Experiments on Boosting with Noise

Dana Glasner and Sara Stolbach
dg2342@columbia.edu,ss3067@columbia.edu

December 12, 2006

Abstract

In this paper we discuss experiments and our results that we have obtained on Boosting with Noise. A boosting algorithm is one that takes a weak PAC learning algorithm and "boosts" it to achieve high accuracy. We examine the Kalai and Servedio paper [1] as our focal point for boosting. We will analyze in depth the algorithm mentioned in [1] known as MMM - Modified MM, which is based on the noise-free MM algorithm [2]. We then show our results on implementation and experimentation of the MMM algorithm.

1 Introduction

Often we are given a algorithm that is "weak", meaning it does not achieve arbitrarily high accuracy on the concept at hand. This is where boosting comes in to the picture. Boosting algorithms take a weak learner and create a strong learner. There are many boosting algorithms available. However, one of the major difficulties is learning with noise which the majority of boosting algorithms do not address.

The standard PAC model does not take the presence of noise into account. However, when dealing with real data, the presence of noise is a given. Therefore, the PAC model has been extended to various different noise models. The simplest noise model is the Random Classification Noise model which we will examine in depth and is defined by having an oracle $EX(c, D, \eta)$ which returns an example (x, b) drawn from D such that $b = c(x)$ with probability $1 - \eta$ and $b = 1 - c(x)$ with probability η. Noise-tolerant boosters provide a black-box method of transforming a noise-tolerant weak learner into a noise-tolerant strong learner. Given a noise tolerant booster, one can convert a learning algorithm with advantage slightly better than $1/2$ into a learning algorithm with arbitrarily high accuracy. Servedio and Kalai [1] have shown that it is "hard" to boost a weak learning algorithm past the noise rate even in the Random Classification Noise model. This means that if one-way functions exist, true noise-tolerant boosters cannot exist. However, [1] do give a method of boosting any noise-tolerant weak learner arbitrarily close to the noise rate. This algorithm is called *modified* MM or MMM and is based on Mansour and McAllester's boosting algorithm (MM) [2] which works by creating a branching program where each node is a hypothesis h and an example x is directed left when $h(x) = 0$ and right when $h(x) = 1$. [1] also define an "okay"-learner, which is a slightly more powerful learner than the weak learner that can be boosted to arbitrarily high accuracy even in the presence of Random Classification Noise.

2 Definitions

Definition 1 *Weak Learner [4] Algorithm A is a weak PAC learner for concept class C with advantage τ if for any $c \in C$, for all D, for all $0 < \delta < 1$, with probability $\geq 1 - \delta$, A outputs h such that $Pr_D[h(x) \neq c(x)] \leq \frac{1}{2} - \tau$.*

Definition 2 *Boosting: [4] A Boosting Algorithm runs a weak learner A*

multiple times using a sequence of carefully chosen different distributions $EX(c, D_1), EX(c, D_2), ldots, EX(c, D_T)$ *returning* $h_1, h_2, ldots, h_T$. *It will then combine the hypotheses* $h_1, h_2, ldots, h_T$ *to construct a final hypothesis* h *with accuracy* ϵ.

3 Intuition for why MM/MMM work

The key reason the MM/MMM boosters work is the simulation of the balanced distribution at the leaves when instantiating the weak learner. Creating the balanced distribution forces "progress" since now the weak learner cannot just return the trivial hypothesis of 0 or 1. This is because if it did so, its error rate would be 1/2 with respect to the balanced distribution and it would not satisfy the definition of a weak learner. The 0 or 1 hypothesis is the output of the hypothesis of the parent leaf l on the examples that reach leaf l_0 and l_1 respectively. Therefore, if the new hypotheses (at leaves l_0 and l_1) would generate the trivial hypotheses, there would be no progress with respect to the parent hypothesis. However, since the distribution is balanced, the weak learner is forced to classify some of the examples classified as 0 by the parent hypothesis as 1 or some of the examples classified as 1 by the parent hypothesis as 0. This ensures that at each stage the hypothesis becomes more refined, and the error drops. Therefore, under the original distribution, the proportion of the examples that end up in the leaf corresponding to the left child of l_0 and have classification 0 will be higher than the proportion of the examples that end up in leaf l_0 and have classification 0.

4 MM Implementation

In order to gear the algorithm for our final goal of Boosting with Noise (MMM), we have modified the algorithm to work by "boosting-by-filtering" where we do not use a fixed sample. The MM algorithm constructs a branching program with each internal node v having an hypothesis h_v generated by the weak learner. Any instance $x \in X$ gives a unique directed path from the root to a leaf (external node) by following the hypothesis h_v of each internal node v [1].

