Combinatorial Fusion with On-line Learning Algorithms

Chris Mesterharm
Department of Computer and Information Sciences
Fordham University
New York, NY, U.S.A.
Email: mesterharm@cis.fordham.edu

D. Frank Hsu
Department of Computer and Information Sciences
Fordham University
New York, NY, U.S.A.
Email: hsu@cis.fordham.edu

***Abstract*—We give a range of techniques to effectively apply on-line learning algorithms, such as Perceptron and Winnow, to both on-line and batch fusion problems. Our first technique is a new way to combine the predictions of multiple hypotheses. These hypotheses are selected from the many hypotheses that are generated in the course of on-line learning. Our second technique is to save old instances and use them for extra updates on the current hypothesis. These extra updates can decrease the number of mistakes made on new instances. Both techniques keep the algorithms efficient and allow the algorithms to learn in the presence of large amounts of noise.**

Keywords: On-line Learning, Voting, Perceptron, Winnow.

I. Introduction

In this paper, we consider a batch based fusion technique for machine learning. We use X to represent the set of all possible instances and Y to represent the set of labels. The labels belong to a finite set Y where $|Y| = k$. We assume we have n classifiers and that each classifier makes predictions on $x \in X$ by giving a vote for each of the possible labels. For simple classifiers, this corresponds to giving a vote of 1 to the predicted label and a vote of 0 to all other labels. For more sophisticated classifiers, we can assign a number to each label. The higher the number, the more confidence the algorithm has in that particular label. These predictions create a k by n matrix A_x.

The goal of our fusion model is to learn a linear combination of these classifier predictions that maximizes the accuracy on future instances. The linear combination is represented by $\mathbf{w}$, a vector of n weights. The prediction of the fusion model is $P = A_x\mathbf{w}$. Each of the values in vector P corresponds to a label $y \in Y$. The fusion algorithm predicts the label with the largest value in P.

This is a common model for fusion. The difficulty is to learn a close to optimal value for $\mathbf{w}$ without making significant assumptions on how the individual classifiers behave. Some of the classifiers could be partially redundant and some of the classifiers could be irrelevant. In addition, it is highly likely that no linear combination of classifiers correctly predicts all future instances.

Fortunately, there is a model of learning that gives strong guarantees for learning these types of combinations. On-line learning assumes learning takes place in a sequence of trials. A trial is decomposed into three steps. First, the learner receives an instance. Next, the learning algorithm predicts the label of the instance. Last, the algorithm receives the true label of the instance and uses this information to refine its future predictions. The goal of the algorithm is to minimize the number of mistakes. In particular, efficient on-line learning algorithms exist that give good upper-bounds on the number of mistakes when learning linear-threshold functions [1], [2].

These on-line linear-threshold algorithms have strong theoretical guarantees. The upper-bounds on mistakes are distribution independent allowing an adversary to generate the instances. In fact, the adversary can generate any instance, even instances that do not correspond to the target linear-threshold function. The upper-bound on mistakes only depends on the complexity of the target function and a loss function that is related to whether the instances are correctly classified with respect to the target function.

In this paper, we use on-line linear-threshold algorithms for fusion. The linear-threshold algorithms we use are Perceptron [1], Winnow [3] and ALMA [4]. We convert these algorithms to our previously defined fusion problem using a technique described in [5]. This technique preserves the bounds on mistakes for these algorithms in this multi-class fusion setting.

We improve the performance of these on-line algorithms for problems where instances are sampled from a distribution using two new techniques. The first technique uses multiple hypotheses to replace the prediction strategy of an on-line algorithm. Instead of predicting with a single hypothesis, a fusion of the saved hypotheses are used for predictions. The second technique saves recent instances. The new algorithm is identical to the basic algorithm except it periodically recycles over the old instances as if they were new trials. These techniques exploit the mistake bound guarantees of these on-line algorithms to heuristically improve their performance when instances are generated by a fixed distribution.

II. On-line Mistake-bounds

In this section, we review typical upper-bounds on mistakes for linear-threshold algorithms. Our primary example is a version of the Winnow algorithm called Balanced Winnow [3]. We give pseudo-code for this algorithm in Figure 1. This algorithm, along with all the algorithms we consider in this paper, is extremely efficient. When using a sparse

Balanced Winnow(α)

Parameters

$\alpha > 1$ is the update multiplier.

Initialization

$t \leftarrow 1$ is the current trial.

$\forall i \in \{1, \ldots, n\}$ $w^+_{i,1} = 1$ are the positive weights.

$\forall i \in \{1, \ldots, n\}$ $w^-_{i,1} = 1$ are the negative weights.

Trials

Instance: $\mathbf{x}_t \in [0, 1]^n$.

Prediction: If $\mathbf{w_t^+} \cdot \mathbf{x_t} \geq \mathbf{w_t^-} \cdot \mathbf{x_t}$

predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.

Update: Let $y_t \in \{-1, 1\}$ be the correct label.

If $y_t(\mathbf{w_t^+} \cdot \mathbf{x_t} - \mathbf{w_t^-} \cdot \mathbf{x_t}) \leq 0$

$\forall i \in \{1, \ldots, n\}$ $w^+_{i,t+1} = \alpha^{y_t x_{i,t}} w^+_{i,t}$

and $w^-_{i,t+1} = \alpha^{-y_t x_{i,t}} w^-_{i,t}$.

