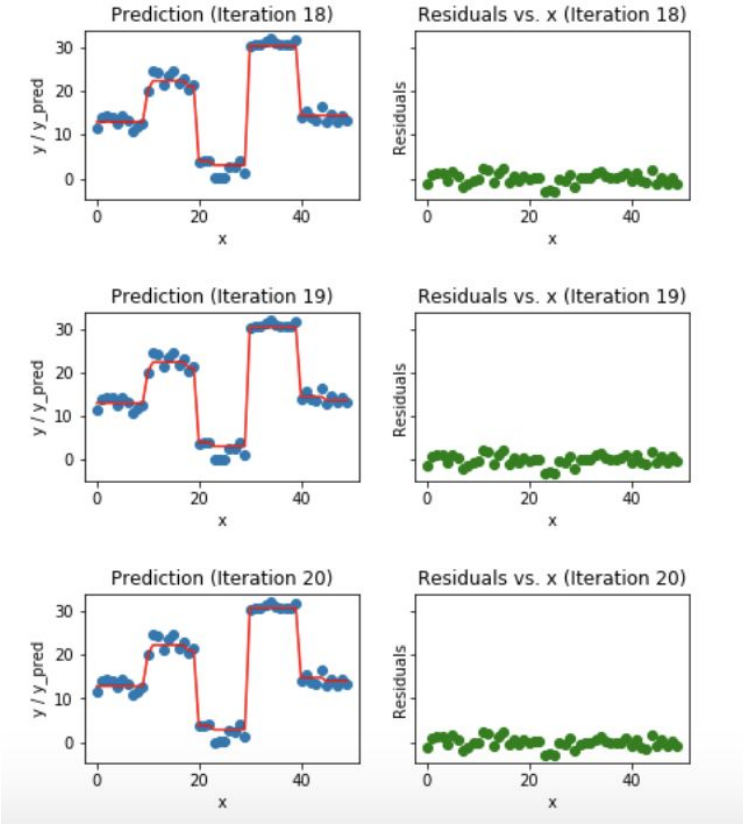
Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

