Transforming Linear-threshold Learning Algorithms into Multi-class Linear Learning Algorithms

Chris Mesterharm
Rutgers Computer Science Department
110 Frelinghuysen Road
Piscataway, NJ 08854
mesterha@paul.rutgers.edu

June 10, 2001

Abstract

In this paper, 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. We will prove that, in the on-line mistake-bounded model of learning, these multi-class learning algorithms have the same mistake bounds as a related two class linear-threshold algorithm. We will also show how sub-experts can be used to solve more traditional problems composed of real valued attributes. This leads to a natural extension of the algorithm to multi-class problems that contain both traditional attributes and sub-experts.

1 Introduction

This paper deals with algorithms that classify a sequence of instances. Each instance belongs to one of k classes. After the algorithm predicts on an instance, the environment returns the correct classification. This information can then be used to improve the classifying 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 [Lit89].

In this paper, we define a new type of 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 all k output classes. The algorithm predicts the class that has the maximum value from a weighted sum of the sub-experts. We call this a linear-max function. The goal of the algorithm is to learn weights for the sub-experts that minimize the total number of prediction mistakes.

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 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 [LW94, CBFH$^+$97] as opposed to a combination of sub-experts.

The majority of this paper will focus on sub-experts. While learning with sub-experts is useful, many traditional learning problems are based on classifying an instance described with real valued attributes. In this paper, we will show how to transform real valued attributes into sub-experts. These new sub-experts can then be used to solve attribute based multi-class problems. In addition, these transformations naturally lead to algorithms that allow a combination of sub-experts and attributes.

Linear-max algorithms are related to linear-threshold functions and were motived by a desire to extend linear-threshold functions from two class prediction to multi-class prediction. Many people have worked on applying linear-threshold learning algorithms to multi-class problems. For example, [GR96, Blu95, Mes00] extend particular linear-threshold algorithms to multi-class problems. The main contribution of this paper is to give a general framework to extend a wide range of linear-threshold functions to multi-class problems. This includes the linear-threshold algorithms Perceptron [DH73] and the Weighted Majority Algorithm (WMA) [Lit89]. 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. For example, [Mes00] specifically generalizes WMA to handle multi-class problems. Using the techniques in this paper, we not only get the same algorithm but the same bound. On the more practical side, since linear-max problems are solved using linear-threshold algorithms, we should get similar real-world performance with both types of algorithms. Linear-threshold algorithms have been used successfully to learn both batch [DKR97, GR96] and on-line [Blu95] problems. Based on these examples, we expect linear-max algorithms to be successful in practice.

Here is an outline for the rest of the paper. In section 2, we define linear-max algorithms. This includes defining the sub-experts, the prediction scheme, and the update procedure. In section 3, we show how the mistake-bounds of a linear-max algorithm are inherited from a related linear-threshold algorithm. In section 4, we give some specific examples of linear-max algorithms. In section 5, we show how to use linear-max algorithms to solve problems that involve standard real valued attributes. In section 6, we give various useful extensions that can be made to the framework of linear-max algorithms. And last, in section 7, we give our conclusions with some ideas for future work.

2 Linear-max algorithms

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

2.1 Sub-experts

First we will 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 $\nu \in R$.[1]

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.

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 class predictions, they still may be useful in a linear-max function. For example, sub-experts might be used to add threshold weights. This can be 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 range of target functions.

2.3 Linear-threshold definition

We are interested in linear-threshold algorithms of the following form:

Initialization

$\mathbf{w} := \mathbf{w}_{init}$ with $\mathbf{w}_{init} \in R^n \cup \infty$.

[1]The values of the prediction differences can be generalized from an interval to certain types of sets. We do not explore this generalization in this paper.

```
**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 $\phi$ be the correct label, and $t$ the current trial.
        $\mathbf{w} := f(\mathbf{w}, (\phi\mathbf{z})_1, \ldots, (\phi\mathbf{z})_t)$.
```

For the linear-threshold function, the update is a function of the current weights and all the instances from previous trials (including the current instance) multiplied by their respective labels. We use the parenthesis notation in the update function to denote the fact that all variables within the parenthesis are from the trial referred to in the subscript, but for the linear-threshold algorithms we mention in this paper, only $\phi\mathbf{z}$ from the current trial will be used. The other trials could be useful in more complicated linear-threshold algorithms such as a support vector machine for sub-experts.

