Using Linear-threshold Algorithms to Combine Multi-class Sub-experts

Chris Mesterharm MESTERHA@CS.RUTGERS.EDU
Rutgers Computer Science Department 110 Frelinghuysen Road Piscataway, NJ 08854 USA

Abstract

We present a new type of multi-class learning algorithm called a linear-max algorithm. Linear-max algorithms learn with a special type of attribute called a sub-expert. A sub-expert is a vector attribute that has a value for each output class. The goal of the multi-class algorithm is to learn a linear function combining the sub-experts and to use this linear function to make correct class predictions. The main contribution of this work is to prove that, in the on-line mistake-bounded model of learning, a multi-class sub-expert learning algorithm has the same mistake bounds as a related two class linear-threshold algorithm. We apply these techniques to three linear-threshold algorithms: Perceptron, Winnow, and Romma. We show these algorithms give good performance on artificial and real datasets.

1. Introduction

In this paper, we define a new type of inductive learning algorithm called a linear-max algorithm. This is an algorithm that learns using a special type of attribute called a sub-expert. We define a sub-expert as a vector attribute that has a value for each possible output class. The value for each class corresponds to the prediction strength the sub-expert gives to each class. A hypothesis prediction can be viewed as a sub-expert. At a minimum, if the hypothesis predicts a single class, the sub-expert can predict 1 for that class and 0 for the remaining classes. Some learning algorithms can use information, such a class probability estimates, to give prediction values for all the sub-expert classes. The linear-max algorithm takes a weighted sum of all the sub-experts and predicts the class that has the maximum value. We call this a linear-max function. The goal of the algorithm is to learn good weights for the sub-experts.

This paper deals with the problem of classifying a sequence of instances. Each instance belongs to one of k classes and is composed of sub-experts. After the algorithm predicts a label for an instance, the environment returns the correct classification. This information can then be used to improve the prediction function for the next instance in the sequence. These three steps, getting an instance, predicting the class, and updating the classifier, are called a trial. The goal of the algorithm is to minimize the total number of prediction mistakes made during the trials. This is commonly referred to as the on-line mistake-bounded model of learning (Littlestone, 1989).

It is intuitive to think of sub-experts as individual classifying functions that are attempting to predict the target function. Even though the individual sub-experts may not be perfect, the linear-max algorithm attempts to learn a linear-max function that combines them and does well on the target. In truth, this picture is not quite accurate. The reason we call them sub-experts and not experts is because even though an individual sub-expert might be poor at prediction, it may be useful when used in a linear-max function. The term experts more commonly refers to the situation where the performance of the algorithm is measured with respect to a single best expert (Cesa-Bianchi et al., 1997; Littlestone & Warmuth, 1994) as opposed to a combination of sub-experts.

Linear-max algorithms are related to linear-threshold algorithms and were motived by a desire to extend linear-threshold functions from two class prediction to multi-class prediction. We accomplish this by using a linear-threshold algorithm to help solve a linear-max problem. This technique is useful both theoretically and practically. It allows the existing mistake bounds from the linear-threshold algorithms to be carried over to the linear-max setting. With no extra work, one can the apply the bounds of existing or future linear-threshold learning algorithms to multi-class problems.

This paper is based on work in Mesterharm (2001) which gives a general framework and proof to extend a wide range of linear-threshold algorithms to multi-class problems. In this paper, we give a slightly less general result due to space considerations; please refer to the technical report for more details. While the proof deals with sub-experts, in Mester-

Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

harm (2001) we give a technique to use sub-experts to solve multi-class problems dealing with real-valued attributes. This produces algorithms that are similar to algorithms that can be derived with techniques given in Nilsson (1965) and Har-Peled et al. (2003). For this reason, we will focus on sub-experts in this paper, however, it is straightforward to combine the two techniques to get hybrid algorithms that combine sub-experts and attributes.

There has been a large amount of previous work on using binary prediction for multi-class attribute based problems. Allwein et al. (2000) and Dietterich and Bakiri (1995) both look at learning separate binary classifiers to make multi-class predictions. Crammer and Singer (2001) gives a family of multi-class Perceptron algorithms with generalized update functions. Har-Peled et al. (2003) gives a method for handling more general multi-class learning problems such as ranking.

For combining experts in an on-line setting much of the work focuses on learning a single best expert (Cesa-Bianchi et al., 1997; Littlestone & Warmuth, 1994). For learning a combination of experts, Blum (1995) gives a version of a multi-class Winnow sub-expert algorithm, while Mesterharm (2000) generalizes that algorithm and gives a stronger mistake bound. This paper builds on the previous work by giving a technique to extend a wide range of linear-threshold algorithms to a combination of sub-experts while retaining many of the benefits including mistake bounds.

