This note was first developed by Kevin Zhu, Tim Won and Saagar Sanghavi on team SKT for CS189, Project T Fall 2020.

Introduction

In September 2009, The BellKor's Pragmatic Chaos team won a grand total of 1 million USD in the infamous Netflix Challenge. In this challenge, competitors were striving to come up with a better recommendation algorithm than the one currently used by Netflix's own algorithm. Bellkor and his team were able to come up with an algorithm that would beat Netflix's algorithm by a whole 10 percent. How, one would ask? It turned out that a single model was not enough. They used a method called gradient boosted decision trees, which instead aggregated the results of over 500 models. This is one prime example of a technique we will discuss today, known as **ensemble methods**.

In this note, we first discuss intuition behind the ensemble method by introducing the **Multiplicative Weights Update (MWU)** framework in which we can analyze them. Next, we will examine two ensemble methods in machine learning known as **Bagging** and **Boosting**, with an emphasis on boosting. We will introduce boosting with **Adaboost** (adaptive boosting), a commonly used boosting algorithm, and conclude by extending a familiar topic, **Orthogonal Matching Pursuit (OMP)**, to the more generalized notion of **Gradient Boosting**.

Ensemble Methods

Suppose you are tackling a problem with machine learning and are considering a few possible approaches. An intuitive question one may ask is, why must we constrain ourselves to a single model? Why not try multiple approaches and somehow combine their powers? This is truly the heart of an ensemble method. In an **ensemble method**, we form a stronger hypothesis by intelligently combining the results of multiple hypotheses. The idea is rather simple and transcendent. At Berkeley, you have seen this concept in previous courses such as EECS16B. As an example, let us take a look at Lagrange interpolation.

In Lagrange interpolation, given (n+1) points, we strive to recover a unique polynomial of degree <= n that passes through these (n+1) points. The heart of the algorithm is to aggregate the information from the points and intelligently combine them, allowing us to

classify our desired polynomial from the space of degree <= n polynomials. Here, without going into too much detail, we explore a new coordinate system with “basis polynomials,” which we then weigh by their corresponding y coordinates and combine linearly into our resulting polynomial. This is just one example of the ensemble concept, and clearly, there is some merit to the method. But, before we dive straight into the ensemble methods in machine learning, first we shall introduce a more general **multiplicative weights update** framework in which we can examine these methods for a fuller understanding of its application in machine learning.

Multiplicative Weights Algorithm (MWU)

Let us define some terminology first. Historically, computer scientists have referred to the sources that we aggregate as “experts.” The most resounding question is, given our experts, how do we combine their results? Perhaps we can take an average, or a weighted average if we believe that some experts are better than others (e.g the existence of a “best” expert). Perhaps we can take the mode, or most common result. Another question we may ask is, given the results of their predictions, are there ways to update our strategy? The answer to these questions can be explained by the multiplicative weights algorithm.

There are two possibilities for the experts. Either they are all “equally skilled”, or there is some difference between them, implying the existence of one or more best expert(s). If they are all the same, then there are no decisions to be made. We may simply take the majority vote (or an average in a continuous setting). The more interesting case is when the experts are of different skill levels (some experts are correct more often than others), then we must somehow distinguish between them and iteratively make choices, deciding which experts are best at any given moment and accordingly weighing their advice more.

This suggests that we should assign a weight to each one of the experts and update the weights based on incoming data. Intuitively, we should update our weights such that if an expert is correct, we would want to weigh their advice more, and if they are incorrect, we would want to weigh their advice less. Over time, this ensures that we make decisions more heavily based on the advice of the stronger experts and thus maximizing our overall performance. How do we make our choice at each step? In this model, we assume that there is some adversary working against us, and thus it is in our best interest to make decisions probabilistically according to the weights (In other schemas, depending on our constraints, we may opt in for a weighted average or other forms of decision making). Also, let us define “loss” to be the negative result of making an incorrect decision. Specifically detailing the algorithm below,

The Algorithm (1):

The algorithm proceeds as follows. It works with a parameter $0 \leq \epsilon \leq 1/2$ that is hardcoded in the algorithm, and it maintains, at each time step t a set of *weights* $w^{(t)} = (w_1^{(t)}, \ldots, w_n^{(t)})$ associated to the strategies, defined as

$$w_i^{(0)} = 1$$

$$w_i^{(t+1)} = w_i^{(t)} \cdot (1-\epsilon)^{\ell_i^{(t)}}$$

The effect of this definition is that strategies that produced high losses in the past have small weight, and strategies that produces low losses have higher weight. At every step, the algorithm chooses allocations proportionally to the weights