Let $f \in C$ be a fixed target function and D be a fixed distribution over X where $D|_l$ denotes D conditioned on $x \in L$. Therefore, in the set L

of leaves l, each leaf l has probabilities $a_l = \Pr_D[x \in L \text{ and } f(x) = 1]$ a nd $b_l = \Pr_D[x \in L \text{ and } f(x) = 0]$. $p_l = \Pr_D[f(x) = 1 | x \in l]$. u_l is the uncertainty in leaf l and $u_l = 2\sqrt{p_l(1-p_l)}$ w_l is the weight of leaf l and $w_l = \Pr_D[x \in l]$. $w_l u_l$ gives the weighted uncertainty of leaf l with respect to L. $w_l u_l$ can be rewritten in terms of a_l and b_l as $2\sqrt{a_l b_l}$.

The MM algorithm will split leaves to increase the accuracy of our classifier, and merge leaves to ensure that the branching program does not get too large. (Kearns and Mansour show that without merges the resulting tree could be exponentially large [5]). Merging leaves will decrease the accuracy of the classifier. We will only perform merges if the cumulative uncertainty increase is substantially less the the uncertainty decrease of the last split. Therefore, we will make progress and the final output of the MM algorithm will be the branching program.

MM Algorithm:
Input:

- desired final error level ϵ $(0 < \epsilon < 1)$
- access to γ-weak learner A
- access to noise-free example oracle $\mathrm{EX}(f, D)$
- delta δ for weak learner A
- instance space X we are trying to boost
- oracle simulator to balance the distribution and create $\mathrm{EX}(f, \hat{D}|_l)$

Algorithm:

1. Start with trivial partition, $L = \{X\}$, so the branching program is a single leaf. Generate $1/\epsilon^2$ examples to create a_l and b_l for the root node.
2. **Get the leaf l_{max} that will reduce the overall uncertainty the most** by constructing candidate splits and finding leaf with max $\{w_l u_l - w_{l_0} u_{l_0} - w_{1_1} u_{l_1}\}$.
3. **perform the split** on l_{max}. Let Δ_S be the reduction for $l_{max} = \{w_l u_l - w_{l_0} u_{l_0} - w_{l_1} u_{l_1}\}$.

4. Stop if the error of the current branching program $\leq \eta + \tau$ where error $= \Sigma_l \min(a_l, b_l)$

5. Set $\Delta_M := 0$.

6. **Get the two leaves $l_a \neq l_b$ that if merged will cause the minimum increase incertainty** by finding the hypothetically merged leaf l_{min} with min $\{w_l u_l - w_{l_a} u_{l_b} - w_{1_b} u_{l_b}\}$.

7. Let z = $\{w_l u_l - w_{l_a} u_{l_b} - w_{1_b} u_{l_b}\}$ of l_{min}.

8. **Merge if safe:** If $\Delta_M + z < \Delta_S/2$ then

   - Merge leaves l_a and l_b to create l_{min} in the branching program.
   - Set $\Delta_M := \Delta_M + z$.
   - Go to step 6.

9. Otherwise, go to step 2.

In step 2, we only examine the leaves where $w_l u_l > \epsilon/(2L)$ since the total contribution of those leaves would be at most $\frac{\epsilon}{2}$.

When we construct candidate splits in step 2, we store the hypothesis h, a_l and b_l that we get from running the weak learner A. This saves time on future searches for constructing splits. In fact, at most we have to run the weak learner t wice - on the two new leaves from the most recent split.

Additionally, in step 6 when we merge 2 leaves, we do not have to sample the distribution to calculate the new a_l, b_l for that leaf. Given a_{l_a}, b_{l_a} and a_{l_b}, b_{l_b}, we can find a_l and b_l since: $a_l = a_{l_a} + a_{l_b}$ and $b_l = b_{l_a} + b_{l_b}$. This saves time since drawing enough examples that reach leaf l to achieve high accuracy can take many draws from EX, especially when l has low weight.

We also input a smaller delta δ_{weak} to the weak learner based on the δ passed into the booster since we must make sure that the failure rate of the overall MM algorithm is bounded by δ.

By a union bound, the overall $\delta < \delta_{weak} \cdot N$, where N is the number of times the weak learner is instantiated. We cannot know ahead of time exactly how many times the weak learner will be called, but we can bound N from above.

By [1] Theorem 4, the maximum number of splits for MM is $maxsplits = \frac{144}{\gamma^4} \log \frac{2}{\epsilon\gamma^2} \log \frac{1}{2\epsilon}$.

Since we only run the weak learner once on each leaf, the maximum number of times the weak learner is instantiated is $maxsplits + L$ where L is the maximum number of leaves at any time.