Else

$\forall i \in \{1, \ldots, n\}$ $w^+_{i,t+1} = w^+_{i,t}$

and $w^-_{i,t+1} = w^-_{i,t}$.

$t \leftarrow t + 1$.

Figure 1. Pseudo-code for the Balanced Winnow algorithm.

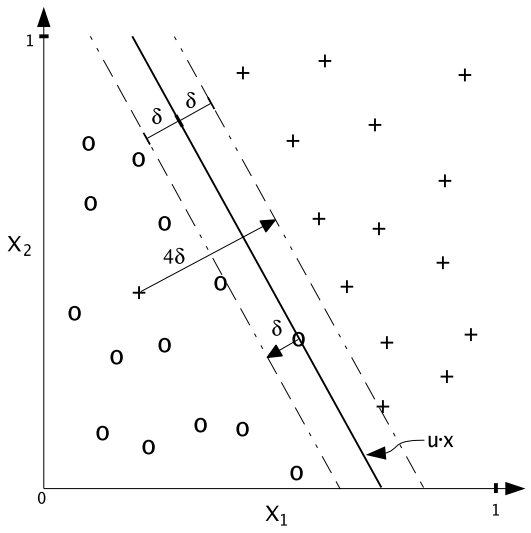

Figure 2. Linear-threshold target function.

representation for instances, each trial takes $O(m_t)$ time where $O(m_t)$ is the number of none zero attributes in the instance during trial t.

The Balanced Winnow algorithm works by keeping track of weights w_i^+ and w_i^- for each attribute. Both of these weights are non-negative, but if w_i^+ is much larger than w_i^- then an nonzero attribute x_i contributes to a prediction of 1. If w_i^- is much larger than w_i^+ then a nonzero attribute x_i contributes to a prediction of $y = -1$. The weights are updated with a multiplication parameter α and are only changed if the algorithm makes a mistake. This is referred to as a mistake-driven algorithm. On-line learning algorithms that perform well against adversaries are commonly mistake-driven.

To help understand the theoretical results, in Figure 2 we give an example instance space and target function for a learning problem that has two continuous attributes and a binary label. The hyperplane that separates the instances is described with a vector of weights $\mathbf{u}$. These target weights are normalized so that $\sum_{i=1}^{n} |u_i| = 1$. Notice that the Balanced Winnow algorithm can learn negative weights through the use of $\mathbf{w}^-$ and can learn a threshold that shifts the hyperplane by adding an attribute that is always 1. This is similar to the threshold technique used by the Perceptron algorithm [1].

Notice that there is a margin of size δ around the $\mathbf{u}$ hyperplane. Any instance inside the margin or on the wrong side of the hyperplane is considered noisy. The amount of noise is the perpendicular distance from the instance to the correct side of the margin. The total amount of noise, N, is just the sum of the noise from all the noisy instances. Therefore $N = \sum_{t=1}^{T} \max(0, \delta - y_t(\mathbf{u} \cdot \mathbf{x_t}))$ where T is the final trial. This is often called the hinge loss [6]. For our purposes, it is useful to consider $\hat{N} = N/\delta$. This is the amount of noise normalized with respect to the size of the margin. In Figure 2, we have two noisy instances where $\hat{N} = 5$.

When the parameter α is set to $1 + \delta$ then the number of mistakes made by Balanced Winnow is at most

$$3.2 \frac{\ln(2n) + \sum_{i=1}^{n} |u_i| \ln |u_i|}{\delta^2} + 1.35\hat{N}. \quad (1)$$

This is a variation of a bound found in [7]. For most problems, the important variables are the number of attributes, n, the size of the margin, δ, and the amount of noise, $\hat{N}$.

For example, consider learning a disjunction with five relevant variables. The best target function for Balanced Winnow is to set u_i to $2/11$ for the five relevant attributes and to set an always 1 threshold attribute weight to $1/11$. The remaining target weights are set to 0. This gives a hyperplane that correctly classifies this disjunction with a margin of $1/11$. The maximum number of mistakes for this problem is at most $388 \ln(2n) + 1.35\hat{N}$.

For someone with experience primarily in batch learning, it might be difficult to interpret the quality of this mistake bound. However, there is an intuitive way to view these adversarial mistake bounds with respect to generalization accuracy in the batch setting. Each time an on-line algorithm gets a new instance it is similar to a test example in the batch setting. After T trials the algorithm has seen T new test examples. Therefore, roughly speaking, the average accuracy of the hypotheses used during the T trials is close to M/T where M is the number of mistakes made by the algorithm. For our example problem, this gives an upper-bound on the average hypothesis accuracy of $(388 \ln(2n) + 1.35\hat{N})/T$. When there are enough noise free instances, with high probability there are accurate hypotheses being generated by the Balanced Winnow algorithm [8].

Because this mistake bound is for adversaries, it applies to any sequence of instances. The only thing that changes the mistake bound from sequence to sequence is the amount of noise with respect to the target function. Considering a fixed $\hat{N}$ value, any correlation between the attributes is allowed and any type of noise is allowed as long as $\hat{N}$ is not exceeded.

Sub-expert Balanced Winnow(α)

Parameters

$\alpha > 1$ is the update multiplier.

Initialization

$t \leftarrow 1$ is the current trial.

$\forall i \in \{1, \ldots, n\}\ w_{i,1} = 1$.