The input to the update function can cause a problem with algorithms that only make updates on mistakes. (These are often called mistake-driven algorithms[Lit95].) When $\phi \sum_{i=1}^{n} w_i z_i > 0$, on the current trial, the algorithm has made the correct prediction. When $\phi \sum_{i=1}^{n} w_i z_i < 0$, the algorithm has made a mistake. However when $\phi \sum_{i=1}^{n} w_i z_i = 0$ on the current trial, there is no way to tell, just from this information, whether the algorithm has made the correct or incorrect prediction. The update function in mistake-driven algorithms must be able to determine if a mistake occurred. To include these algorithms, we will modify any mistake-driven algorithm so that it makes updates when $\phi \sum_{i=1}^{n} w_i z_i \leq 0$. For all the mistake-driven algorithms we cover in this paper, this modification will not change the mistake bounds.

2.4 Linear-max 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 will use the update procedure from a linear-threshold algorithm to perform updates on a linear-max algorithm.

```
**Initialization**
    $\mathbf{w} := \mathbf{w}_{init}$ with $\mathbf{w}_{init} \in R^n$.
**Trials**
    **Instance:** Sub-experts $(x_1, \ldots, x_n)$.
    **Prediction:** Let $\zeta_{\mathbf{w}}(j) = \sum_{i=1}^{n} w_i x_i^j$. Pick two different
        classes $\lambda_1$ and $\lambda_2$ such that for all $j \neq \lambda_1 \neq \lambda_2$.
        $\zeta_{\mathbf{w}}(\lambda_1) \geq \zeta_{\mathbf{w}}(\lambda_2) \geq \zeta_{\mathbf{w}}(j)$. Predict class $\lambda_1$.
    **Update:** Let $\rho$ be the correct label.
        If $\rho = \lambda_1$ then $A = \lambda_2$ else $A = \lambda_1$.
        $\mathbf{w} := f(\mathbf{w}, (\mathbf{x}^\rho - \mathbf{x}^A)_1, \ldots, (\mathbf{x}^\rho - \mathbf{x}^A)_t)$.
```

While the update function is the same, the linear-max has different inputs for f. Instead of the instance times the label, the function depends on the sub-expert's prediction for the correct class minus the prediction for an incorrect class. There is some freedom in picking this incorrect class, but to keep things simple we will just cover one possibility. If the global prediction of the algorithm is wrong, we pick $A = \lambda_1$ to be the incorrect class. If the global prediction is correct then $\lambda_1 = \rho$, therefore we pick $A = \lambda_2$ to be the incorrect class. These input values are similar to the $\phi\mathbf{z}$ values used in the linear-threshold function. When the instance is correct

$\sum_{i=1}^{n} w_i(x_i^\rho - x_i^A)_t \geq 0$, and when the instance is wrong $\sum_{i=1}^{n} w_i(x_i^\rho - x_i^A)_t \leq 0$. We will prove the linear-max algorithm has the same bounds as the related linear-threshold algorithm by showing that the learning of the linear-max algorithm is equivalent to the linear-threshold algorithm learning with these new special instances.

3 Mistake bounds

In this section, we will prove that the linear-max algorithm has the same mistake bounds as its related linear-threshold algorithm. Throughout this section assume we are running both the linear-max and the linear-threshold algorithm.

Initialize both algorithms with the same weights. Assume we are given an instance of the linear-max problem. We will use the linear-max algorithm to transform these sub-experts into an instance for the linear-threshold algorithm. This instance will force the linear-threshold algorithm to make the same weight updates as the linear-max algorithm.

Use the linear-max algorithm to make a global prediction with the sub-experts. The environment will return the correct label ρ. Take the top two classes from the voting function. Formally this is λ_1 and λ_2 such that for all $j \neq \lambda_1 \neq \lambda_2$ $\zeta_{\mathbf{w}}(\lambda_1) \geq \zeta_{\mathbf{w}}(\lambda_2) \geq \zeta_{\mathbf{w}}(j)$. Create a new instance for the linear-threshold algorithm of the form $z_i = x_i^\rho - x_i^A$ where $A = \lambda_2$ if $\rho = \lambda_1$ and $A = \lambda_1$ if $\rho \neq \lambda_1$. The label for this instance is $\phi = 1$. Input $\mathbf{z}$ into the linear-threshold algorithm. Because of our earlier assumptions on sub-experts each $z_i \in [-\nu, \nu]$. 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 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 $\sum_{i=1}^{n} w_i x_i^\rho - \sum_{i=1}^{n} w_i x_i^{\lambda_1} \leq 0$. This can be rewritten as $\sum_{i=1}^{n} w_i(x_i^\rho - x_i^{\lambda_1}) \leq 0$. Because the linear-threshold algorithm uses the same weights, the instance $z_i = x_i^\rho - x_i^{\lambda_1}$ causes the linear-threshold algorithm to predict -1. Since we have constructed the problem such that the correct label for the linear-threshold algorithm is always $\phi = 1$, 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.[2] Using this upper-bound we can apply any bound from a linear-threshold algorithm to its related linear-max algorithm.

