Gradient Boosting and Weak Learners

(https://github.com/learn-co-curriculum/dsc-gradient-boosting-and-weak-learners) (https://github.com/learn-co-curriculum/dsc-gradient-boosting-and-weak-learners/issues/new/choose)

Introduction

In this lesson, we'll explore one of the most powerful ensemble methods around -- gradient boosting!

Objectives

You will be able to:

- Compare and contrast weak and strong learners and explain the role of weak learners in boosting algorithms
- Describe the process of boosting in Adaboost and Gradient Boosting
- Explain the concept of a learning rate and the role it plays in gradient boosting algorithms

Weak learners and boosting

The first ensemble technique we learned about was ***Bagging***, which refers to training different models independently on different subsets of data by sampling with replacement. The goal of bagging is to create variability in the ensemble of models. The next ensemble technique we'll learn about is ***Boosting***. This technique is at the heart of some very powerful, top-of-class ensemble methods currently used in machine learning, such as ***Adaboost*** and ***Gradient Boosted Trees***.

In order to understand boosting, let's first examine the cornerstone of boosting algorithms -- ***Weak Learners***.

Weak learners

All the models we've learned so far are ***Strong Learners*** -- models with the goal of doing as well as possible on the classification or regression task they are given. The term ***Weak Learner*** refers to simple models that do only slightly better than random chance. Boosting algorithms start with a single weak learner (tree methods are overwhelmingly used here), but technically, any model will do. Boosting works as follows:

1. Train a single weak learner
2. Figure out which examples the weak learner got wrong
3. Build another weak learner that focuses on the areas the first weak learner got wrong
4. Continue this process until a predetermined stopping condition is met, such as until a set number of weak learners have been created, or the model's performance has plateaued

In this way, each new weak learner is specifically tuned to focus on the weak points of the previous weak learner(s). The more often an example is missed, the more likely it is that the next weak learner will be the one that can classify that example correctly. In this way, all the weak learners work together to make up a single strong learner.

Boosting and random forests

Similarities

Boosting algorithms share some similarities with random forests, as well as some notable differences. Like random forests, boosting algorithms are an ensemble of many different models with high inter-group diversity. Boosting algorithms also aggregate the predictions of each constituent model into an overall prediction. Both algorithms also make use of tree models (although this isn't strictly required, in the case of boosting).

Differences

1: Independent vs. iterative

The difference is in the approach to training the trees. Whereas a random forest trains each tree independently and at the same time, boosting trains each tree iteratively. In a random forest model, how well or poorly a given tree does has no effect on any of the other trees since they are all trained at the same time. Boosting, on the other hand, trains trees one at a time, identifies the weak points for those trees, and then purposefully creates the next round of trees in such a way as to specialize in those weak points.

2: Weak vs. strong

Another major difference between random forests and boosting algorithms is the overall size of the trees. In a random forest, each tree is a strong learner -- they would do just fine as a decision tree on their own. In boosting algorithms, trees are artificially limited to a very shallow depth (usually only 1 split), to ensure that each model is only slightly better than random chance. For this reason, boosting algorithms are also highly resilient against noisy data and overfitting. Since the individual weak learners are too simple to overfit, it is very hard to combine them in such a way as to overfit the training data as a whole -- especially when they focus on different things, due to the iterative nature of the algorithm.

3: Aggregate predictions

The final major difference we'll talk about between the two is the way predictions are aggregated. Whereas in a random forest, each tree simply votes for the final result, boosting algorithms usually employ a system of weights to determine how important the input for each tree is. Since we know how well each weak learner performs on the dataset by calculating its performance at each step, we