Another goal of this paper is to show that sub-expert algorithms are useful for practical machine learning problems. Linear-threshold algorithms have been used successfully on many practical problems. Since linear-max problems are solved using linear-threshold algorithms, we should get similar real-world performance with both types of algorithms. In addition, the Winnow sub-expert extension has already given state of the art performance on a calendar scheduling problem in Blum (1995). In this paper, we perform experiments to show that it is useful to extend other algorithms, besides Winnow, with sub-experts. We give experiments with sub-experts using Perceptron (Rosenblatt, 1962), normalized Winnow (Littlestone, 1989), and the Relaxed On-line Maximum Margin Algorithm (Romma) (Li & Long, 2000), and we show that both Perceptron and Romma give superior performance on specific types of problems.

2. Linear-max Algorithms

In this section, we give the definitions of a linear-max algorithm and its related linear-threshold algorithm. Before we give the formal definitions we need to cover how sub-experts work, how they make predictions, and how the predictions are combined.

2.1. Sub-experts

First we give the definition of a sub-expert. A sub-expert makes k predictions, one for each possible output class. These predictions correspond to the rating a sub-expert gives to each class; higher numbers corresponding to a better rating. Let $x_i^j \in R$ be the prediction sub-expert i gives for class j. Notice that the rating of a class is relative. Looking at class a and b, a sub-expert prefers class a over b if $x_i^a - x_i^b > 0$. It is these differences between class predictions that are crucial to the operations of the multi-class algorithms in this paper. Sometimes, it is useful to bound the size of these differences. In those cases, assume for each expert i and all $a, b \in \{1, \ldots, k\}$ that $x_i^a - x_i^b \in [-\nu, \nu]$ where $\nu \in R$.

Here are some examples of the types of knowledge that sub-experts can encode. For these examples assume that $\nu = 1$. A sub-expert that predicts $\langle 1, 0, 0 \rangle$ gives maximal preference to class 1 over class 2 or 3. This same sub-expert makes no distinction between class 2 and 3. A sub-expert that predicts $\langle -1, 0, 0 \rangle$ again makes no distinction between class 2 and 3 but maximally prefers class 2 or 3 over class 1. Lastly a sub-expert that predicts $\langle 0, 0, 0 \rangle$ makes no distinction between any of the classes. Notice that adding a constant to each prediction class in a sub-expert does not change the differences between classes. For example $\langle 1/3, 1/3, 1/3 \rangle$ is the same prediction as $\langle 0, 0, 0 \rangle$, and $\langle 0, 1, 1 \rangle$ is the same prediction as $\langle -1, 0, 0 \rangle$. In general, we can represent all sub-experts by restricting all x_i^j to $[0, \nu]$, but sometimes a wider range of prediction values is useful for notational economy or algorithm efficiency.

2.2. Prediction

Now we will show how to use several sub-experts to make a global prediction. Assume there are n sub-experts, and each sub-expert is assigned a weight. Let $\mathbf{w}$ be the vector of n weights where w_i is the weight of sub-expert i. We combine the information from the weights and the sub-experts to compute a vote for each class.

$$w_1 \begin{pmatrix} x_1^1 \\ \vdots \\ x_1^k \end{pmatrix} + w_2 \begin{pmatrix} x_2^1 \\ \vdots \\ x_2^k \end{pmatrix} + \cdots + w_n \begin{pmatrix} x_n^1 \\ \vdots \\ x_n^k \end{pmatrix}$$

Define the voting function for class j and weights $\mathbf{w}$ as $\zeta_{\mathbf{w}}(j) = \sum_{i=1}^{n} w_i x_i^j$. The algorithm predicts the class with the highest vote, $\arg\max_j \zeta_{\mathbf{w}}(j)$. (On a tie the algorithm predicts any class involved in the tie.) We call the function computed by this prediction scheme a linear-max function since it is the class of the maximum value taken from a linear combination of the sub-expert predictions.

Even though some sub-experts may not individually give accurate predictions, they may be useful in a linear-max

function. For example, sub-experts might be used to add threshold weights. This is done by adding an extra sub-expert for each class. A threshold sub-expert would always predict with 1 for its corresponding class and 0 for the remaining classes. While a threshold expert makes a poor classifier by itself, when combined in a linear-max algorithm, it gives a useful expansion of the set of target functions.

2.3. Linear-threshold Algorithm Definition

We will show how any linear-threshold algorithm of the following form can be transformed to a linear-max algorithm.