The essence of the linear-max algorithm can be seen in the above proof. We have constructed the linear-max algorithm in a way that mirrors the instance construction of the proof. We use a linear-threshold algorithm to solve a linear-max problem. We generate the instances using the prediction procedure of the linear-max algorithm. We then use the linear-threshold algorithm to learn on these instances. The linear-max algorithm performs well if the linear-threshold algorithm can learn a good classifier with these instances. If the linear-threshold algorithm needs to make any assumptions on the behavior of the instances then in order to get the mistake bound to carry over to the linear-max setting we need to ensure these same assumptions hold for the new transformed instances. For example, if the attributes, z_i need to have a certain

[2]On voting ties, it is possible that the linear-max algorithm makes the correct prediction while linear-threshold makes a mistake.

distribution, we would need to assure that any $x_i^{\rho} - x_i^{A}$ also has this distribution.[3] Fortunately many algorithms and proof techniques do not make such stringent assumptions on the instances. In this paper, we will give bounds using one of the more popular techniques for analyzing the performance of on-line linear-threshold algorithms [Lit89]. The only assumption needed for these results is that there exists a set of weights that perfectly classifies the data. The proof gives a finite upper-bound on the number of mistakes made by the algorithm. Crucial to the proof is the existence of a gap between the two classes where no instances occur.

More formally, let $\mathbf{u}$ be a vector of n weights that correctly classifies the linear-threshold concept. There may be many $\mathbf{u}$ vectors that correctly classify the data. For example, we can multiply $\mathbf{u}$ by any constant without changing its predictions. Because of this, we will normalize $\mathbf{u}$ so that $\sum_{i=1}^{n} u_i = 1$.[4] Now let $\mathbf{z}$ be the instance and ϕ be the correct label for trial t of the linear-threshold algorithm. Define the gap as $\delta_t = \phi \sum_{i=1}^{n} u_i z_i$. (Notice that δ_t is always positive since the weights correctly classify the instance.) Because the instances come from the linear-max algorithm, we substitute $z_i = x_i^{\rho} - x_i^{\lambda_1}$ and $\phi = 1$. This gives $\delta_t = \sum_{i=1}^{n} u_i x_i^{\rho} - \sum_{i=1}^{n} u_i x_i^{\lambda_1} = \zeta_{\mathbf{u}}(\rho) - \zeta_{\mathbf{u}}(\lambda_1)$. Since we do not know what class λ_1 the algorithm will predict in the future, we set $\hat{\delta_t} = \min_{j \neq \rho}(\zeta_{\mathbf{u}}(\rho) - \zeta_{\mathbf{u}}(j))$. This is the smallest delta that could occur on a mistake during trial t. We are interested in the minimum possible gap over all trials, so let $\delta = \min_t \hat{\delta_t}$. This δ parameter will have a strong influence on the bounds of the linear-max algorithms. As we shall see, a smaller δ corresponds to a harder problem.[5]

We should stress that the above assumptions only apply to the bounds of one particular proof technique. Any type of mistake bound for a linear-threshold algorithm can be applied to a similar linear-max algorithm. For example, in [Lit89] mistake bounds are given for linear-threshold algorithms that allow the algorithm to handle a finite number of noisy instances that are not correctly classified by the target vector $\mathbf{u}$. The algorithms are the same algorithms that apply to the no noise case; the only change is in the proof technique to show that the algorithms are robust to some noise. In [Lit91], bounds are given for a large class of algorithms, including certain linear-threshold algorithms, when learning with instances generated from a distribution. In this case, the bounds apply to problems with a potentially infinite number of noisy trials. These and any future results also apply to the related linear-max algorithms. However, to simplify our presentation, any bounds we give in the rest of the paper refer to the above no noise framework.

4 Specific linear-max algorithms

In this section, we will give some transformations of linear-threshold algorithms into linear-max algorithms. All the original linear-threshold algorithms are mistake-driven, so we have the made the modifications mentioned in section 2.3.

4.1 Perceptron algorithm

This is the multi-class version of the classic Perceptron algorithm [Ros62, MP69, DH73]. It fits directly into the linear-threshold framework needed for the linear-max transformation.