Every time the weak learner is instantiated there is a δ_{weak} chance that it will fail. Therefore, the probability that one of the $maxsplits + L$ instantiations will fail is $\delta_{weak} \cdot (maxsplits + L)$. We want this to be $\leq \delta$. Therefore, we set $\delta_{weak} = \delta/(maxsplits + L)$.

5 MMM Implementation

Given the previous values defined in MM algorithm, we can modify them for use in the MMM algorithm. The noisy probablity can be defined as $\tilde{p_l} = \Pr[label = 1 | x \in l]$. Since $\tilde{p_l} = p_l(1-\eta) + (1-p_l)\eta$, then $p_l = \frac{\tilde{p_l}-\eta}{1-2\eta}$. MMM can therefore estimate p_l within the additive error of c by estimating $\tilde{p_l}$ within an additive $\frac{c}{1-2\eta}$ [1]. We also assume that we know the noise rate η. If we did not know it, we could run the algorithm by going through possible values for η, by starting with a small value and gradually increasing, and the algorithm would succeed when we had used the proper η

MMM Algorithm:
Input:

- access to γ-weak learner A
- accuracy value $\tau > 0$
- noise rate $\eta, 0 < \eta < \frac{1}{2}$
- access to example oracle $\mathrm{EX}(f, D, \eta)$
- delta δ for weak learner A
- instance space X we are trying to boost
- oracle simulator to balance the distribution and create $\mathrm{EX}(f, \hat{D}|_l, \eta)$

Algorithm:

1. Start with trivial partition, $L = \{X\}$, so the branching program is a single leaf. Generate $2 \cdot \frac{\log(40/\delta)}{\tau^2(1-2\eta)^2}$ examples to create a_l and b_l for the root node.

2. **Get the leaf l_{max} that will reduce the overall uncertainty the most** by constructing candidate splits and findi ng leaf with max $\{w_l u_l - w_{l_0} u_{l_0} - w_{1_1} u_{l_1}\}$. We achieve this by running the weak learner using the balanced oracle $\mathrm{EX}(f, \hat{D}|_l, \eta)$.

3. **perform the split** on l_{max}. Let Δ_S be the redution for $l_{max} = \{w_l u_l - w_{l_0} u_{l_0} - w_{l_1} u_{l_1}\}$.

4. Stop if the error of the current branching program $\leq \eta + \tau/2$ where error $= \Sigma_l \min(a_l, b_l)$

5. Set $\Delta_M := 0$.

6. **Get the two leaves $l_a \neq l_b$ that if merged will cause the minimum increase incertainty** by finding the hypothe tically merged leaf l_{min} with min $\{w_l u_l - w_{l_a} u_{l_b} - w_{1_b} u_{l_b}\}$.

7. Let z $= \{w_l u_l - w_{l_a} u_{l_b} - w_{1_b} u_{l_b}\}$.

8. **Merge if safe:** If $\Delta_M + z < \Delta_S/2$ then

   - Merge leaves l_a and l_b to create l_{min} in the branching program.
   - Set $\Delta_M := \Delta_M + z$.
   - Go to step 6.

9. Otherwise, go to step 2.

We use the same improvements mentioned in MM in MMM.

In addition, during both the splitting and merging process we need not examine leaves where $\min\{p_l, 1 - p_l\} \geq \eta + \tau/2$. This ensures that we can efficiently simulate the noisy balanced oracle. If the leaf does not satisfy this condition it may not be possible to do so [1].

Similarly to calculating δ_{weak} for MM, by [1] Theorem 6, the maximum number of splits for MMM is: $maxsplits = O(\frac{1}{\gamma^4} \log \frac{1}{\tau\gamma} \log \frac{1}{\tau})$. Clearly, the number of leaves at a given time must be less than $2 \cdot maxsplits$ so the

total number of times the weak learner is instantiated is $O(\frac{1}{\gamma^4}\log\frac{1}{\tau\gamma}\log\frac{1}{\tau})$. Therefore, we set $\delta_{weak} = \delta/maxsplits$.

Again, when we merge 2 leaves, we do not have to sample the distribution to calculate the new a_l, b_l, p_l, w_l for that leaf. Given $a_{l_a}, b_{l_a}, w_{l_b}$ and $a_{l_b}, b_{l_b}, w_{l_b}$, we can find a_l, b_l, w_l, p_l since: $a_l = a_{l_a} + a_{l_b}, b_l = b_{l_a} + b_{l_b}, w_l = w_{l_a} + w_{l_b}, p_l = a_l/w_l$.