Initialization
    $\mathbf{w} := \mathbf{w}_{init}$ with $\mathbf{w}_{init} \in R^n$.
Trials
    Instance: Attributes $\mathbf{z} \in [-\nu, \nu]^n$ with $\nu \in R \cup \infty$.
    Prediction: Predict $\tau = 1$ if $\sum_{i=1}^{n} w_i z_i > 0$
        otherwise predict $\tau = -1$.
    Update: Let $\phi \in \{-1, 1\}$ be the correct label
        $\mathbf{w} := f(\mathbf{w}, \phi\mathbf{z}, \phi)$

For the linear-threshold algorithm, the update is a function of the current weights and the attributes multiplied by the label. For example, Perceptron uses $f(\mathbf{w}, \phi\mathbf{z}, \phi) = \mathbf{w} + \phi\mathbf{z}$ for updates on mistakes.

2.4. Linear-max Algorithm Definition

Next we define the related linear-max algorithm. The important thing to note is that the algorithm uses the same update function f as the linear-threshold algorithm. This is the key that binds them together. We use the update procedure from a linear-threshold algorithm to perform updates on the related linear-max algorithm.

Initialization
    $\mathbf{w} := \mathbf{w}_{init}$ with $\mathbf{w}_{init} \in R^n$.
Trials
    Instance: Sub-experts $(\mathbf{x_1}, \ldots, \mathbf{x_n})$.
    Prediction: Let $\zeta_{\mathbf{w}}(j) = \sum_{i=1}^{n} w_i x_i^j$.
        Predict class λ such that for all $j \neq \lambda$
        $\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(j)$.
    Update: Let $\rho \in \{1, \ldots, k\}$ be the correct label.
        If $\rho \neq \lambda$ then $\mathbf{w} := f(\mathbf{w}, (\mathbf{x}^\rho - \mathbf{x}^\lambda), 1)$.

When the linear-max algorithm makes a mistake, we update with an instance that would cause the linear-threshold algorithm, with the same weights, to make a mistake. There is some freedom in picking this incorrect instance. For example, for a problem with $k = 5$ classes let $(y_1, y_2, y_3, y_4, y_5)$ be the five classes sorted based on the voting function. The algorithm predicts $\lambda = y_1$ and assume the correct label $\rho = y_4$. Consider the following $k - 1$ instances for the linear-threshold algorithm: $(x^\rho - x^{y_1})$, $(x^\rho - x^{y_2})$, $(x^\rho - x^{y_3})$, $(x^\rho - x^{y_5})$ where all have a label $\phi = 1$. The first three of these correspond to instances that would be misclassified. For example, $\sum_{i=1}^{n} w_i(x^\rho - x^{y_3}) \leq 0$. The last is correct based on the current weights. We could use any of the first three for the update, but to keep things simple, we pick $x^\rho - x^\lambda$. We conjecture that other possible instances, such as linear combinations of the above, will give a sub-expert based family of algorithms similar to the ultraconservative algorithms of Crammer and Singer (2001).

3. Mistake Bound Proof

In this section, we prove the linear-max algorithm has the same bounds as the related linear-threshold algorithm. Throughout this section assume we are running a linear-max and a linear-threshold algorithm.

First we will show that the algorithms always have the same weight values. Initialize both algorithms with the same weights. Assume we are given an instance of the linear-max problem. Use the linear-max algorithm to make a global prediction with the sub-experts. This prediction is λ such that for all $j \neq \lambda$ $\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(j)$. The environment will return the correct label ρ. If the prediction is a mistake ($\lambda \neq \rho$) then create a new instance for the linear-threshold algorithm of the form $z_i = x_i^\rho - x_i^\lambda$. Because of our earlier assumptions on sub-experts, each $z_i \in [-\nu, \nu]$. Let the label for this instance be $\phi = 1$. Input $\mathbf{z}$ and ϕ into the linear-threshold algorithm and update the weights in both the linear-threshold and linear-max algorithms. Since both algorithms have the same update function and now have the same input parameters, both algorithms will make the same weight updates. Repeat this procedure for each trial. Using a simple inductive argument, both algorithms will always have the same weights.

Now we will show that every time the linear-max algorithm makes a mistake then the linear-threshold makes a mistake. This can be seen by looking at the prediction procedure of the linear-max algorithm. A mistake can only occur in the linear-max algorithm if the vote for the correct label is less than or equal to the vote for the predicted label. This means that $\sum_{i=1}^{n} w_i x_i^\rho - \sum_{i=1}^{n} w_i x_i^\lambda \leq 0$. This can be rewritten as $\sum_{i=1}^{n} w_i(x_i^\rho - x_i^\lambda) \leq 0$. Because the linear-threshold algorithm uses the same weights, the instance $z_i = x_i^\rho - x_i^\lambda$ causes the linear-threshold algorithm to predict -1. Since we have constructed the instance such that the correct label for the linear-threshold algorithm is $\phi = 1$, this causes a mistake. Therefore the number of mistakes made by the linear-max algorithm is upper-bounded by the number of mistakes made by the linear-threshold algorithm. Using this upper-bound, we can apply any mistake bound from a linear-threshold algorithm to its related linear-max algorithm.