Trials

Instance: Sub-experts $(\mathbf{x}_{1,t}, \ldots, \mathbf{x}_{n,t})$ where $\mathbf{x}_{i,t} \in [0,1]^k$

Prediction: Let $\zeta_{\mathbf{w}}(j) = \sum_{i=1}^{n} w_{i,t} x_{i,t}^{j}$

Predict a class $\hat{y}_t$ such that for all $j \neq \hat{y}_t$

$\zeta_{\mathbf{w}}(\hat{y}_t) \geq \zeta_{\mathbf{w}}(j)$.

Update: Let y_t be the correct label. If $\zeta_{\mathbf{w}}(\hat{y}_t) \geq \zeta_{\mathbf{w}}(y_t)$

$\forall i \in \{1, \ldots, n\}\ \ w_{i,t+1} = \alpha^{x_{i,t}^{y_t} - x_{i,t}^{\hat{y}_t}} w_{i,t}$.

$t \leftarrow t + 1$.

Figure 3. Pseudo-code for Sub-expert Balanced Winnow.

This gives a guarantee on performance that does not depend on simplifying assumptions such as fixed label noise. Any instances and any type of noise is allowed.

Unfortunately, a single noisy instance can cause a large increase in $\hat{N}$. In the example disjunction problem, a noisy instance where all five relevant attributes are 1 but the label is 0 causes an increase of 10 in $\hat{N}$. (This is the maximum possible increase for this concept.) This causes the mistake bound to grow by $1.35(10) = 13.5$ mistakes. Therefore, according to this analysis, a single mislabeled instance could cause 14 additional mistakes.

A large increase in mistakes due to noise could make these algorithms perform poorly in practice. Assume, we are trying to learn a concept that has a minimum error rate of 10%. If these noisy instances cause the error rate to grow by a factor of 13.5 then the bound becomes meaningless. Fortunately, when the instances are sampled from a distribution, the performance of these on-line linear-threshold algorithms improves [9], [10]. In fact, the motivation of this work is to take the solid theoretical and practical results of on-line algorithms and to improve their performance when instances are generated by a distribution that contains noisy instances.

III. Sub-expert Conversion

In this section, we review a transformation that coverts a linear-threshold algorithm into an algorithm that can combine prediction vectors. Each prediction vector is of size k where k is the number of labels. This technique was originally described in [5], where the prediction vectors were called sub-experts. More details can be found in that paper.

In Figure 3, we give a version of Balanced Winnow that combines sub-experts. This version of the algorithm is still extremely efficient. Assuming the sub-experts are stored in a sparse format, one trial of Sub-expert Balanced Winnow takes $O(m_t)$ where m_t is the number of nonzero values in all the sub-experts.

The Sub-expert Balance Winnow algorithm is derived from Balance Winnow by a simple transformation. The key to the transformation is to generate an instance vector that can be used by the original linear-threshold algorithm. This instance vector is equal to $\mathbf{z} = \mathbf{x}_t^y - \mathbf{x}_t^{\hat{y}}$ where $\mathbf{x}_t^y$ are the predictions of all the sub-experts on the correct label and $\mathbf{x}_t^{\hat{y}}$ are the predictions of all the sub-experts on the predicted label.

When both algorithms us the same weights, instance $\mathbf{z}$ is predicted correctly by the linear-threshold algorithm if and only if the sub-expert version makes the correct prediction. This forces the two algorithms to make an identical number of mistakes. The sub-expert version effectively has the same mistake bound on the sub-expert problem as the linear-threshold algorithm has on the $\mathbf{z}$ based problem.

Because the original and sub-expert versions of an algorithm are so similar, it is possible to use techniques designed for a linear-threshold algorithm and apply them with little modification to the sub-expert algorithm. Therefore, in the rest of this paper, we give techniques without explicitly specifying whether the algorithm is linear-threshold or sub-expert. However, our later experimental work is focused on fusion with sub-experts.

In this paper, we do not restrict ourselves to just Balanced Winnow; we also use the Perceptron [1] and ALMA [4] algorithms. The linear-threshold to sub-expert transformation for both ALMA and Perceptron is identical and retains the mistake bound and algorithmic efficiency of the original linear-threshold algorithms. All trials can be performed in $O(m_t)$ [7].

IV. On-line to Batch Conversion

To get good results using on-line algorithms in a batch setting, one must often modify the algorithm. An on-line algorithm that is designed to minimize the number of mistakes against an adversary is not optimized for problems where instances are sampled from a distribution.

Many adversarial algorithms have unstable accuracy in their hypotheses when instances are generated by a distribution. Instead of a smooth increase in accuracy over the course of the learning trials, these algorithms have an accuracy that can jump over a range of values. While this accuracy, on average, tends to increase, just selecting the final hypothesis from a sequence of trials can result in poor performance. This final hypothesis might not have high accuracy.

A typical example of this instability can be seen in Figure 4. The figure contains a graph that shows the accuracy during a typical run of Balanced Winnow. The learning problem is to predict whether or not an area of land has forest cover that is of type spruce [11]. The accuracy is measured at every trial by sampling with a test set of 10,000 test instances. As can be seen, the accuracy is unstable.

One cause of this instability is noisy instances. A noisy instance perturbs the current hypothesis of the algorithm, and extra mistakes are required to correct these changes. As explained in Section II, a large number of mistakes might be necessary to fix the hypothesis. See [7] for more details on the issue of instability with mistake-driven algorithms.