6 Intuition of why it is hard to boost past the noise rate

When boosting to the noise rate, if we encounter a leaf with a $p_l < \eta$ fraction of positive (or negative) examples then we do not consider this leaf anymore. However, if we are trying to boost past the noise rate, we would still have to consider this leaf and be able to create a balanced distribution in order to get any useful information from the weak learner. Otherwise the weak learner could just output the trivial hypothesis of 0 (or 1), which would give no additional information relative to the parent hypothesis. However, [1] show that although creating a balanced distribution is necessary to ensure progress, it is hard to create such a distribution when $p_l < \eta$. This is because when $p_l < \eta$, a positively labeled random example is actually a true negative example with probability $> 1/2$. This is because an example is labeled positive with probability: $(\eta)(1 - p_l) + (1 - \eta)(p_l)$. Therefore, the probability that an example is a true positive given that it is labeled positive is: $\frac{(1-\eta)(p_l)}{(\eta)(1-p_l)+(1-\eta)(p_l)}$. This is less than $1/2$ whenever $(1-\eta)(p_l) < (\eta)(1-p_l)$, which is true whenever $p_l < \eta$. So even if we rejected ALL the negatively labeled examples and kept all of the positively labeled examples, we still would not have a balanced distribution.

Additionally, this would mean that if we tried to follow the balancing method given for MMM above to get $Pr_D[f(x) = 0 \wedge \text{ not rejected}] = Pr_D[f(x) = 1 \wedge \text{ not rejected}]$ we would get $(1 - p_l)((1 - \eta)(1 - p_r) + \eta) = p(\eta(1 - p_r) + 1 - \eta)$. Solving for p_r (the probability of rejecting a negatively labeled example), we get: $p_r = \frac{1-2p_l}{1-p_l-\eta}$ which is > 1 when $p_l < \eta$.

If you tried randomly rejecting some of the positively labeled examples, the proportion of true positives and true negatives would stay the same. So it seems that in the case where $p_l < \eta$ the only way to effectively balance the distribution is by knowing which examples that are labeled positive are true

positives and which are true negatives. But the booster cannot know this since its only knowledge about the function comes from the classifications of the instantiated weak learners.

When [1] prove that it is hard to boost past the noise rate, they use a p-biased pseudorandom function family as the concept class the booster is learning since this family is hard to learn. This way, they ensure that the booster cannot learn the function on its own, without using the weak-learner. p-biased pseudorandom function families exist based on the assumption that one-way functions exist. Therefore, [1] show that if one-way functions exist then black-box noise tolerant boosters do not exist.

7 Weak Learner Implementation

To test our boosting implementation, we used a strong learning algorithm for the class of conjunctions over $\{0,1\}^n$ with RCN that we discussed in class. We used this learner to test our implementation of both the MM algorithm (by setting $\eta = 0$) and also to test our implementation of the MMM algorithm.

Initially, we planned to use the strong learning algorithm to generate a "weak" hypothesis by running the strong learning algorithm with a constant ϵ close to .5. However, when running the strong learning algorithm, we noticed that even when we set epsilon to be very close to .5, the hypothesis outputte by the learner was the correct hypothesis with very high probability. In terms of our booster implementation, this would mean that after generating only the hypothesis for the root node the booster would already reach the desired accuracy and we would not be able to meaningfully test our implementation. Therefore, we randomized the hypothesis generated by the strong learner to make sure that it was not better than some chosen accuracy *acc* so that we could measure the effects of the booster.

Algorithm for learning conjunctions of size n with Random Classification Noise [3]:

- $p_i = Pr[c(x) = 1 | x_i = 0] \cdot Pr[x_i = 0]$
- $q_i = Pr[c(x) = 1 | x_i = 0]$
- $\diamond_i = Pr[label = 1 | x_i = 0]$

1. begin with a new hypothesis containing all n variables.

2. for $i = 1$ to n

   - approximate $Pr[x_i = 0]$ to accuracy $\epsilon/4n$ with probability δ/n by drawing $\frac{8\log(n/\delta)\cdot\eta^2}{\epsilon^2}$ examples.
   - approximate $\diamond_i$ to accuracy $\epsilon/4n$ with probability δ/n by drawing $\frac{8\log(n/\delta)\cdot\eta^2}{\epsilon^2}$ examples.
   - $q_i = \frac{\diamond_i - \eta}{1-2\eta}$
   - $p_i = q_i \cdot Pr[x_i = 0]$
   - if $p_i > \epsilon/2n$, remove i from the hypothesis.

To convert the strong hypothesis h into a weaker hypothesis h_{weak} with maximum accuracy acc, when evaluating $h_{weak}(x)$, return 0 or 1 randomly with probability $r = 2 \cdot (1 - acc)$. With probability $1 - r$ return $h(x)$.