Of course, the linear-max algorithm only has a good bound if the the linear-threshold algorithm has a good bound. In order to get a good bound, the instances sent to the linear-threshold algorithm need to satisfy certain algorithm and proof dependent assumptions. For example, if the attribute z_i needs to have a certain distribution, we would need $x_i^\rho - x^\lambda$ to also have this distribution. This makes it difficult to use the linear-max transformation for many Bayesian algorithms that make distributional assumptions. Fortunately many algorithms have bounds that only depend on the existence of target weights that perfectly classify the data.

Assume there exists a set of target weights that perfectly classify the sub-experts. These same weights must perfectly classify any instance passed to the linear-threshold algorithm since every instance is of the form $(x^\rho - x^\lambda)$ with label 1. These instances will always predict 1 since $\sum_{i=1}^n w_i x_i^\rho$ must return the highest vote. Using these weights for the linear-threshold algorithm will give the mistake-bound. While this perfect classification assumption may seem overly strong, the proof technique can be modified to allow noisy instances (Littlestone, 1989; Littlestone, 1991). Most of these modifications will carry over to the sub-expert setting, however to simplify our presentation, any bounds we give in the rest of the paper refer to the no-noise framework.

4. Specific Linear-max Algorithms

In this section, we give some transformations of linear-threshold algorithms into linear-max algorithms. This is done by replacing the generic update function f from Section 3 with a specific update function from a linear-threshold algorithm. We also include some relevant details for the various algorithms such as mistake bounds.

4.1. Perceptron Algorithm

This is the multi-class version of the classic Perceptron algorithm (Rosenblatt, 1962). It fits directly into the linear-threshold framework of the linear-max transformation.

Initialization
$\forall i \in \{1, \ldots, n\}\ w_i := 0.$
Trials
Instance: Sub-experts $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ where $\mathbf{x}_i \in R^k$.
Prediction: Let $\zeta_{\mathbf{w}}(j) = \sum_{i=1}^n w_i x_i^j$
Predict a class λ such that for all $j \neq \lambda$
$\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(j)$.
Update: Let ρ be the correct label. If $\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(\rho)$
$\forall i \in \{1, \ldots, n\}\ \ w_i := w_i + (x_i^\rho - x_i^\lambda)$.

The bounds for the linear-max Perceptron algorithm depend on certain problem dependent parameters. These are the same parameters that are used for the linear-threshold algorithm, but the values of the parameters come from the linear-max problem. Let $s = \max_{\textit{trials}} \|(\mathbf{x}^\rho - \mathbf{x}^\lambda)\|_2$. Let $\mathbf{u}$ be a weight vector that correctly classifies all instances. Let $\delta = \min_{\textit{trials}} (\min_{j \neq \rho} (\zeta_{\mathbf{u}}(\rho) - \zeta_{\mathbf{u}}(j)))$; this is a margin value where no instance occur. With these parameters, the number of mistakes is less than or equal to $s^2 \|\mathbf{u}\|_2^2 / \delta^2$ (Duda & Hart, 1973).

4.2. Romma Algorithm

This is the multi-class version of the Romma algorithm (Li & Long, 2000). Again it is a fairly straightforward to transform Romma to the the linear-max setting.

Initialization
$\forall i \in \{1, \ldots, n\}\ w_i := 0.$
Trials
Instance: Sub-experts $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ where $\mathbf{x}_i \in [0,1]^k$
Prediction: Let $\zeta_{\mathbf{w}}(j) = \sum_{i=1}^n w_i x_i^j$
Predict a class λ such that for all $j \neq \lambda$
$\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(j)$.
Update: Let ρ be the correct label. If $\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(\rho)$
Let $z_i := x_i^\rho - x_i^\lambda$.
Let $c := \frac{\|\mathbf{z}\|_2^2 \|\mathbf{w}\|_2^2 - \mathbf{w} \cdot \mathbf{z}}{\|\mathbf{z}\|_2^2 \|\mathbf{w}\|_2^2 - (\mathbf{w} \cdot \mathbf{z})^2}$,
Let $d := \frac{\|\mathbf{w}\|_2^2 (1 - \mathbf{w} \cdot \mathbf{z})}{\|\mathbf{z}\|_2^2 \|\mathbf{w}\|_2^2 - (\mathbf{w} \cdot \mathbf{z})^2}$
If first update then
$\forall i \in \{1, \ldots, n\}\ \ w_i := z_i / \|\mathbf{z}\|_2$.
Else
$\forall i \in \{1, \ldots, n\}\ \ w_i := c w_i + d z_i$.