can see which weak learners do better on hard tasks. Think of it like this -- harder problems deserve more weight. If there are many learners in the overall ensemble that can get the same questions right, then that tree isn't super important -- other trees already provide the same value that it does. This tree will have its overall weight reduced. As more and more trees get a hard problem wrong, the "reward" for a tree getting that hard problem correct goes higher and higher. This "reward" is actually just a higher weight when calculating the overall vote. Intuitively, this makes sense -- trees that can do what few other trees can do are the ones that we should probably listen to more than others, as they are the most likely to get hard examples correct. Since other trees tend to get this wrong, we can expect to see a general split of about 50/50 among the trees that do not "specialize" in the hard problems. Since our "specialized" tree has more weight, its correct vote will carry more weight than the combined votes of the half of the "unspecialized" trees that get it wrong. It is worth noting that the "specialized" trees will often do quite poorly on the examples that are easy to predict. However, since these examples are easier, we can expect a strong majority of the trees in our ensemble to get it right, meaning that the combined, collective weight of their agreement will be enough to overrule the trees with higher weights that get it wrong.

Understanding Adaboost and Gradient boosting

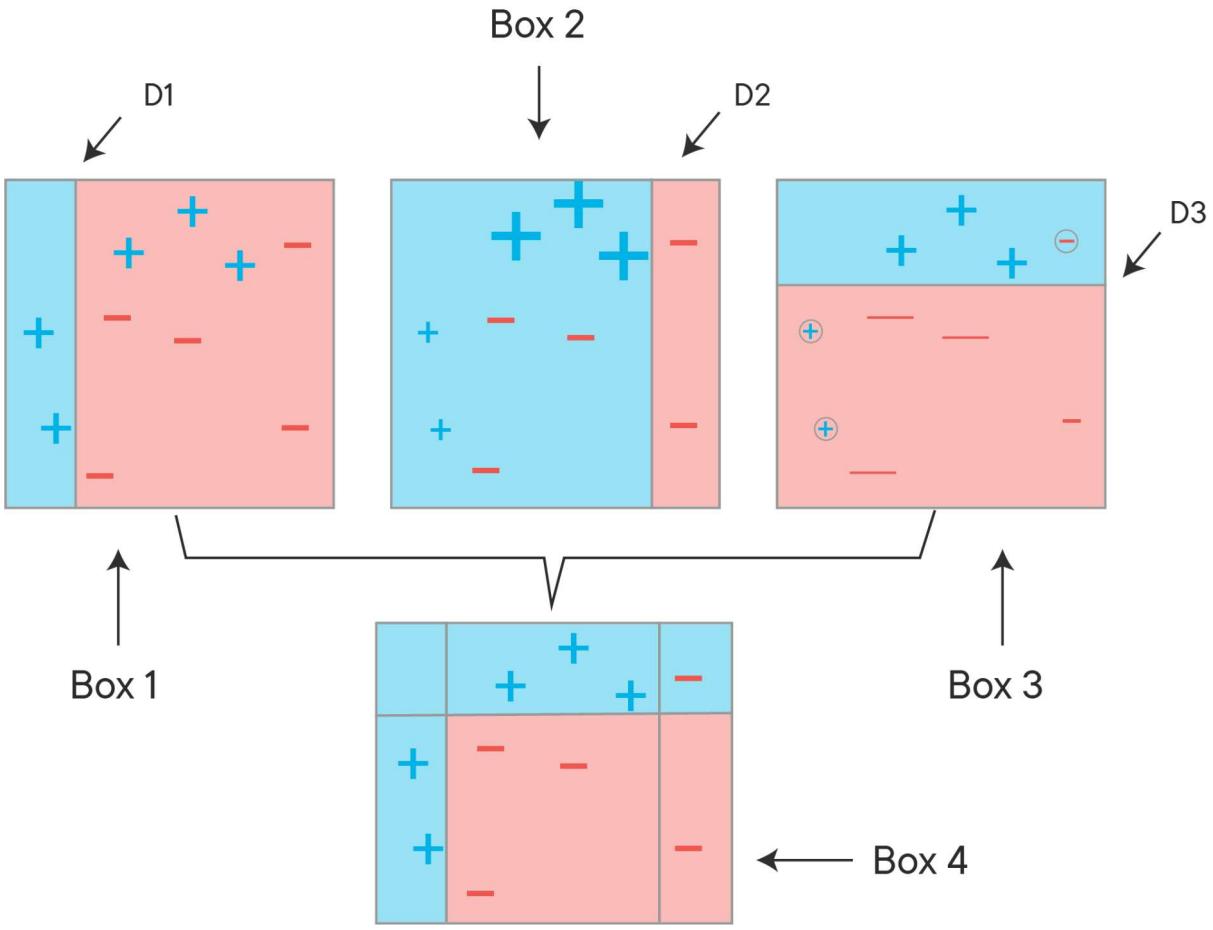
There are two main algorithms that come to mind when Data Scientists talk about boosting: ***Adaboost*** (short for Adaptive Boosting), and ***Gradient Boosted Trees***. Both are generally very effective, but they use different methods to achieve their results.