8 Oracle Implementation

We created an example oracle $EX(c, D)$ that takes a distribution and a concept and draws examples randomly according to the inputted distribution. Since we were dealing with a finite domain, the distribution is just an array that maps each element in the domain to an integer. We can use this mapping to find the weights of each element by viewing $weight_x = map_x/totalSum$ where $totalSum$ is the sum of all the entries in the array. We also have a feature for generating an arbitrary distribution and for choosing a random concept from the concept class.

To draw an example from the distribution, we first assigned the elements x_i in the domain consecutive disjoint ranges of values, where the size of the range assigned to element x_i is map_{x_i}. Then we choose a random number uniformly from 0 to $totalSum - 1$ and check to see which element's range the random number fell in. We then return the corresponding element as the randomly drawn element.

Our example oracle can also be used for a RCN example oracle $EX(c, D, \eta)$. This is simply done by keeping track of the noise rate η and with probability η, flipping the label of the example before returning it from a draw. It is run with $\eta = 0$ to simulate a non-noisy oracle.

We also created an example oracle that can make adversarial draws specific to our booster implementation. In adversarial noise, with probability η,

the adversary can decide whether or not to flip the label of an example. This means that the probability that a label is flipped is at most η, but may be less. In our implementation, the oracle takes the current branching program as input and puts noise only examples x such that $h(x) = 0$, where h is the hypothesis of the root node. This means that there is noise rate η on the left branch of the root node and noise rate 0 on the right branch. This is unlike the case of RCN where we were able to assume that the noise rate at each leaf of the branching program was η. The oracle could also be modified to put noise only on any other selection of target nodes in the branching program. We chose this adversarial strategy because in the MMM boosting algorithm, in order to create the balanced distribution and to decide which leaves to split or merge effectively, it is necessary to know the noise rate at each leaf. Therefore, a conceivable adversarial strategy against the MMM algorithm would be to make it hard to effectively balance the distribution at each leaf by varying the proportion of corrupted examples that reach different leaves.

We simulated the oracle as $EX(c, \hat{D}|_l, \eta)$ by taking the current branching program and current leaf as input and creating a balanced distribution ($Pr_D[f(x) = 1] = Pr_D[f(x) = 0]$) over the examples that end up in that leaf. The way the balanced distribution is created is different when there is no noise and when there is RCN.

The oracle $EX(f, \hat{D}_l)$ is simulated by randomly flipping a coin before drawing from $EX(f, D)$. If the coin is heads, wait until a positive example is drawn that reaches l. Otherwise, wait for a negative example to be drawn that reaches l [1].

Algorithm for simulating $EX(f, \hat{D}_l, \eta')$ given access to $EX(f, D, \eta)$ [1]:
Given an example that reaches l:

1. If $p_l \leq 1/2$:
   - **Labeled 0:** Reject with probability $p_r = \frac{1-2p_l}{1-p_l-\eta}$, keep with probability $1 - p_r = \frac{p_l-\eta}{1-p_l-\eta}$.
   - **Labeled 1:** Flip its label with probability $p_f = \frac{(1-2p_l)\eta(1-\eta)}{(1-p_l-\eta)(p_l+\eta-2p_l\eta)}$, don't flip with probability $1 - p_f$.
2. If $p_l > 1/2$ then $p_l \leftarrow 1 - p_l$

   (flip the roles of 0 and 1 in the above algorithm)

- **Labeled 1:** Reject with probability $p_r = \frac{1-2p_l}{1-p_l-\eta}$, keep with probability $1 - p_r = \frac{p_l-\eta}{1-p_l-\eta}$.
- **Labeled 0:** Flip its label with probability $p_f = \frac{(1-2p_l)\eta(1-\eta)}{(1-p_l-\eta)(p_l+\eta-2p_l\eta)}$, don't flip with probability $1 - p_f$.

The new noise rate $\eta' = \frac{1}{2} - \frac{p_l-\eta}{2(p_l+\eta-2p_l\eta)}$.

9 Experiments

Computer and OS used to run experiments on: IBM Thinkpad, T42 Kubuntu Linux and Red Hat Linux 2.6.9-11.EL on Columbia clic machines.

9.1 MM

ϵ	time (in seconds)	no. splits	no. merges	errors (testing)
0.07	19.834	8.859	6.010	679.523
0.09	6.704	4.472	2.442	840.588
0.1	3.950	3.558	1.724	922.256
0.15	0.565	1.595	0.280	1172.380
0.2	0.130	1.175	0.085	1864.565
0.3	0.040	1.020	0.000	2049.515
0.5	0.045	1.000	0.000	2565.560

Table 1

We tested the MM algorithm on $\epsilon = .07 - .5$. It was tested on 200 random monotone conjunctions where the size of the instance space, $n = 5$. We ignore conjunctions which learn the concept right away (no splits and merges) since they don't teach us anything and overpower the statistics of the other cases. We chose $n = 5$ since the concept is randomly chosen and as n gets larger the probability of the concept being easier to learn increases. This allows less merges and splits to occur on most cases whereas the cases where many splits will occur will take more time and can be too much for a standard desktop to handle.