The bounds found in Li and Long (2000) for Romma are the same as the bounds given above for the Perceptron algorithm.

4.3. Normalized Winnow Algorithm

This is the multi-class version of the normalized Winnow algorithm that learns linear threshold functions (Littlestone, 1989). This algorithm is similar to the multi-class Winnow algorithm in Blum (1995). The main difference is that it allows expert predictions to be real numbers.

Initialization
$\forall i \in \{1, \ldots, n\}\ w_i := 1.$
Trials
Instance: Sub-experts $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ where $\mathbf{x}_i \in [0,1]^k$
Prediction: Let $\zeta_{\mathbf{w}}(j) = \sum_{i=1}^n w_i x_i^j$
Predict a class λ such that for all $j \neq \lambda$
$\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(j)$.
Update: Let ρ be the correct label. If $\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(\rho)$
$\forall i \in \{1, \ldots, n\}\ \ w_i := \alpha^{x_i^\rho - x_i^\lambda} w_i$.

Again the bounds for the linear-max Winnow algorithm depend on certain problem dependent parameters. Let $\mathbf{u}$ be a

weight vector that correctly classifies all instances. Let $\delta = \min_{trials}(\min_{j \neq \rho}(\zeta_{\mathbf{u}}(\rho) - \zeta_{\mathbf{u}}(j)))$ and $\alpha = (1-\delta)^{-1/2}$. The number of mistakes is less than or equal to $\ln(n)/\delta^2$ (Littlestone, 1989). Mesterharm (2000) has a specific analysis that derives the same mistake bound for this linear-max algorithm.

4.4. Other Linear-max Algorithms

Of course, these are not the only linear-threshold algorithms that can be transformed into linear-max algorithms. For example, Quasi-additive algorithms are a recent and relatively unexplored class of linear-threshold learning algorithms (Grove et al., 1997) that include an uncountably infinite number of algorithms including Perceptron and normalized Winnow. The linear-max setting is sufficiently flexible to include all the Quasi-additive algorithms and more. In this paper, we only talk about Perceptron, Romma, and Winnow since these are popular efficient algorithms that have performed well in practice.

We can also change the linear-max definition to allow more updates. If there are k classes, consider the $k-1$ instances mentioned in Section 2.4. These instances can be used for updates. This is suggested in (Har-Peled et al., 2003) for on-line learning of the attribute based problem. We can even go further and repeatedly cycle through these instances until all are correctly classified. While these techniques should improve performance for problems without noise, it could lead to problems when noise becomes excessive.

Another variation is to use nonconservative algorithms. A nonconservative linear-threshold algorithm makes updates on some instances even though the algorithm has not made a prediction mistake on the instance. These more aggressive updates will not improve performance against an adversary, but often improve performance in practice (Block, 1962; Li & Long, 2000). If the linear-threshold algorithm is nonconservative, we can make the linear-max algorithm nonconservative by passing some of the correctly classified instances mentioned above.

Last, we can use these instances with a Support Vector Machines (SVM) (Cortes & Vapnik, 1995). These instances encode the information about the correct sub-expert predictions. After T trials there will be a total of $T(k-1)$ instances, and we can pass all these instances to the SVM to learn weights for the sub-experts. We can then use the weights or the support vectors to make predictions in the linear-max setting. This allows us to use a SVM to combine multi-class hypotheses. It may also be possible to use a SVM with a kernel function or even a wider range of learning algorithms to expand beyond simple linear combinations of sub-experts, but one needs to be careful about the effects of nonlinearity.

5. Experiments

In this section, we give experiments to show that multi-class algorithms give good performance, and that the various versions, that correspond to different linear-threshold algorithms, give different and non-dominating results on realistic problems. We perform experiments with normalized Winnow, Perceptron, and Romma. For all these algorithms, we include a set of experts described in the problem and a set of threshold experts. For a k class problem there are k threshold experts with each expert always predicting 1 for its respective class.

The problems we look at are often sparse in the number of experts active on a given trial and in the number of non-zero predictions made by an active expert. An expert is active if at least one of it's label predictions is non-zero. Let r be the number active experts in an instance. Let s be the maximum number of label predictions that are non-zero in any expert. All three algorithms use $O(rs)$ time to predict and update on a trial.

These predictions and updates are fairly straightforward to implement using a sparse representation for the instances. An instance is internally represented by a list of active experts. Furthermore, for each active expert, the algorithm keeps a list of the labels that have non-zero predictions and their value. This representation allows an efficient implementation of any linear-max algorithm that is based on a linear-threshold algorithm that can perform predictions and updates in $O(q)$ time where q is the number of active real-valued attributes. Most algorithms, including Perceptron and normalized Winnow, are already in this form, and many other algorithms can be modified to get $O(q)$ updates. For example, even though the update function of the linear-threshold Romma depends on the norm of the total number of attributes, we store extra information to allow $O(q)$ updates. Using these efficient algorithms, the running time for all of the following experiments combined is a few minutes on a 700 MHz machine.