There are many techniques to improve the performance of on-line algorithms when instances are generated by a distribution. Some of these techniques have focused on reducing the number of mistakes in the on-line setting [9], [12], [13]. Others

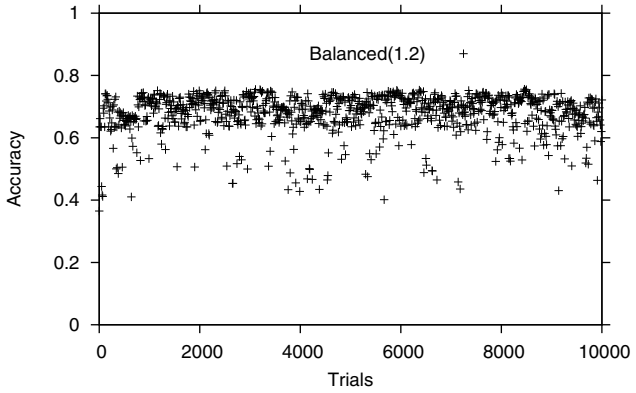

Figure 4. Accuracy of Balanced with $\alpha = 1.2$ on forest cover problem.

techniques have focused on generating a single hypothesis for the batch setting [8], [14].

We focus on reducing the number of mistakes in the on-line setting; however, our techniques can also be applied to the batch setting. Our techniques attempt to remove the instability of the algorithm hypotheses and increasing their accuracy. This means that the final hypotheses generally has a high accuracy. For the batch setting, the training data is used for the trials and this final hypothesis is returned as the classifier. Therefore, our techniques are effective for both types of problems.

Another advantage of working in the on-line setting is that it makes it easier to select the best options for an algorithm. In the batch setting, one must always be concerned when comparing the performance of different options. Typically these comparisons require validation data, and it is problematic to use the validation data for other aspects of the algorithm. Because the on-line model combines training and testing, it is easier to select the best option for an algorithm.

For example, the Balanced Winnow algorithm has a parameter α, and the algorithm's performance can vary greatly based on the setting of this parameter. Instead of running several Balanced algorithms on training data and then using validation data to select the best performer, in the on-line model one can run all the algorithms and select the version of Balanced Winnow that makes the fewest mistakes at the end of the trials. The various versions get to learn with all the data, and they are tested during the trials.

While the previous explanation is a heuristic argument, the Weight Majority Algorithm (WMA) is a more theoretically justified way of selecting the best version of an algorithm [15]. WMA gives guarantees on performance showing that WMA can not make many more mistakes than the best version that is being used. WMA's guarantees are strong enough to allow an adversarial type of analysis; however, we find that, when dealing with a distribution generating the instances, predicting with the algorithm that is currently making the fewest mistakes works well.

The main disadvantage with the on-line model is the necessity to build a classifier after every instance. This is not a problem with the algorithms considered in this paper. These algorithms only take $O(m_t)$ time to perform a trial, where m_t is the number of non-zero attributes in an instance. This is optimal in the sense that it takes $O(m_t)$ time just to read in an instance. One of our goals when modifying these algorithms is to retain this efficiency.

V. Hypothesis Voting

Our first refinement for on-line learning is a voting procedure. Voting modifies the predictions of an on-line algorithm by saving several old hypothesis from past trials. For any trial, each of these saved hypothesis predicts a score for the various labels and the scores for each label are added together. The voting procedure predicts the label with the largest score. Combining the predictions of the hypotheses using their raw scores instead of their individual predictions improves performance [16], [17].

There is a wide range of previous work on using voting techniques to improve the performance of prediction algorithms. In this paper, we only consider previous work that deals with on-line learning.

Our work primarily builds off the on-line hypotheses voting ideas in [9]. The basic idea in [9] is to take a random uniform sample of $h = 30$ hypotheses selected with replacement from the previous trials. A majority vote of these h hypotheses is used for predictions. Related techniques in [12] and [13] use bagging [18] and boosting [14] as motivation to improve on-line voting. We find that the bagging techniques do not perform as well as our voting techniques on the adversarial algorithms we study. Boosting algorithms seem more promising. Unfortunately, boosting requires the basic algorithms to weight the importance of instances. This is beyond the scope of this paper.

In [19], a technique is given that votes by combining the scores of all previous hypotheses. This is efficiently implemented for linear-threshold functions by storing the average value of each weight. When used with the basic algorithms of this paper, the averaging technique run in $O(n)$ time where n is the number of attributes. We have found this averaging technique particularly effective for sub-expert problems.

A. Voting Motivation

The main motivation for our voting technique is based on the previous observation that many adversarial algorithms have unstable accuracy in their hypothesis when instances are generated by a distribution. Voting helps solve this problem in two ways. First, it uses a large number of hypotheses to remove the effects of poor hypotheses. If there are few bad hypotheses, they do not out vote the majority of good hypotheses. Second voting improves accuracy by combining the influence of somewhat independent hypothesis. This is similar to bagging [18]. When a noisy instance occurs, the hypothesis is often perturbed to a poor hypothesis. The algorithm will continue to make updates on good, non-noisy trials in an attempt to correct the hypothesis. Since the trials are drawn from a distribution, the hypotheses are randomly perturbed and corrected based on the distribution. For many distributions, this causes the partially correct hypotheses to perform well

on different instances. Based on this intuition, accurate yet diverse hypotheses are likely to be spread out over the various trials.

B. Voting Algorithm

In this section, we give the details of transforming a basic algorithm B into voting algorithm V-B. We develop the voting technique gradually by giving simpler forms of the algorithm and continually adding details until we get the complete algorithm. These techniques work for either basic linear-threshold algorithms or basic sub-expert algorithms.