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


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

Gradient Boosting over Iterations

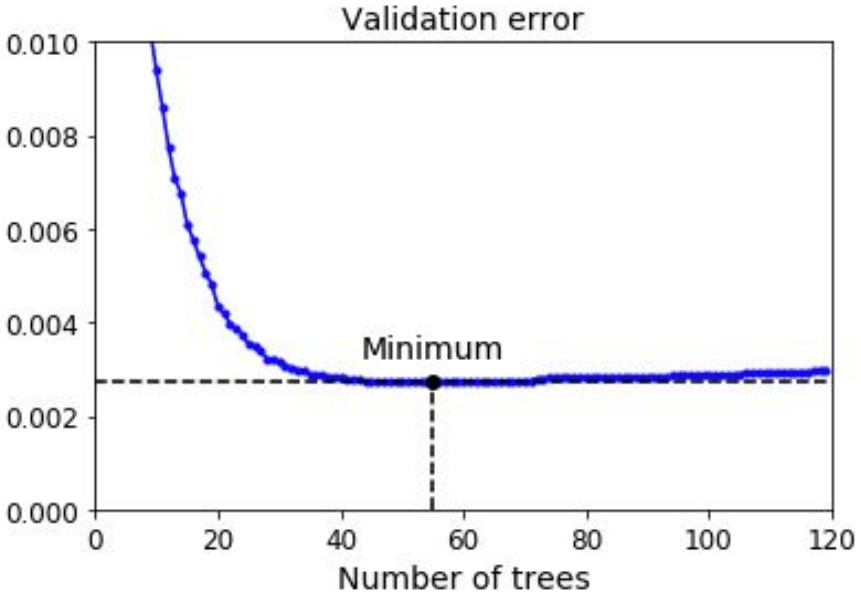


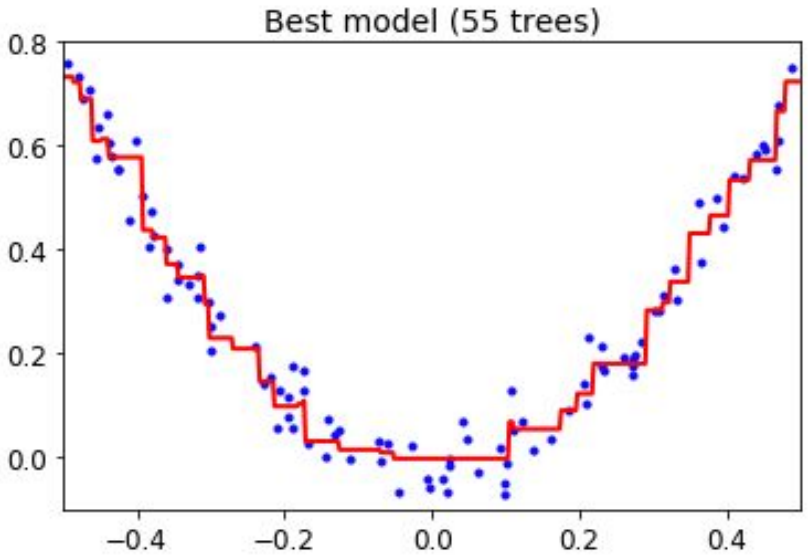


Code Implementation

```
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

```
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.