5.1. Artificial Data

Our first experiments deal with artificial data. Let n be the total number of experts where r of these experts are relevant. For each trial, every relevant expert randomly picks a class from k classes. The expert predicts 1 for the selected class and 0 for the other classes. The class that is selected most often, from the relevant experts, is the class of the instance. We place an order on the relevant experts to handle ties. If there is a tie in the voting for a class, the instance is labeled according to the prediction of the lowest expert that is involved in the tie. The $n-k$ irrelevant experts are active with probability p. If an irrelevant expert is active it randomly picks a class and predicts 1 for that class and 0 for the remaining classes. In the rest of the paper, we will

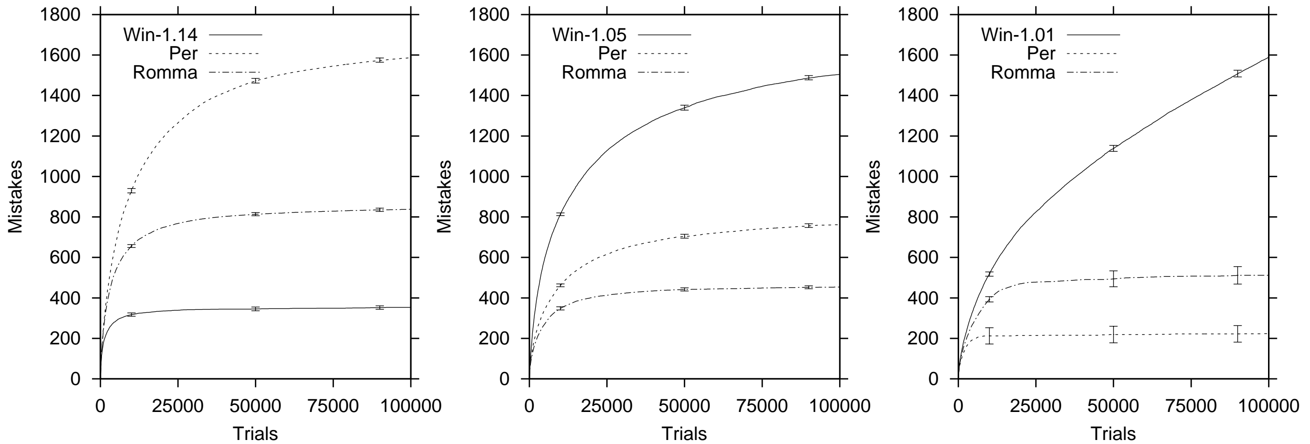

Figure 1. All graphs are for the artificial data concept. The first graph has 5 relevant out of 300 total experts, 3 classes and .5 irrelevant expert probability. The second graph has 10 relevant out of 100 experts, 5 classes and .05 irrelevant expert probability. The third graph has 2 relevant out of 100 experts, 20 classes and .001 irrelevant expert probability. The multiplier use by Winnow is included in the key.

refer to this as the majority learning problem.

Our primary purpose for the artificial data experiments is to clearly show that none of the three algorithms dominates. In figure 1, we show three graphs with a different algorithm giving the best result in each graph. For Winnow, we picked the multiplier parameter that gives the best performance. This multiplier is given in the graph key after the name Win. The graphs give the number of mistakes on the y-axis and the the number of trials on the x-axis. The slope of the graph corresponds to the error rate of the algorithm. Each experiment was run for 100,000 trials, and was averaged over 20 runs. We give confidence intervals at three points along the curve using a t distribution.

The first graph shows normalized Winnow performing best. The majority problem is described by $r = 5$, $n = 300$, $k = 3$, and $p = .5$. It is fairly easy to get normalized Winnow to perform best by using a small number of relevant experts with a large number of irrelevant experts. To further separate the algorithms, we made the instances dense with a large value of p. This will force poor performance by Perceptron and Romma since their bounds depend on the 2-norm of the instance.

The second graph is for Romma. The majority problem is described by $r = 10$, $n = 100$, $k = 5$, and $p = .05$. This problem avoids the region where Winnow dominates.

The last graph shows Perceptron performing the best. The majority problem is described by $r = 2$, $n = 100$, $k = 20$, and $p = .001$. We had some difficulty in getting Perceptron to perform best. Perceptron and Romma have similar bounds and, for the majority problem, Romma often out performs Perceptron. Eventually, we were able to get better performance with a sparse problem that has a large number of classes and only a few relevant experts.