The averages for each ϵ-value are listed in Table 1. As expected, the larger the error margin, the faster it will run and the more errors the branching program will make. It appears that when $n = 5$, as expected, as ϵ decreases, average time increases error decreases. It is also interesting to note (See

Figure 1) that as the value of ϵ decreases, the difference between splits and merges increases. Although it is not shown in the tables, as is expected, the smaller the conjunction, the longer it takes to learn; i.e. x_1 takes longer to learn than $x_1 \wedge x_2$. This is in part due to the fact that it is more difficult for the oracle to find examples that belong solely to x_1 than to $x_1 \wedge x_2$.

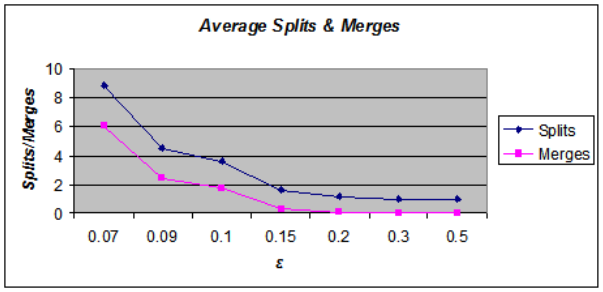

Figure 1: Splits and Merges

9.2 MMM

η	τ	time (in seconds)	no. splits	no. merges	errors (testing)
0.1	0.01	15.075	6.305	2.46	1028.49
0.1	0.03	2.17	2.885	0.715	1192.62
0.1	0.05	0.467	1.997	0.277	1350.616
0.2	0.01	3.18	2.84	0.47	1847.075
0.2	0.03	0.31	1.815	0.105	2034.14
0.2	0.05	0.13	1.42	0	2085.435

Table 2

We ran the MMM algorithm 200 times with $n = 5$, and ignore cases without splits and merges (as in MM). each on reasonable τ and η values. Based on the averages in Table 2 it is clear that the higher the value of τ, the more errors that occur when testing the branching program. In addition, the more noise there is the faster the program will run, but on the other hand there will be significantly more errors when testing the branching program.

9.3 MMM without introducing extra noise

η	τ	time (in seconds)	no. splits	no. merges	errors (testing)
0.1	0.01	15.37475	6.4569	2.516	1027.694389
0.1	0.03	1.73	3.005	0.75	1195.71
0.1	0.05	0.595	2.12	0.365	1309.13
0.2	0.01	3.85	2.76	0.465	1876.885
0.2	0.03	0.36	1.865	0.125	2034.91
0.2	0.05	0.085	1.065	0	2398.405

Table 3

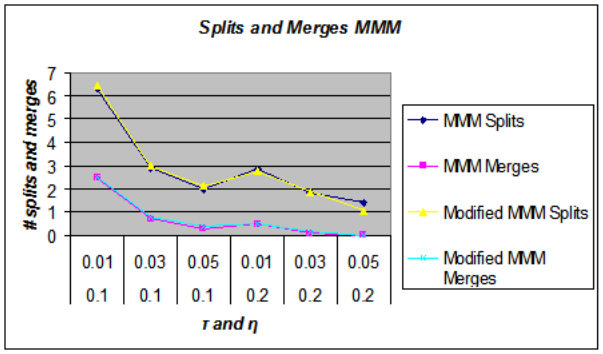

Figure 2: Splits and Merges

When the oracle draws examples in the MMM algorithm, it introduces extra noise in order to keep the noise rate on positive and negative examples roughly equal. It seems that introducing extra noise would be counterintuitive to what the MMM algorithm tries to do. We ran the algorithm without introducing extra noise (As before: 200 times, $n = 5$) . The averages are listed in Table 3. The averages are virtually the same as with the extra noise. Figure 2 plots the average splits and merges with and without the extra noise. The only reason we would need to have the noise rate be the same on positive and negative examples is in our conjunctions learner when we calculate $q_i = \frac{\diamond_i - \eta}{1-2\eta}$, we assume here that η is uniform over both positive and negative examples. However, it seems that the slight variance in noise rate in positive and negative examples is not enough to throw off our weak learner and cause it to remove/not remove so many variables from the hypothesis that its accuracy falls below 1/2. However, this experiment could

be repeated again with weak learners that are more sensitive to variations in the noise rate.