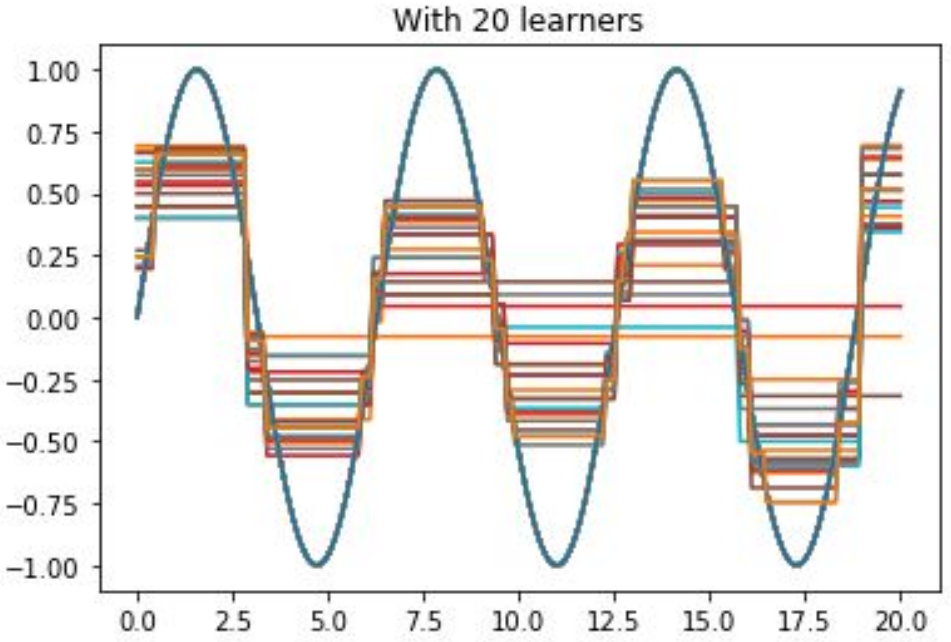
Shrinkage Visualized

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
```

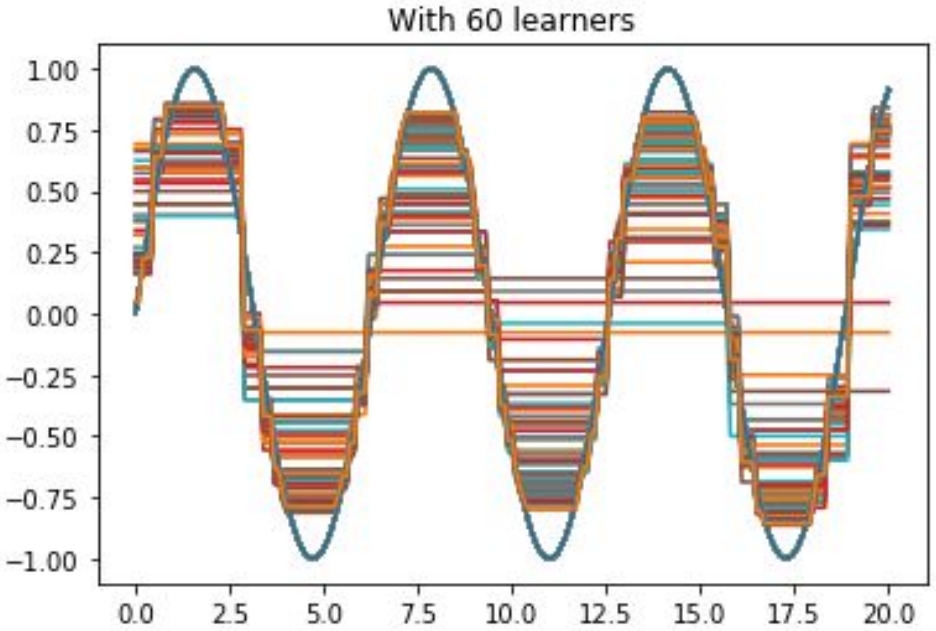
Shrinkage - Slow Convergence

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



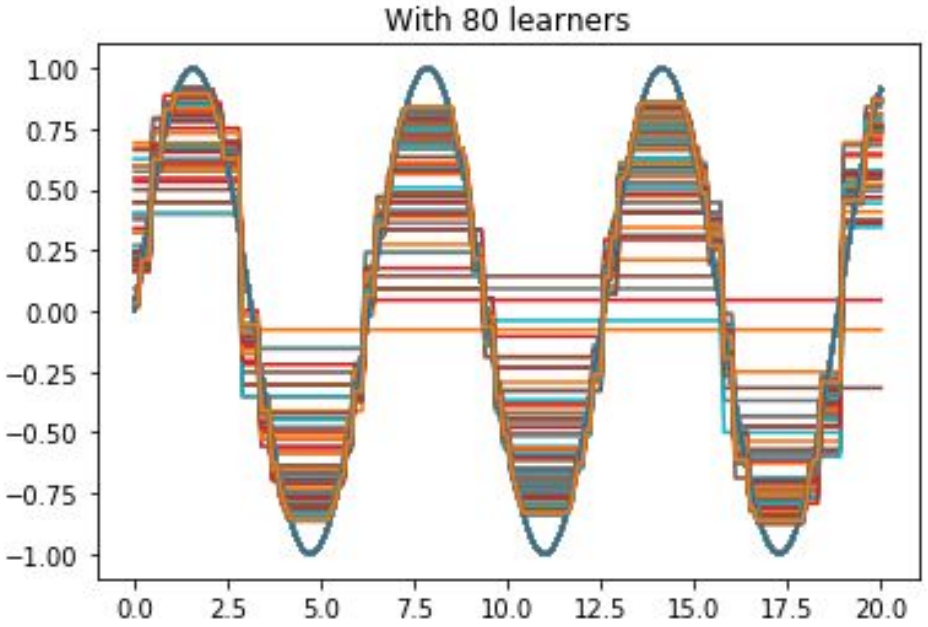
Shrinkage - Slow Convergence

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