5.2. Calendar Data

Our second set of experiments deals with the Calendar Apprentice problem (Mitchell et al., 1994). In this problem, the goal of the algorithm is to predict information about a meeting that is entered into a calendar application. There are a total of 34 text features used to describe a calendar event. The algorithm predicts the day, duration, location, and start-time of a meeting. When predicting a particular label we include the other labels as features in the instance. The dataset is divided into two users. In this paper, we report results for user two with a total of 553 instances.

Our reason for running the Calendar problem is to see if Perceptron or Romma can improve the performance on any of the label prediction tasks. Blum (1995) used a multi-class version of normalized Winnow to show a performance improvement over C4.5 on these problems. Since our algorithm is essentially the same, we did not expect to see much difference in the results of the Winnow algorithm. However, while the Winnow results for user one was similar to those reported in Blum (1995)[1], we did get different results for user two. We assume the discrepancy can be attributed to differences in feature selection.

We follow the technique of Blum (1995) to create sub-experts for the calendar problem. For every trial, we look at all pairs of features and create an sub-expert that predicts based on the past occurrences of the feature pair. The sub-expert predicts 1 for the label that occurred most frequently over the last five times the feature pair appeared in an instance. In the case of ties, we split the vote evenly across the labels that tied on the prediction. The first time a feature pair appears the sub-expert does not make a prediction.

[1] For user one on the Calendar problem, Winnow performed better than Perceptron and Romma for all four tasks.

Table 1. Accuracy of algorithms on 553 data points from calendar domain. The number after Win is the multiplier used for Winnow.

PROBLEM	COMBINED	PERCEPTRON	ROMMA	WIN-1.5	WIN-2	WIN-3	WIN-5	WIN-10
DAY	**0.468**	**0.468**	0.427	0.407	0.400	0.382	0.382	0.380
DURATION	**0.749**	0.676	0.644	0.741	**0.749**	0.741	0.747	0.729
LOCATION	0.700	0.575	0.552	0.698	**0.703**	0.696	0.682	0.680
START	0.711	0.600	0.557	0.640	0.662	0.673	0.700	**0.716**
AVERAGE	**0.657**	0.580	0.545	0.622	0.628	0.623	0.627	0.626

The Calendar problem does present some difficulties for the formal linear-max setting. At the start of the algorithm, we do not know the number of classes, and we do not know the number of experts. There is not enough space to go into detail, but it is straightforward to add new classes and experts as they appear. We also need to deal with experts that do not predict every trial. This is not a problem because experts who predict zero for all labels do not effect the prediction and do not get updated. For computational speedup, we just ignore these instances in the sparse instance representation.

Table 1 gives the accuracy of the various algorithms on the four tasks. We include several versions of Winnow that correspond to different α multiplier parameters. As can be seen in the table, Perceptron gives better performance on the day prediction task. In addition, we also find that choosing a large multiplier can improve normalized Winnow's performance on the start task.

Given that different algorithms perform well on different tasks, we need a way to combine the performance of the algorithms. The entry called Combined in Table 1 gives the accuracy of the Weighted Majority algorithm (WMA) (Littlestone & Warmuth, 1994) when combining the results of all the other algorithms. WMA keeps a weight for each basic algorithms and predicts according to the weighted sum of the basic algorithm predictions. Each basic algorithm predicts one of the labels and every time a basic algorithm predicts incorrectly, its weight is divided by a multiplier. We looked at a few multiplier values, and while they all were similar, 1.5 gave slightly better performance.

The average performance of the combined WMA is a few percentage points better than the best results for user two given in Blum (1995). However, we did perform slightly worse on the day and location problems. We assume this is due to a difference in representation. However, the point of our experiments is not to give the best performance possible. Instead, we want to show, for a realistic problem, that non-Winnow linear-max algorithms can give good performance. We could most likely get better performance by either combining the algorithms in Blum (1995) with our own using WMA or further exploring the features that lead to the differences in performance.

While WMA is similar to the multi-class techniques we describe in this paper, it has a few important differences. First, it always updates. It does not matter if the global prediction of WMA is correct, the weights are still updated. Second, the update only depends on whether the basic algorithm makes the wrong prediction. The update does not depend on the difference between the correct label and the algorithm's predicted label. These differences are important. For this second layer of learning, we just want to quickly find which algorithm is making the least mistakes and predict with that basic algorithm. Using the algorithms presented in this paper, we will eventually find the better performing basic algorithm, however since these multi-class algorithms do not update as aggressively as WMA, they will make more mistakes in this process. If there is no good combination of experts to learn, it is better to use WMA. As can be seen, in Table 1, WMA often performs as well the best from the group of algorithms. At worst it makes a few extra mistakes. When no basic algorithm dominates all the problems, this allows WMA to give a performance average that exceeds all the basic algorithms.