9.4 The Merging Factor

We noticed that although it may seem like the booster would run faster without merges, the merges can actually help the algorithm run faster. This is because if the branching program is kept smaller, there are less leaves to attempt to split (which entails running the weak learner) and also because more splits cause the weight of the leaves to get smaller faster and it becomes harder to simulate the balanced distribution at that leaf. The line in the MM/MMM algorithm that determines how many merges occur is: **Merge if safe:** If $\Delta_M + z < \Delta_S/2$.

An interesting evaluation is to see how the algorithm performs with different ratios besides for 1/2. We replaced the line above with **Merge if safe:** If $\Delta_M + z < \Delta_S/k$, with varying k. If k is made larger, less merges should occur and the branching program will become larger. If k is smaller, but still greater than 1, then more merges will occur and overall progress may be slower.

We ran the MM algorithm with k varying from 1.1 to 5 in increments of .3: 20 times for each value of k and calculated average run-time, splits, and merges for each k. Our results show that when k is very small, the booster takes longer to run (see Figure 3 and Figure 4). However, run times seem to vary in the middle.

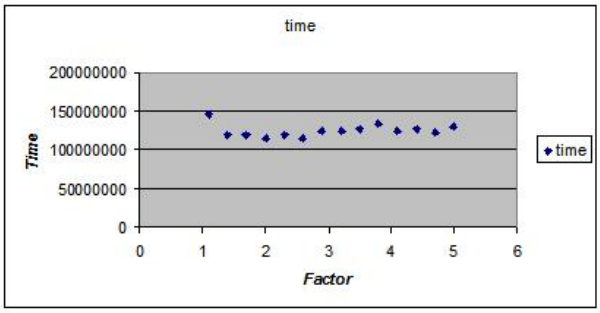

Figure 3: Time

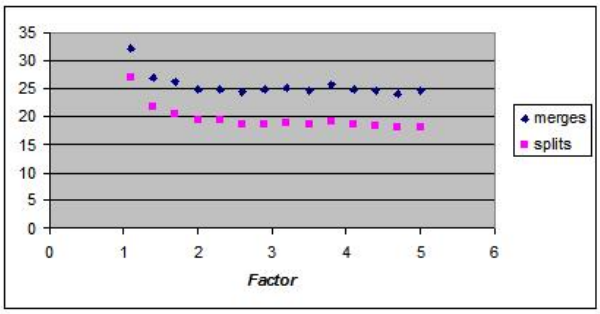

Figure 4: Merges and Splits

9.5 Boosting with Okay Learner

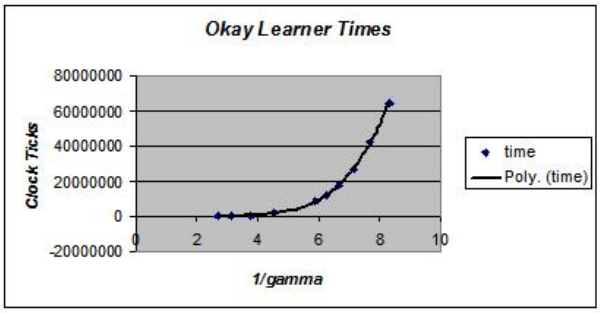

Figure 5: Time

When boosting an "okay" learner, a balanced distribution does not have to be created. This is because progress is guaranteed even when a majority of the examples are positive or negative. The trivial hypothesis of 0 or 1 is not valid for the "okay" learner since it must correctly classify a fraction of both the positive and negative examples. Since a balanced distribution does not have to be created, the proportion of corrupted examples at each node does not have to be known. Therefore, we tried boosting an adversarial noise-tolerant "okay" learner to the noise rate using the method given for "okay" learners under RCN.

However, the proportion of corrupted examples at each leaf must be known to be able to calculate p_l for each leaf and determine the best splits and merges. Since we were only boosting to the noise rate, we only calculated $\hat{p}_l$ and used it as an estimate for p_l. Then we based our decision of which

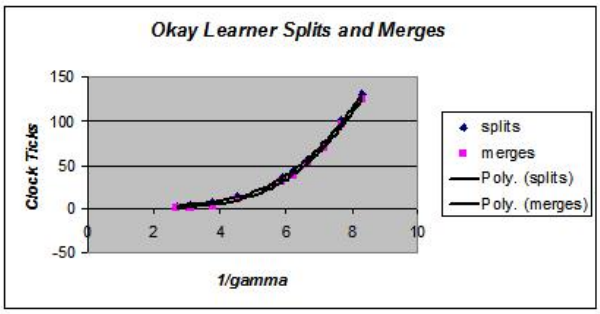

Figure 6: Splits and Merges

leaves to split and merge based on the empirical value calculated for $\hat{p}_l$. Our assumption was that if $\hat{p}_l$ is high then the true p_l is probably also high.