For voting, we want to choose hypotheses that are spread out over the trials to increase the likelihood that the hypotheses are independent. The further apart the hypotheses the more likely that they have been perturbed in different ways. Our first refinement picks h hypotheses from regularly spaced trials. For example, if $h = 4$ and the algorithm is at trial 100, we vote with hypotheses from trial 25, 50, 75, and 100. However to always keep a close to equal spacing between hypotheses, we would need to store all the old hypotheses. Instead, we only replace voting hypotheses at certain trials in order to continually double the spacing between the voting hypotheses. Let $(t_1, t_2, \ldots)$ represent the trial numbers from which we have selected hypotheses for voting. For example, when $h = 4$, we fill up the voting with hypotheses $(1, 2, 3, 4)$. To double the spacing we continually double the trial numbers: $(2, 4, 6, 8)$, $(4, 8, 12, 16)$, and so on replacing half of the existing hypotheses every time we double the spacing.

To further refine the above technique, we search for high accuracy hypotheses. Assume the above technique has hypotheses from trials $(128, 256, 384, 512)$. Instead of these trials, we search a window of trials centered around these numbers. The size of the window is one plus the minimum of w and half of the current hypotheses spacing. Letting $w = 100$, our example gives trial 256 a window from 224 to 288. As soon as we reach trial 224, we save the current hypothesis for voting. Call this hypothesis h_1. For each following trial, up to trial 288, if we get a new hypothesis, we estimate its accuracy. If that accuracy is higher than the estimate for h_1, we replace h_1 with the new hypothesis. This testing and replacement continues until we reach the end of the window. We estimate the accuracy by keeping track of the number of instances the hypothesis predicted correctly during the on-line learning. Optionally, we can improve this estimate by saving r recent instances to test the hypothesis.

Our next modification is simple and essentially free. We predict with the basic algorithm if it is has made less mistakes than the voting. We keep track of how many mistakes the voting procedure would make if it was making every prediction and compare it to the number of mistakes made by the basic algorithm. Another free modification is to use the current hypothesis of the basic algorithm as a voting hypothesis. Even though it may not always be accurate, it is the most current hypothesis.

If the voting procedure is doing poorly, it may be caused by a large number of early, poor hypotheses. These poor hypotheses can stay around a long time because of the doubling technique. For example, if $h = 30$, the hypothesis from trial 16 will still be used in the voting procedure during trial 480. We want to be able to restart the voting procedure to remove these old hypotheses and replace them with more recent hypotheses. We restart the voting procedure when the basic algorithm is making fewer mistakes than voting. In order to give the voting a chance to learn, we wait a certain number of trials before considering a restart. At the beginning, we wait d trials. After every restart, we double this number of trials. For example, after the third restart, the algorithm will wait an additional $8d$ trials before it checks for a restart.

Our last refinement is a way to combine v different learning algorithms into a single voting technique. This is especially useful for algorithms that have parameter choices. Run the v basic algorithms on the same sequence of instances. For each basic algorithm we keep track of the total number of mistakes. When a hypothesis is needed for voting, we take the current hypothesis from the basic algorithm that is making the least mistakes. This is an inexpensive way to effectively use the best algorithm for a particular problem. We call this algorithm V-Combine. This is the main voting algorithm of this paper.

The cost of our voting technique is small enough to be practical with the efficient on-line learning algorithms we use in this paper. Let n be the number of attributes and m be the maximum number of nonzero attributes. Most of the basic algorithms in this paper can be implemented to perform predictions and updates in $O(m)$. Assuming T is the current trial number, the total cost for V-Combine is $O(vmT + hmT + (rwm + wn)h(\log T)^2)$. While this may seem expensive, the parameters can be fixed at reasonably small values to help control the cost. For this paper, we always set $h = 20$, $r = 100$, and $w = 100$.

In chapter VII, we give experiments that show this voting technique improves performance. However, it is important to realize that the technique is designed for problems where an instance is generated by a distribution. If the instances are generated by an adversary then the voting procedure can cause extra mistakes. However, there is a limit to the number of extra mistakes caused by the voting procedure. Because the voting procedure defaults to predicting with the basic algorithm when basic algorithm is making fewer mistakes than voting, the algorithm has a mistake bound against an adversary that is close to the basic algorithm's mistake bound.

VI. Instance Recycling

In this section, we explain the technique of instance recycling and explain how it improves the performance of certain on-line learning algorithms when instances are generated by a distribution.

The idea is similar to the cycling instances that is used in batch learning. When using on-line algorithms, such as Perceptron, for a batch problem, one often extends a limited number of trials by cycling through the sequence of instances several times. It is hoped the extra updates that occur when cycling will improve the accuracy of the final hypothesis. Here

we modify that technique for on-line learning where the goal is to minimize the number of mistakes. We show that these techniques can improve the performance of on-line algorithms.

A. Instance Recycling Algorithm

Assume B is an on-line algorithm. Let R-B refer to the transformed algorithm that recycles old instances. The technique saves the s most recent instances in an array and uses these old instances to perform extra trials. Any mistake occurring on an old instance is not counted as a real mistake on the learning problem since it is entirely internal to the algorithm. For this reason, we call mistakes on old instances internal mistakes.

