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 instead aggregated the results of over 500 models. This is one prime application 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), which is 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.

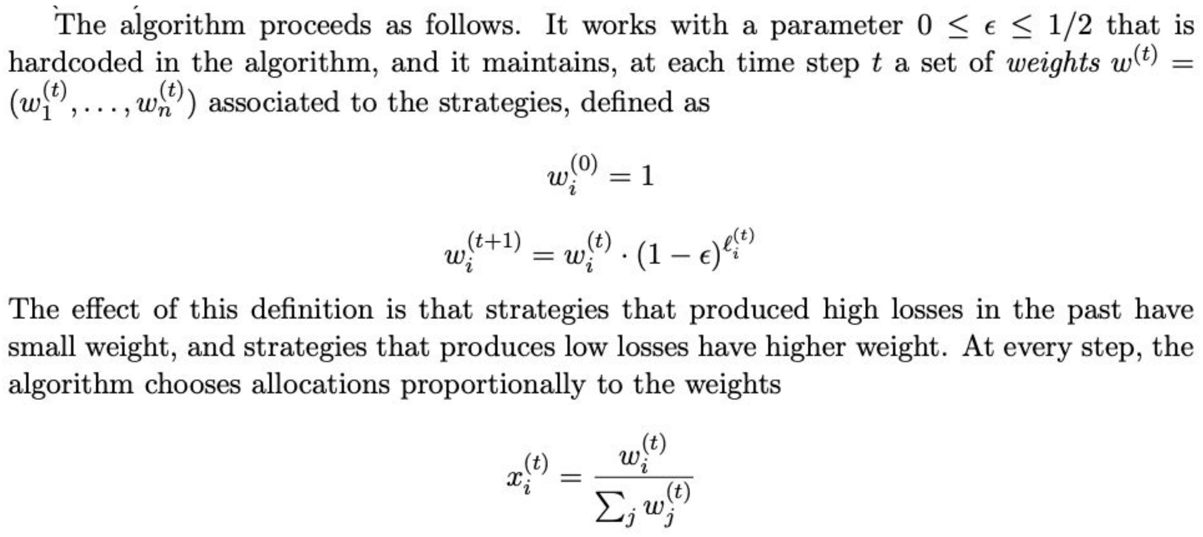
Multiplicative Weights Algorithm (MWU)

The multiplicative weights algorithm and experts framework has numerous real life applications in fields such as finance and game theory. For our purposes, we will strive to learn a basic understanding of it in order to use it as a lens for machine learning ensemble methods. 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. (Think: Why would a deterministic strategy be a poor choice, given the adversary?) (FOOTER 0): in other schemas, depending on our constraints, we may opt in for other forms of decision making, e.g. a weighted average. 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 $0 \le \epsilon \le 1/2$ 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

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

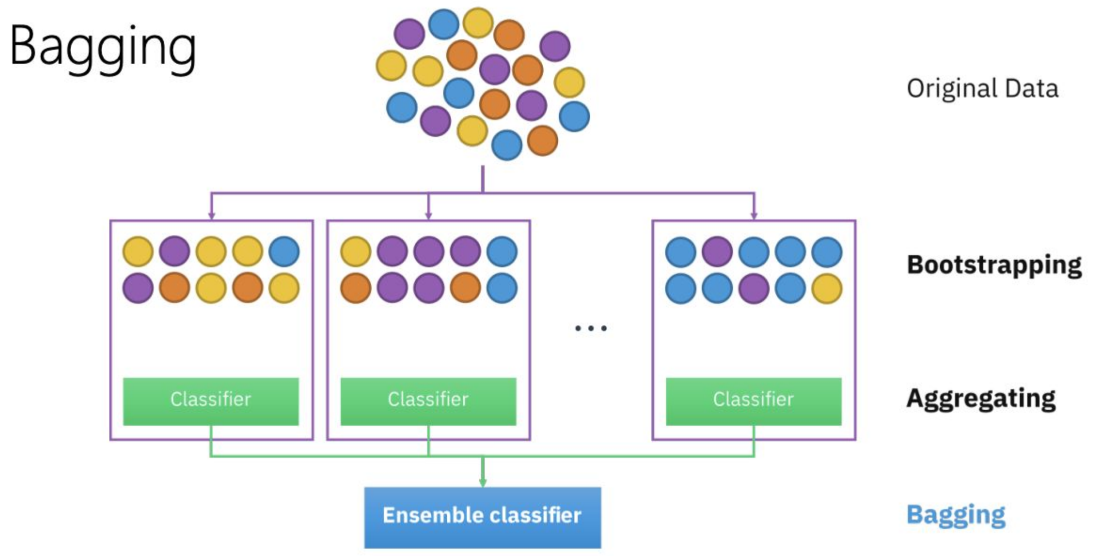
Note that the way we update weights in this algorithm follows the intuition we described. An expert is penalized for incorrect choices and rewarded for correct choices, optimally so described by this algorithm, although we will not prove it in this note due to the mathematical rigor needed. (FOOTER:) 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. Note since we are doing with replacement, there could potentially (very likely) 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