Initialization

[3] This may make it difficult to use the linear-max transformation for many Bayesian algorithms.

[4] Even after normalization there may still be different $\mathbf{u}$ vectors that correctly classify the data. In principle we can choose the one that minimzes the bound.

[5] In fact, most of the extra difficulty of multi-class problems is represented in the δ parameter. In realistic problems, as the number of classes increase the value of δ tends to go down making the problem more difficult.

```
    $\forall i \in \{1, \ldots, n\}\ w_i := 0.$
**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 $\lambda$ such that for all $j \neq \lambda$
        $\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(j)$.
    **Update:** Let $\rho$ 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 parmaeters[MP69]. 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 $s = \max_{\mathit{trials}} ||(\mathbf{x}^\rho - \mathbf{x}^\lambda)||_2$. Let $\mathbf{u}$ be a weight vector that correctly classifies all instances. And let $\delta = \min_{\mathit{trials}} (\min_{j \neq \rho}(\zeta_{\mathbf{u}}(\rho) - \zeta_{\mathbf{u}}(j)))$. With these parameters, the number of mistakes $\leq s^2 ||\mathbf{u}||_2^2 / \delta^2$.

4.2 WMA algorithm

This is the multi-class version of the WMA algorithm that learns linear threshold functions [Lit89]. The original linear-threshold algorithm deals with attributes $z_i \in [0,1]$. Using a linear transformation the attributes can be shifted to $[-1,1]$. In this form, the algorithm can be transformed into a linear-max algorithm. This algorithm is identical to the algorithm Committee previously reported in [Mes00].

```
**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 $\lambda$ such that for all $j \neq \lambda$
        $\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(j)$.
    **Update:** Let $\rho$ 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 WMA algorithm depend on certain problem dependent parameters[Lit89]. Let $\mathbf{u}$ be a weight vector that correctly classifies all instances. Let $\delta = \min_{\mathit{trials}} (\min_{j \neq \rho}(\zeta_{\mathbf{u}}(\rho) - \zeta_{\mathbf{u}}(j)))$ and $\alpha = (1-\delta)^{-1/2}$. The number of mistakes $\leq 2\ln(n)/\delta^2$. [Mes00] has a specific analysis that derives the same mistake bound for this linear-max algorithm.

Unlike the Perceptron algorithm, WMA cannot directly represent negative weights. A common solution in WMA is to add an extra modified copy of every attribute. In the above algorithm, for each sub-expert $\mathbf{x_i}$ add a new sub-expert $-\mathbf{x_i}$. (It can be verified that $-\mathbf{x_i}$ still satisfies the requirements for a sub-expert.) In effect, this allows the algorithm to learn sub-experts with negative weights. A target where expert $\mathbf{x_i}$ has weight $-.1$ is equivalent to a target where expert $-\mathbf{x_i}$ has weight .1 and expert $\mathbf{x_i}$ has weight 0. While these additions will double the number of sub-experts, because of the logarithmic nature of the bound, it will have a small effect.

4.3 Quasi-additive algorithms

The Quasi-additive algorithms are a recent and relatively unexplored class of linear-threshold learning algorithms [GLS97]. The behavior and proof of these algorithms depends on the function $f()$ satisfying certain conditions that are given in [GLS97]. In particular, certain choices result in an infinite range of algorithms that include and interpolate between WMA and Perceptron[GLS97]. Since the general framework of Quasi-additive linear-threshold algorithms fits directly into the assumptions of this paper, all Quasi-additive algorithms can be transformed into linear-max algorithms.

```
Initialization
    $\forall i \in \{1,\ldots,n\}\ v_i := 0$.
    Set $a > 0$ where $a \in R$.
    $\forall i \in \{1,\ldots,n\}\ \ w_i := f(v_i)$.
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 $\lambda$ such that for all $j \neq \lambda$
        $\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(j)$.
    Update: Let $\rho$ be the correct label. If $\zeta_{\mathbf{w}}(\lambda) \geq \zeta_{\mathbf{w}}(\rho)$
        $\forall i \in \{1,\ldots,n\}\ \ v_i := v_i + a(x_i^{\rho} - x_i^{\lambda})$.
        $\forall i \in \{1,\ldots,n\}\ \ w_i := f(v_i)$.
```

The bounds for a Quasi-additive algorithm depend upon the specific choice for the f function. Once this choice is made, the bound will depend on various parameters that describe the problem.