Algorithm R-B is the same as algorithm B except that the update operation is extended. After a normal update, algorithm R-B cycles through the old instances. The old instances are treated as new trials including a potential update of the hypothesis. Every time R-B updates the hypothesis with an instance, a variable associated with the instance is incremented. This variable keeps track of the number of times the instance has been used in an update. When the counter reaches u, the instance can no longer be used for updates. The cycling continues until a pass through all the usable old instances does not generate any updates. This is guaranteed to terminate since there is a maximum number updates for each instance. The process repeats on the next new instance that causes an update.

The cost of this recycling technique is small enough to be practical with the efficient on-line learning algorithms we use in this paper. At most the algorithm can update every instance u times. This means the algorithm updates $O(uT)$ times. After every update, we may search the entire old instance list for a new instance to update. We may need to make a prediction on each of these old instances giving a total cost of $O(usmT)$. In practice, the number of predictions is only a fraction of this number since we generally have fewer updates. In this paper, we set $u = 5$ and $s = 100$ for all experiments.

B. Instance Recycling Justification

Instance recycling is effective for a variety of reasons. The algorithms we consider have a bound on the total number of mistakes for instances generated by an adversary. Based on the recycling update procedure, this bound is on the total number of mistakes including internal mistakes. Therefore internal mistakes can potentially reduce the number of mistakes on new instances.

Recycling is particularly effective for mistake-driven algorithms. Mistake-driven algorithms skip many updates on instances because the instance is predicted correctly. In a sense, these instances are wasted because they have no effect on the mistake-driven algorithm. However, given the hypothesis accuracy instability of these algorithms, a past instance may no longer be predicted correctly after an update. Therefore it is useful to recycle these old instances for more updates. An update from an old instance may even have a cascading effect. Another old instance may be predicted incorrectly causing further updates. These extra updates cause internal mistakes that help lower the number of real mistakes.

Our final reason the recycling technique gives good performance is that it increases the accuracy of the hypothesis used by mistake-driven algorithms. Recycling is finished when there are no more possible updates. This means that either the update count for each instance is at its maximum or all the old instances that are not at their maximum are predicted correctly. If we assume that most instances are at their maximum then the algorithm has made a large number of internal mistakes, which should help lower the number of real mistakes. If many of the instances are not at their maximum count then these instances are predicted correctly by the current hypothesis. Therefore, we must have a hypothesis that is accurate for a sample of recent instances. In practice, we find that many of the instances do not reach the maximum count and therefore the recycling often returns fairly accurate hypotheses.

The recycling parameter u is useful when dealing with the effects of noise. If we know their is no noise in the concept then setting $u = \infty$ will give the best results. The algorithm should recycle over the old instances until the hypothesis is consistent with these instances. Every internal mistake can help lower the number of real mistakes. As noise is added to the problem, updating on a noisy instance can mislead the algorithm. Recycling tends to focuses on these noisy instances, and this has the potential to increase the number of mistakes. Fortunately, $u = 1$ is a safe choice. It preserves the adversarial mistake bound of the basic algorithm because it makes at most one mistake on each noisy instance.

VII. Artificial Data Experiments

In this paper, we focus our experiments on the sub-expert problem. For extensive experiments with linear-threshold functions, refer to [7]. In particular, we are interested in how voting and instance recycling improve sub-expert algorithms when dealing with instance noise in the case of batch learning.

Our experiments use the following artificial data. Let n be the total number of sub-experts where r of these sub-experts are relevant. For each trial, every sub-expert randomly picks a class from k classes. Each sub-expert predicts 1 for its selected class and 0 for the other classes. The class that is selected most often from the relevant sub-experts is the class of the instance. We place an order on the relevant sub-experts to handle ties. If there is a tie in the voting between several classes, the instance is labeled according to the smallest class value involved in the tie. To add noise to the problem, we use a parameter p. With probability p, the label for an instance is randomly changed to one of the other labels. In the rest of the paper, we refer to this as the majority learning problem.

In order for the algorithms to learn how to label the previously mentioned sub-expert ties, we need to include k threshold sub-experts, one for each class. The threshold sub-expert corresponding to class i always predicts 1 for class i and 0 for the remaining classes. For example, when $r = 10$, $k = 5$, and $n = 20$ then the target function for Balanced Winnow gives each relevant expert a weight of $1/12$, and

the 5 threshold experts weights of $4/60$, $3/60$, $2/60$, $1/60$, and 0. (The higher threshold expert wins prediction ties.) This gives $\delta = 1/60$ and an upper-bound on mistakes of $7329 + 1.35\hat{N}$. As we will see, our experimental results are much more positive. In fact, our main algorithm VR-Combine makes on average only 34 mistakes on this problem over 5000 trials when there are no noisy instances.

For our experiments, we use the 15 versions of Sub-expert Balanced Winnow with an α multiplier range from 1.01 to 1.6. It is possible that all these Winnow algorithms will perform poorly because 1.01 is too large for the margin of the problem. Therefore, we also use Perceptron and 15 versions of Sub-expert ALMA(p). The ALMA(p) algorithm has a parameter p that controls its behavior. When $p = 2$ the algorithm behaves like Perceptron, with additive updates. As p increases the updates become more multiplicative. We use p values from 2 to 9 in steps of 1/2. For all these cases, an upper-bound on the number of mistakes exists [4].