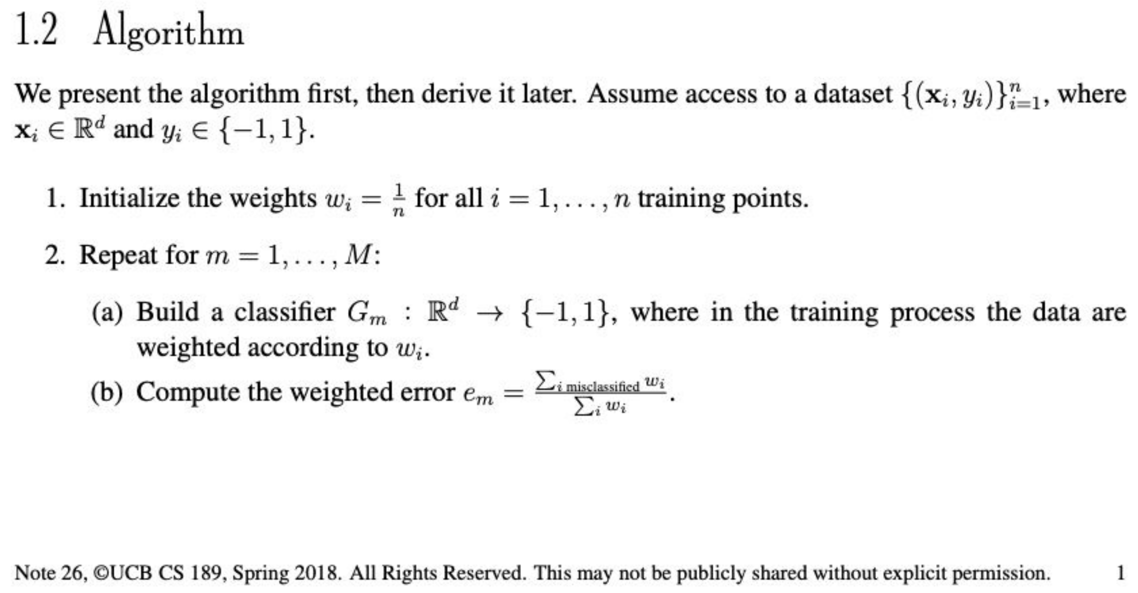
This technique is useful for unstable methods, like decision trees. As you may recall from a previous note, decision trees are very inconsistent depending on the data set or the features. Thus, by using bagging, since we apply the model to different sets/features and take the average, assuming that the results are relatively uncorrelated, this results in lower variance and generally better performance. Sound familiar? In fact, you have already seen an application of bagging, which has this effect. Random forests are just the result of applying bagging to decision trees!

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 iterative updates. Instead, we may just take the average or majority vote!

Boosting

Boosting is another method of combining models together for better performance, however, in a more principled manner than bagging. We will see that boosting can be seen as a cousin of the OMP algorithm and also shares similarities to the multiplicative weight update framework. Let us first examine specifically a boosting algorithm called **AdaBoost**, and use that as a starting point from which we can then generalize.

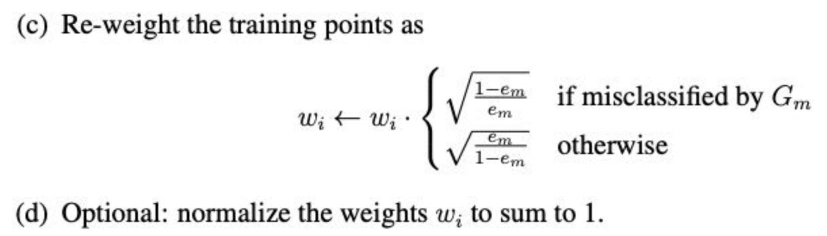
In Adaboost, we create M different models, this time through a more intelligent process of iterative weight updates. The intuition is that some data points will be harder to classify than others. For example, consider a scenario in which you are trying to classify between digits. A poorly written “4” may look a lot like a “9” and thus be much harder to classify, say between a “1” and an “8”. Thus, we should focus more of our energy on these harder points - sounds like we should give each data point a weight. After each iteration, if we misclassified the point, we should increase its weight, and if we classified it correctly, we should decrease its weight, thus effectively shifting our focus towards the harder points. After doing this for each point, we now train a new model from the weighted points. Next, we repeat the weight update process on our new model. Repeat. Specifically detailing the algorithm below (2),



1.2 Algorithm

We present the algorithm first, then derive it later. Assume access to a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$.