To create an adversarial noise-tolerant "okay"-learner, we cheated. The hypothesis actually "knows" the concept that is to be learned. So on an input x, the hypothesis can just evaluate the concept on x. If x evaluates to 1, the hypothesis returns 1 with probability acc_1 and if x evaluates to 0 the hypothesis returns 0 with probability acc_2. Otherwise, it returns the incorrect value.

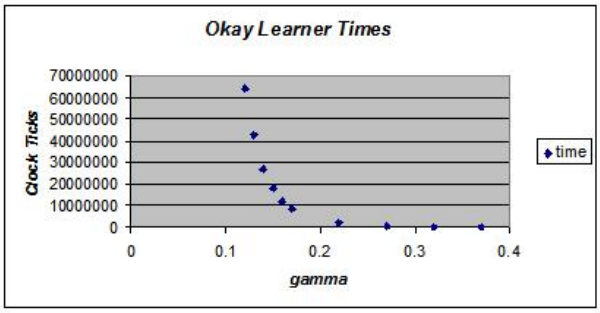

Figure 7: Time

The following is the algorithm for boosting an "okay"-learner (with our modification):

Modify the MM algorithm in the following ways:

- Estimate p_l using $\hat{p}_l$
- In Step 2, run the adversarial noise-tolerant γ-okay learner using the unbalanced conditional distribution $EX(f, D|_l, \eta)$.

We ran the boosting algorithm on this "okay"-learner with γ ranging from .12 to .37 using the adversarial-noise oracle described above and running the boosting algorithm 20 times for each value of γ. We plotted $1/\gamma$ vs. Time and we were able to fit the curve with $(1/\gamma)^4$. This shows that even when estimating the best splits and merges using $\hat{p}_l$, the run-time of the algorithm is still polynomial in $1/\gamma$. See Figure 5, Figure 6, Figure 7 and Figure 8.

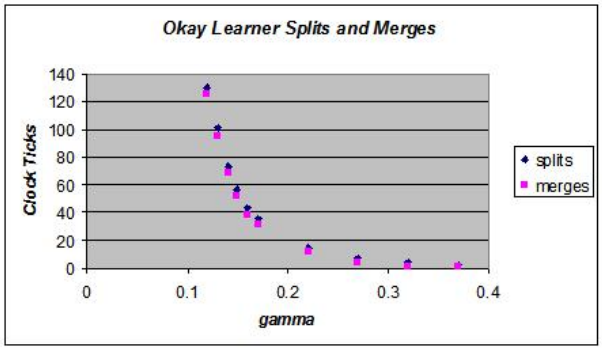

Figure 8: Splits and Merges

10 Future Work

The MMM algorithm has been setup so that it need not change regardless of the Instance Space, Concept Class, Weak Learner, etc...that we are trying to boost. We would like to perform tests on various concept classes other than monotone conjunctions. It would be very interesting to see how the MMM algorithm would fair with real world examples as well as larger instance spaces and more data. One of the main problems with this is run-time. Larger instance spaces and larger data tend to take too long for a standard desktop. If we did not have the time-constraint we could test many more of these scenario's easily.

We would also like to explore various weak learners as opposed to our strong learner disguised as a weak learner. In addition, we would like to see how the MMM algorithm fairs on various noise models such as adversarial and nasty noise [6]. We would also like to see how MMM compares to other boosters.

11 Conclusion

We have analyzed the MM, MMM, and various experiments on them. It is quite evident that the MMM algorithm works well as a noise booster. The difficulty arises in the details and deciding what values to use for the various inputs such as the accuracy, and the noise rate. We were not able to test our experiments on as much data as we would have liked due to time constraints and processor speed. In particular, the simulated oracle took an extreme amount of time to find an example that reached the current leaf and such examples had to be chosen for both running the weak learner and calculating a_l and b_l for each leaf. However, from our experiments we can determine some general trends.

The MMM does fairly well when the correct values are chosen and is a good choice to use to boost learning concepts where the data available involves noise.

12 References

1. A. Kalai, R. Servedio. Boosting in the Presence of Noise. *In Proceedings of the 35th Annual Symposium on the Theory of Computing* 195-205, 2003.

2. Y. Mansour and D. McAllester. Boosting using branching programs. *Journal of Computer and System Sciences*, 64(1):103-112, 2002.

3. R. Servedio, Monotonic Conjunctions. *Columbia University, Class Lecture*, Fall 2006.

4. R. Servedio, Boosting. *Columbia University, Class Lecture*, Fall 2006.

5. N. H. Bshouty, N. Eiron, and E. Kushilevitz, *PAC learning with nast noise*, Theoretical Computer Science 288 (2002), no. 2, 255-275.