$$x_i^{(t)} = \frac{w_i^{(t)}}{\sum_j w_j^{(t)}}$$

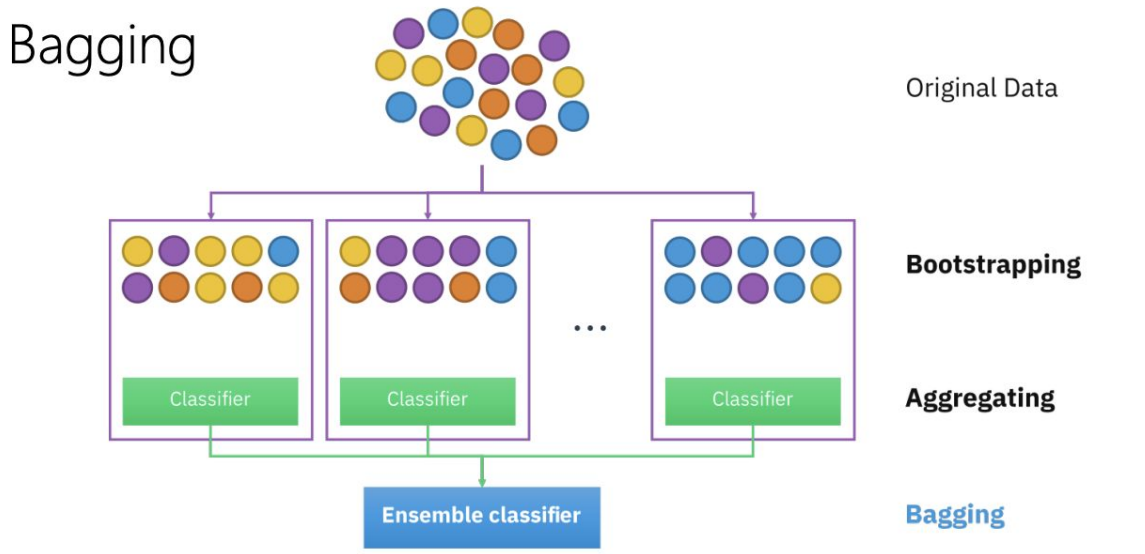
Note that the way we update weights in this algorithm follows the intuition we described. An expert is punished for incorrect choices and rewarded for correct choices, optimally so described by this algorithm although we will not prove it in this note due to the mathematical rigor needed. (FOOTER:) The optimal value of the parameter epsilon may be calculated analytically, $\epsilon = \sqrt{\frac{\ln n}{T}}$, the derivation of which we will not show. Furthermore, note in this case, we are not assuming any prior on the experts' skill levels, and thus they are initialized uniformly to the same weight.

The performance of our algorithm is measured by "regret", defined to be the difference between our loss, and the best possible loss incurred by choosing the best expert at each iteration. Intuitively, it does not make sense to compare to a 0 loss benchmark if a 0 loss benchmark cannot be rationally derived from our experts' advice. Instead, we should compare to the best expert that we could have realistically followed the entire time. Amazingly, with some amount of mathematical analysis, we can prove that the performance of this algorithm minimizes regret and yields a result that is *almost* as good compared to having picked the best expert since the very beginning!

Bagging

Now having seen the multiplicative weights update framework, let us explore what applications of this type are seen in machine learning. Indeed, it will turn out that ensemble methods share some striking similarities with the algorithm above. First, we shall introduce the method of bootstrap aggregation, or **bagging**, although we will not go into depth since this will be the focus of another note.

In bagging, there is first the bootstrap, and then the aggregation. Bootstrapping means: given some training data, we generate m new training sets of size n-prime from that data by uniformly sampling from the training data with replacement. (2) FOOTER: It turns out, if n-prime = n and n is large, then on expectation, (1- 1/e) = 63.2% of samples in a new training set will be unique. Notice since we are doing with replacement, there could potentially be repeats. We then fit a model for each of the m new training sets. From there, we aggregate the results of the m models somehow, for example taking the majority vote for a discrete classification problem or taking an average for a continuous regression problem.



https://upload.wikimedia.org/wikipedia/commons/thumb/c/c8/Ensemble_Bagging.svg/1280px-Ensemble_Bagging.svg.png

This technique is most useful for unstable methods, like decision trees. As you may recall from a previous note, decision trees are very inconsistent depending on the data set or the features. Thus, by using bagging, since we apply the model to different sets/features and take the average, assuming that the results are relatively uncorrelated, this results in lower variance and generally better performance. In fact, you already know what bagging is on decision trees. Those are known as random forests.

Note the similarity between bagging and the multiplicative weights framework. In a sense, we may view bagging as the rather simple formulation of the multiplicative weights update algorithm when all the experts are equally skilled. Here, we do not need to distinguish between any of the experts or make any iterations. Instead, we may just

take the average or weighted majority vote. In boosting, we will see similarities to the case where the experts are not equally skilled.

Boosting

Boosting combines models (especially weak learners) together for better performance
Focus of the boosting:

- Find learners that can correctly classify data point that current model is wrong about
- Weighing models by current performance each time a model is added
- “Harder” points classified with more models

Sources:

https://cs170.org/assets/notes/notes-mw.pdf (1)
https://www.eecs189.org/static/notes/n26.pdf

Footer (1): epsilon optimum
Footer (2): bagging repeats

https://medium.com/analytics-vidhya/decisiontree-classifier-working-on-moons-dataset-using-gridsearchcv-to-find-best-hyperparameters-ede24a06b489