Adaboost

Adaboost was the first boosting algorithm invented. Although there have been marked improvements made to this algorithm, Adaboost still tends to be quite an effective algorithm! More importantly, it's a good starting place for understanding how boosting algorithms actually work.

In Adaboost, each learner is trained on a subsample of the dataset, much like we saw with ***Bagging***. Initially, the bag is randomly sampled with replacement. However, each data point in the dataset has a weight assigned. As learners correctly classify an example, that example's weight is reduced. Conversely, when learners get an example wrong, the weight for that sample increases. In each iteration, these weights act as the probability that an item will be sampled into the "bag" which will be used to train the next weak learner. As the number of learners grows, you can imagine that the examples that are easy to get correct will become less and less prevalent in the samples used to train each new learner. This is a good thing -- if our ensemble already contains multiple learners that can correctly classify that example, then we don't need more that can do this. Instead, the "bags" of data will contain multiple instances of the hard examples, thereby increasing the likelihood that the learner will create a split that focuses on getting the hard example correct.

The following diagram demonstrates how the weights change for each example as classifiers get them right and wrong.



Pay attention to the colors of the pluses and minuses -- pluses are meant to be in the blue section, and minuses are meant to be in the red. The decision boundary of the tree can be interpreted as the line drawn between the red and blue sections. As you can see above, examples that were misclassified are larger in the next iteration, while examples that were classified correctly are smaller. As we combine the decision boundaries of each new classifier, we end up with a classifier that correctly classifies all of the examples!

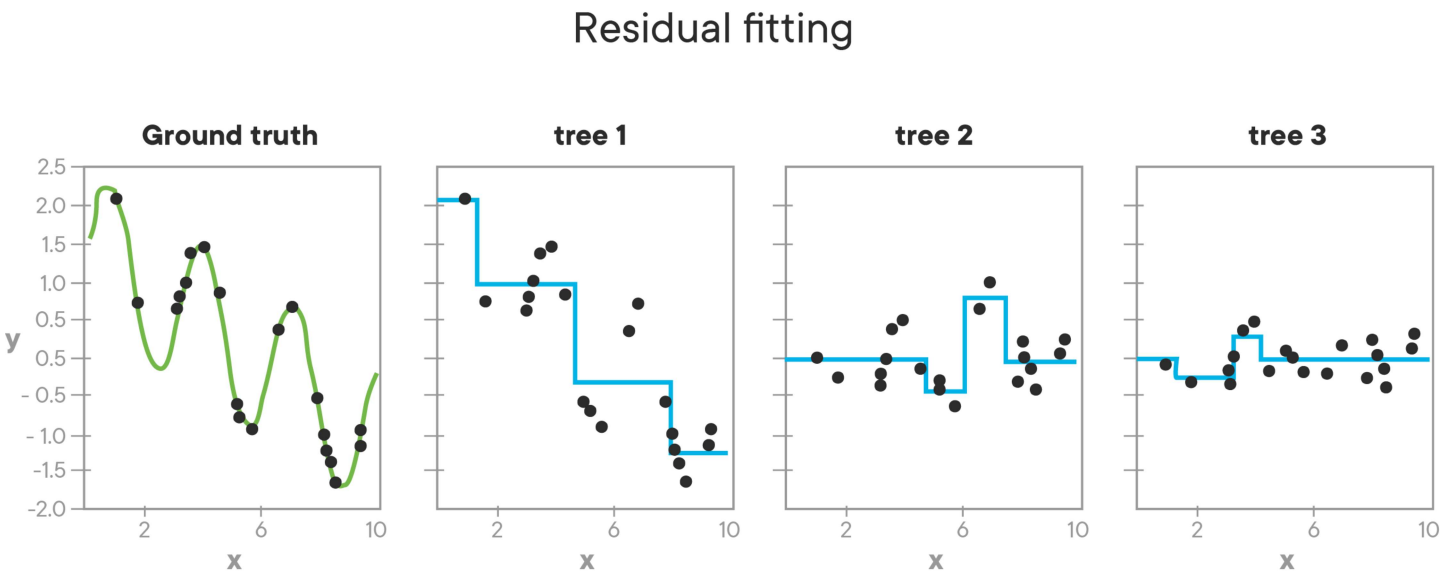
Key Takeaway: Adaboost creates new classifiers by continually influencing the distribution of the data sampled to train each successive learner.