6. Conclusion

In this paper, we have given a general transformation to convert a linear-threshold learning algorithm into multi-class linear-max learning algorithm. The benefit of this transformation is that it allows the learning of multi-class functions while preserving the theoretical properties of various linear-threshold algorithms.

Linear-max algorithms learn target functions that are composed of sub-experts that make predictions on all classes. They are a natural choice for combining the predictions of different hypotheses such as those generated by different learning algorithms. However, for some problems, it may be difficult to come up with a large set of sub-experts. In the future, we want to come up with standard way to generate useful sub-experts for different types of problems.

Experiments in this paper show that a range of linear-max algorithms are useful in practice. Experiments with Per-

ceptron, Romma, and Winnow show that no single algorithm dominates. In the future, we want to look at ways to effectively combines the performance of these different types of algorithms. As a preliminary investigation in this paper, we looked at combining several basic multi-class algorithms with WMA. The basic algorithms we combined included several versions of Winnow with different multiplier parameters, but there are other possibilities for parameters selection. Romma includes a parameter to make more aggressive updates on correctly classified instances (Li & Long, 2000). In fact, all linear-threshold algorithms could benefit from this type of aggressive updating. The key problem is how to properly set the parameter. In the future, we want to look at ways to effectively search an infinite parameter space to get good performance. As can be seen with the Calendar experiments, parameter choice can effect performance on practical problems for these algorithms.

Acknowledgments

We thank Haym Hirsh and Nick Littlestone for stimulating this work. and Haym Hirsh for reading over the paper and providing valuable comments and corrections.

References

Allwein, E. L., Schapire, R. E., & Singer, Y. (2000). Reducing multiclass to binary: A unifying approach for margin classifiers. *17th International Conference on Machine Learning* (pp. 9–16). Morgan Kaufmann.

Block, H. D. (1962). The perceptron: A model for brain functioning. *Reviews of Modern Physics*, *34*, 123–135.

Blum, A. (1995). Empirical support for winnow and weighted-majority algorithms: Results on a calendar scheduling domain. *Proceeding of the Twelfth International Conference on Machine Learning* (pp. 64–72).

Cesa-Bianchi, N., Freund, Y., Haussler, D., Helmbold, D. P., Schapire, R. E., & Warmuth, M. K. (1997). How to use expert advice. *Journal of the Association for Computing Machinery*, *44*, 427–485.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, *20*, 273–297.

Crammer, K., & Singer, Y. (2001). Ultraconservative online algorithms for multiclass problems. *14th Annual Conference on Computational Learning Theory* (pp. 99–115). Springer, Berlin.

Dietterich, T. G., & Bakiri, G. (1995). Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, *2*, 263–286.

Duda, R. O., & Hart, P. (1973). *Pattern classification and scene analysis*. New York: Wiley.

Grove, A. J., Littlestone, N., & Schuurmans, D. (1997). General convergence results for linear discriminant updates. *Proceedings of the Tenth Annual Conference on Computational Learning Theory* (pp. 171–183).

Har-Peled, S., Roth, D., & Zimak, D. (2003). Constraint classification for multiclass classification and ranking. *Neural Information Processing Systems 15*. MIT Press.

Li, Y., & Long, P. (2000). The relaxed online maximum margin algorithm. *Neural Information Processing Systems Twelve* (pp. 498–504). MIT Press.

Littlestone, N. (1989). *Mistake bounds and linear-threshold learning algorithms*. Doctoral dissertation, Computer Science, University of California, Santa Cruz. Technical Report UCSC-CRL-89-11.

Littlestone, N. (1991). Redundant noisy attributes, attribute errors, and linear-threshold learning using winnow. *Proceedings of the Third Annual Conference on Computational Learning Theory* (pp. 147–156).

Littlestone, N., & Warmuth, M. K. (1994). The weighted majority algorithm. *Information and Computation*, *108*, 212–261.

Mesterharm, C. (2000). A multi-class linear learning algorithm related to winnow. *Neural Information Processing Systems Twelve* (pp. 519–525). MIT Press.

Mesterharm, C. (2001). *Transforming linear-threshold learning algorithms into multi-class linear learning algorithms* (Technical Report dcs-tr-460). Rutgers University.

Mitchell, T., Caruana, R., Freitag, D., McDermott, J., & Zabowski, D. (1994). Experience with a personal learning assistant. *Communications of the ACM*, *37*, 81–91.

Nilsson, N. J. (1965). *Learning machines: Foundations of trainable pattern-classifying systems*. New York, NY: McGraw-Hill.

Rosenblatt, F. (1962). *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Washington, DC: Spartan Books.