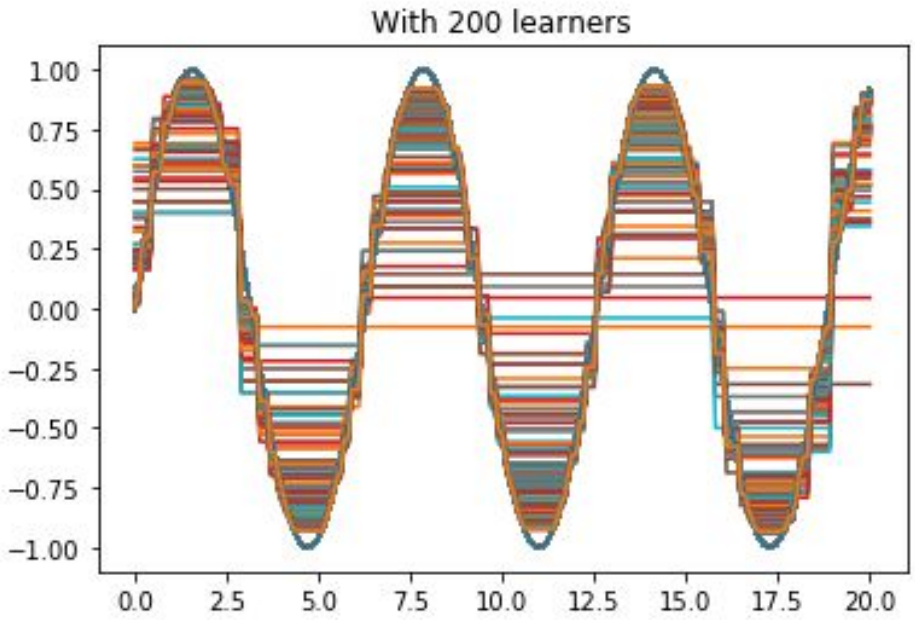
Shrinkage - Slow Convergence

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

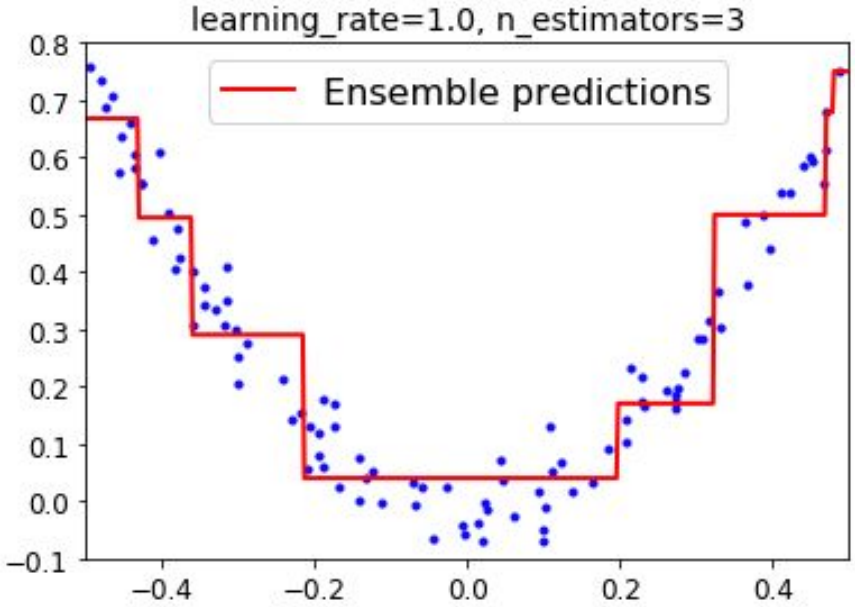


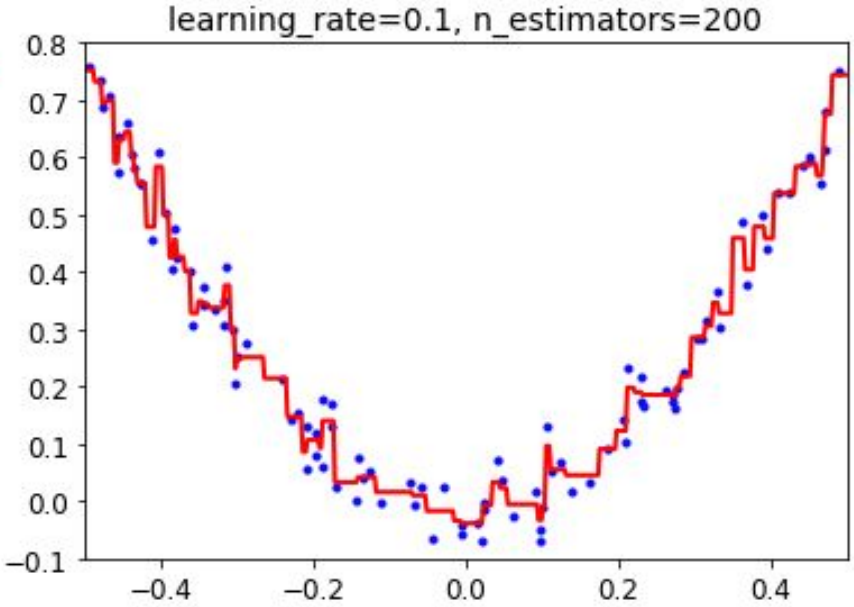
Shrinkage - Slow Convergence

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



Comparing Learning Rates





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

Any Questions?