Gradient boosting

Gradient Boosted Trees are a more advanced boosting algorithm that makes use of ***Gradient Descent.*** Much like Adaboost, gradient boosting starts with a weak learner that makes predictions on the dataset. The algorithm then checks this learner's performance, identifying examples that it got right and wrong. However, this is where the gradient boosting algorithm diverges from Adaboost's methodology. The model then calculates the ***Residuals*** for each data point, to determine how far off the mark each prediction was. The model then combines these residuals with a ***Loss Function*** to calculate the overall loss. There are many loss functions that are used -- the thing that matters most

is that the loss function is ***differentiable*** so that we can use calculus to compute the gradient for the loss, given the inputs of the model. We then use the gradients and the loss as predictors to train the next tree against! In this way, we can use ***Gradient Descent*** to minimize the overall loss.

Since the loss is most heavily inflated by examples where the model was wrong, gradient descent will push the algorithm towards creating a new learner that will focus on these harder examples. If the next tree gets these right, then the loss goes down! In this way, gradient descent allows us to continually train and improve on the loss for each model to improve the overall performance of the ensemble as a whole by focusing on the "hard" examples that cause the loss to be high.



Learning rates

Often, we want to artificially limit the "step size" we take in gradient descent. Small, controlled changes in the parameters we're optimizing with gradient descent will mean that the overall process is slower, but the parameters are more likely to converge to their optimal values. The learning rate for your model is a small scalar meant to artificially reduce the step size in gradient descent. Learning rate is a tunable parameter for your model that you can set -- large learning rates get closer to the optimal values more quickly, but have trouble landing exactly at the optimal values because the step size is too big for the small distances it needs to travel when it gets close. Conversely, small learning rates means the model will take a longer time to get to the optimal parameters, but when it does get there, it will be extremely close to the optimal values, thereby providing the best overall performance for the model.

You'll often see learning rates denoted by the symbol, γ -- this is the greek letter, ***gamma***. Don't worry if you're still hazy on the concept of gradient descent -- we'll explore it in much more detail when we start studying deep learning!

The `sklearn` library contains some excellent implementations of Adaboost, as well as several different types of gradient boosting classifiers. These classifiers can be found in the `ensemble` module, which you will make use of in the upcoming lesson.

Summary

In this lesson, we learned about ***Weak Learners***, and how they are used in various ***Gradient Boosting*** algorithms. We also learned about two specific algorithms -- ***AdaBoost*** and ***Gradient Boosted Trees***, and we compared how they are similar and how they are different!

How do you feel about this lesson?

Have specific feedback?

Tell us here! (https://github.com/learn-co-curriculum/dsc-gradient-boosting-and-weak-learners/issues/new/choose)