In preliminary experiments, we noticed that a large number of voting hypotheses were needed to optimize the results. In particular, as the number of classes in the majority learning problem increases, more voting hypothesis are needed to reduce the mistakes. Previous experiments with binary labels had show that $h = 20$ voting hypothesis was generally sufficient [7]. However, using sub-experts and $k = 20$ classes, we need $h = 500$ voting hypothesis to get the best results. Because of the expense associated with this many voting hypotheses, we decided to include another type of basic algorithm. As described in Section V, Freund and Schapire [19] use a technique that keeps track of a single voting hypothesis that is the average of all the previous hypotheses from every trial of on-line learning. This efficiently allows the algorithm to vote with all of the old hypotheses. For a given basic algorithm B, we call this averaged version A-B.

Even with averaging, we still need a way to get select the best algorithm for a problem. One possibility is to create an algorithm that computes an average using hypotheses from the basic algorithm that is currently making the least mistakes. Unfortunately, we find that the hypotheses from different algorithms are not so easy to average, and this sometimes degrades performance. Instead, we continue to use the V-Combine algorithm with $h = 20$ voting hypotheses, but we include the averaging algorithms as basic algorithms from which it can select hypotheses. Therefore, our experiments have 62 basic algorithms: the original 31 basic algorithms and 31 averaged versions of these algorithms.

In Figure 5, we give results for the majority learning problem when $r = 10$, $k = 5$, $n = 20$, and $p = .05$. This graph includes plots of the total number of mistakes made by various algorithms during 5000 trials. This creates a monotonically increasing plot for each algorithm where the slope of the plot is the average error rate of the algorithm at that trial. The plots are fairly smooth because the results are averaged over 20 independent random runs of the experiment. The most expensive algorithm is VR-Combine where each of the 20 runs took approximately 5.5 seconds on an AMD 2600+.

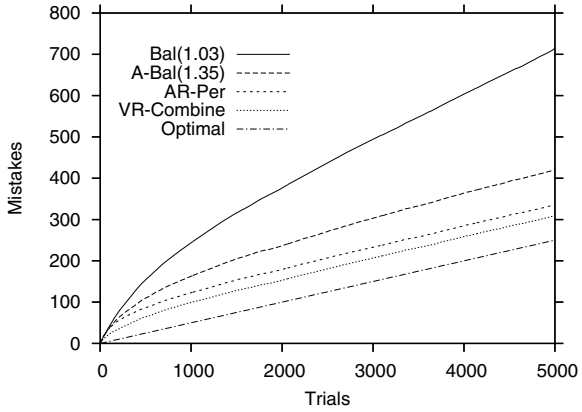

Figure 5. Mistake curves on majority learning problem with $r = 10$, $k = 5$, $n = 20$, and $p = .05$

Notice how the algorithms have a greater error rate at the beginning of the trials and then settle towards a fixed slope line. This is typical for our average plots when dealing with distributions generating the data. At the beginning there is a learning phase. This is followed by a phase where the algorithm does not improve its average error rate. This transition is gradual, and when learning is difficult, such as when there are many attributes, it can require more instances than available to get to the fixed error phase.

In Figure 5, we plot the best algorithm from different groups. First, we plot the best basic algorithm from our set of 31 algorithms without averaging or instance recycling. Next, we plot the best averaging algorithm without instance recycling. This is followed by the best averaging algorithm with instance recycling. The instance recycling uses $s = 100$ and $u = 5$. All of our recycling algorithms use these values as they were found to give good performance without excessive cost on our experiments. The final two algorithms are VR-Combine and the optimal algorithm. Notice that both VR-combine and AR-Per give close to optimal performance with only a small increase in the number of mistakes and an asymptotic error rate that is roughly equal to the optimal classifier. This is typical; VR-Combine performs the best in all of our experiments.

Our main goal with these algorithms is to reduce the effects of noisy instances on the performance of these algorithms. In particular, for fusion problems we are interested in a batch setting that learns the final weights to combine a set of classifiers. In Table I, we give the error-rate of the final hypothesis of an algorithm with a range of p values for the majority learning problem when $r = 10$, $k = 5$, and $n = 20$. The experiments consists of 5000 trials and the last hypothesis is sampled with 50,000 instances to estimate its accuracy. We averaged our results by independently performed this experiment 20 times. In the table, we report the best algorithm out of a set of algorithms, and we use a t-test to give a 95% confidence interval for each result.

As can be seen in the table, VR-Combine always gives the

Table I
FINAL ERROR-RATE ON MAJORITY LEARNING PROBLEM WITH $r = 10$, $k = 5$, $n = 20$, AND 5000 TRAINING TRIALS.

Noise	VR-Combine	Best(AR-B)	Best(A-B)	Best(B)
		AR-Perceptron	A-Perceptron	Alma(2)
$p = 0$	$0 + .00001$	$0 + .00001$	$.00110 \pm 0.00021$	$.00738 \pm 0.00090$
		AR-Perceptron	A-ALMA(4)	Balanced(1.03)
$p = .01$	$.01029 \pm 0.00012$	$.01029 \pm 0.00012$	$.01149 \pm 0.00028$	$.02892 \pm 0.00124$
		AR-Perceptron	A-Balanced(1.35)	Balanced(1.03)
$p = .05$	$.05148 \pm 0.00023$	$.05172 \pm 0.00026$	$.05781 \pm 0.00092$	$.10245 \pm 0.00312$
		AR-Perceptron	A-Balanced(1.5)	Balanced(1.03)
$p = .1$	$.10237 \pm 0.00027$	$.10300 \pm 0.00040$	$.11211 \pm 0.00112$	$.21769 \pm 0.00500$
		AR-Balanced(1.2)	A-Balanced(1.6)	Balanced(1.01)
$p = .2$	$.20285 \pm 0.00118$	$.20713 \pm 0.00062$	$.22684 \pm 0.00184$	$.31884 \pm 0.00148$
		AR-Balanced(1.25)	A-Balanced(1.6)	Balanced(1.01)
$p = .3$	$.31155 \pm 0.00132$	$.31047 \pm 0.00121$	$.33598 \pm 0.00143$	$.45549 \pm 0.00255$
		AR-Balanced(1.3)	A-Balanced(1.6)	Balanced(1.01)
$p = .4$	$.42097 \pm 0.00121$	$.42271 \pm 0.00128$	$.43952 \pm 0.00118$	$.54821 \pm 0.00269$