1. Initialize the weights $w_i = \frac{1}{n}$ for all $i = 1, \ldots, n$ training points.
2. Repeat for $m = 1, \ldots, M$:
   (a) Build a classifier $G_m : \mathbb{R}^d \to \{-1, 1\}$, where in the training process the data are weighted according to w_i.
   (b) Compute the weighted error $e_m = \frac{\sum_{i \text{ misclassified}} w_i}{\sum_i w_i}$.



   (c) Re-weight the training points as

$$w_i \leftarrow w_i \cdot \begin{cases} \sqrt{\frac{1-e_m}{e_m}} & \text{if misclassified by } G_m \\ \sqrt{\frac{e_m}{1-e_m}} & \text{otherwise} \end{cases}$$

   (d) Optional: normalize the weights w_i to sum to 1.

Let us dive more into the details. The observant reader may have asked - how exactly do we train a model on data points which are weighted differently? Similarly to bagging, we use resampling. However, this time we sample based on a distribution according to the weights (this is why we normalize the weights!) instead of sampling uniformly. Thus, since the higher weight data points have higher probability mass and are more likely to be chosen for the training of the next model. This serves the purpose of the reweighting since now we are focusing our energy on those data points.

Take a step back. Now, we have our M models. How do we aggregate their results? We need a second layer of weights for these models. Although the proof is out of scope, it turns out we can analytically derive an expression for these weights, call them alpha_m from [1… M]):

$$\alpha_m = \frac{1}{2} \ln \left(\frac{1 - e_m}{e_m} \right)$$

Observe that this choice of alpha_m gives higher weight to models with lower error. From these weights, finally we take a weighted majority vote for classification or weighted average for regression.

This technique is useful for low-variance, high-bias predictors (note, the opposite use case of bagging).

Note the similarity between boosting and the multiplicative weights update framework. We can interpret Adaboost as a sort of two-layered experts framework, where we first treat each individual data point as an expert and assign weights for each of those, and then assign each of the M models we generate as experts and assign weights for each of those. First for the inner layer, we update the weights of each data point just like how we do in the multiplicative weights algorithm - penalizing and rewarding respectively for our desired behavior and making our next choices accordingly. Now having derived M models from this weight update process, for the outer layer, we now just need to aggregate the results of the M models as a final iteration.

Finally, you may have noticed that this algorithm is no more than a cousin of **Orthogonal Matching Pursuit (OMP)**. Indeed, the heart of the two algorithms are one in the same. From the nature of the problems, the algorithms use a similar greedy heuristic at each iteration. For OMP, this involves taking the signal with max cross correlation since this the strong beacon signals could potentially mask the weaker beacon signals. For boosting this involves emphasizing the data points our current model misclassifies (and vice versa) since these points are more likely to be intrinsically harder data points to classify. Both, at each new iteration, start working from the based on the residual left from the previous iteration. At the end, both solve some optimization problem in order to yield the final coefficients. Besides the beautiful symmetry behind the matching pursuit of the two algorithms, these algorithms have great applications across machine learning today.

Gradient Boosting

In the previous section when we explored Adaboost, we limited ourselves to a certain loss function. Although we do not show the mathematical details, this loss function is the exponential loss function. By allowing ourselves a degree of freedom in choosing a (differentiable) loss function, we now create a more generalized set of boosting algorithms known as **Gradient Boosting** algorithms.

Conclusion

In this note, we explored in great detail the intuition behind ensemble methods, first from the point of view of the experts framework. We examined the multiplicative weights update algorithm and some of its analogous counterparts in machine learning - bagging and boosting. We took a deeper investigation on boosting - exploring Adaboost and gradient boosting, and discovering that its heart lies with the same intuition that lies in the heart of the OMP. Recall the Netflix challenge discussed in the beginning of the note. It turns out that the ensemble method that the winning team used was in fact gradient boosting on decision trees. This powerful set of techniques is now another tool in your machine learning toolbox, and we invite you to play with them in the notebook.

Sources:

https://cs170.org/assets/notes/notes-mw.pdf (1)
https://www.eecs189.org/static/notes/n26.pdf (2)

Footer (0): MWU model
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