lowest error-rate, and it is close to the optimal result. As the amount of noise increases, it does have more difficulty, but this can be mitigated by increasing the amount of training data. However, the basic version of the on-line algorithm sometimes has a final average error-rate that triples the noise. This is for the best of the basic algorithms; some algorithms perform much worse. In general, it is important to use voting and instance recycling techniques for both on-line learning and batch learning. In on-line learning, we can reduce the number of mistakes, and in batch learning, we can reduce the final error-rate.

VIII. CONCLUSIONS

In this paper, we look at several inexpensive ways to improve the performance of adversarial on-line learning algorithms when they are learning with instances generated by a fixed distribution. Our focus is on using these modified algorithms to learn a fusion of classifiers, and we perform experiments to show that even with large amounts of noise our algorithms can reliably learn good combinations in either a batch or on-line setting.

This work is the first step in our goal of designing a fusion algorithm that can search a large number of potential classifiers to find a close to optimal combination. In the future, we plan on exploring various normalizations of classifier predictions to improve performance. The VR-Combine algorithm should be able to select the ¡normalization that works best for a particular problem. For example, sometimes it is best to combine the raw scores and other times it is best to combine the rank. VR-Combine is able to tolerate a large number of classifiers and determine what gives the best combination. This is similar in motivation to the rank-score techniques of [17], [20].

REFERENCES

[1] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*. New York: Wiley, 1973.

[2] N. Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm," *Machine Learning*, vol. 2, pp. 285–318, 1988.

[3] ——, "Mistake bounds and linear-threshold learning algorithms," Ph.D. dissertation, Computer Science, University of California, Santa Cruz, 1989, technical Report UCSC-CRL-89-11.

[4] C. Gentile, "A new approximate maximal margin classification algorithm," *Machine Learning*, vol. 2, pp. 213–242, 2001.

[5] C. Mesterharm, "Transforming linear-threshold learning algorithms into multi-class linear learning algorithms," Rutgers University, Tech. Rep. dcs-tr-460, 2001.

[6] C. Gentile and M. K. Warmuth, "Linear hinge loss and average margin," in *Advances in Neural Information Processing Systems 11*. MIT Press, 1999, pp. 225–231.

[7] C. Mesterharm, "Improving on-line learning," Ph.D. dissertation, Department of Computer Science, Rutgers, The State University of New Jersey, 2007.

[8] N. Littlestone, "From on-line to batch learning," in *Proceedings of the Second Annual Conference on Computational Learning Theory*, 1989, pp. 269–284.

[9] ——, "Comparing several linear-threshold learning algorithms on tasks involving superfluous attributes," in *Proceeding of the Twelfth International Conference on Machine Learning*, 1995, pp. 353–361.

[10] ——, "Redundant noisy attributes, attribute errors, and linear-threshold learning using Winnow," in *Proceedings of the Third Annual Conference on Computational Learning Theory*, 1991, pp. 147–156.

[11] C. B. D.J. Newman, S. Hettich and C. Merz, "UCI repository of machine learning databases," 1998; http://www.ics.uci.edu/∼mlearn/MLRepository.html. [Online]. Available: http://www.ics.uci.edu/$\sim$mlearn/MLRepository.html

[12] N. C. Oza and S. J. Russell, "Experimental comparisons of online and batch versions of bagging and boosting," in *Knowledge Discovery and Data Mining*, 2001, pp. 359–364. [Online]. Available: citeseer.ist.psu.edu/454938.html

[13] A. Fern and R. Givan, "Online ensemble learning: An empirical study," *Machine Learning*, vol. 53, no. 1-2, pp. 71–109, 2003.

[14] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," in *Proceedings of the Thirteenth International Conference on Machine Learning*, 1996, pp. 148–156. [Online]. Available: citeseer.ist.psu.edu/freund96experiments.html

[15] N. Littlestone and M. K. Warmuth, "The weighted majority algorithm," *Information and Computation*, vol. 108, pp. 212–261, 1994.

[16] D. F. Hsu and I. Taksa, "Comparing rank and score combination methods for data fusion in information retrieval," *Information and Retrieval*, vol. 8, no. 3, pp. 449–480, 2005.

[17] D. F. Hsu, Y. S. Chung, and B. S. Kristel, *Combinatorial Fusion Analysis: Methods and Practice of Combining Multiple Scoring Systems*. Idea Group Inc., 2006.

[18] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996. [Online]. Available: citeseer.ist.psu.edu/breiman96bagging.html

[19] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," in *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, 1998.

[20] J.-M. Yang, Y.-F. Chen, T.-W. Shen, B. S. Kristal, and D. F. Hsu, "Consensus scoring criteria for improving enrichment in virtual screening," *Journal of Chemical Information and Modeling*, vol. 45, no. 4, pp. 1134–1146, 2005.