As can be seen above, Quasi-additive algorithms have a limited amount of state information; all information about previous trials is stored in the $\mathbf{v}$ vector. It is an interesting research question to determine how the performance of linear-threshold algorithms can be improved by storing more information from previous trials. Fortunately, because of the general update procedure allowed in this paper, a linear-threshold algorithm can store any amount of information about previous trials and still satisfy the linear-max requirements.

5 Attributes to sub-experts

Often there are no obvious sub-experts to use in solving a learning problem. Many times the only information available is a set of attributes. For attributes in $[0,1]$, we will show how to use a linear-max algorithm to represent a natural kind of k class target function, a linear machine. To learn this target, we will transform each attribute into k separate sub-experts.

5.1 Attribute target (linear machine)

A linear machine [DH73] is a prediction function that divides the feature space into disjoint convex regions where each class corresponds to one region. The predictions are made by comparing the value of k different linear functions where each function corresponds to a class.

More formally, assume there are $m-1$ attributes and k classes. Let $z_i \in [-\nu,\nu]$ be attribute i where $\nu \in R \cup \infty$. (The value of ν should be chosen to match the value used in the desired linear-max algorithm.) Assume the target function is represented using k linear functions of the attributes. Let $\psi_\mu(j)=\sum_{i=1}^{m} \mu_i^j z_i$ be the linear function for class j where μ_i^j is the weight of attribute i in class j. We have used the standard trick of adding one extra attribute, z_m, set to 1. This is needed for the constant portion of the linear functions. The target function labels an instance with the class of the largest ψ function. (Ties are not allowed.) Therefore, $\psi_\mu(j)$ is similar to $\zeta_{\mathbf{u}}(j)$, the target voting function for class j used in the linear-max algorithms. Just like the linear-max algorithm, we also want to normalize the target weights of the linear machine. Since only the relative difference between the ψ functions matter, we can divide all the ψ functions by any constant. We normalize the weights to sum to 1 so that $\sum_{i=1}^{n}\sum_{j=1}^{k} \mu_i^j = 1$. At this point, without loss of generality, assume that the original ψ functions are normalized.

5.2 Transforming the attributes

The transformation works as follows: convert attribute z_i into k sub-experts. Each sub-expert will always vote for one of the k classes with value z_i. The target weight for each of these sub-experts is the corresponding target weight of the attribute, class pair used in the ψ functions. Do this for every attribute.

$$z_i \Longrightarrow \mu_i^1 \begin{pmatrix} z_i \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \mu_i^2 \begin{pmatrix} 0 \\ z_i \\ 0 \\ \vdots \end{pmatrix} + \cdots + \mu_i^k \begin{pmatrix} 0 \\ \vdots \\ 0 \\ z_i \end{pmatrix}$$

The above target transformation creates mk sub-experts that vote with a ζ function that is the same as the ψ function of the linear machine. Therefore a linear machine target function can be represented as a special type of linear-max target function. This allows us to use the general bounds from linear-max algorithms on linear machines.

This transformation provides a simple procedure for solving linear machine problems. While the details of the transformation may look cumbersome, the actual implementation of the algorithm is relatively simple. There is no need to explicitly keep track of the sub-experts. Instead, the algorithm can use a linear machine type representation. Each class keeps a vector of weights, one weight for each attribute. The predictions are based on the linear functions that vote for each of the classes. During an update, only two sets of class weights are changed.

5.3 Combining attributes and sub-experts

These transformations give a straightforward way to combine sub-experts and attributes in a learning problem. Use the transformations to create new sub-experts from the attributes and combine them with the original sub-experts. It may even be desirable to break original sub-experts into attributes and use both in the algorithm because some sub-experts may perform better on certain classes. For example, if it is felt that a sub-expert is particularly good at class 1, we can perform the following transformation.

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \Longrightarrow \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \begin{pmatrix} x_1 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ x_1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ x_1 \end{pmatrix}$$

Now, instead of using one weight for the whole sub-expert, the linear-max algorithm can also learn based on the sub-expert's performance for the first class. In the same vein, it may be useful to add constant attributes to a set of sub-experts to act as a kind of multi-class threshold. These add only k extra sub-experts, but allow the algorithm to represent a larger set of target functions.

6 Extensions

In this section, we will show that the number of classes and the number of sub-experts do not need to be fixed at the start of the learning algorithm. In fact, we give extensions to linear-max algorithms that allow the possibility of an unlimited number of extra sub-experts and classes. This is similar to the infinite attribute model of [Blu92].

6.1 New sub-experts

First we will consider adding new sub-experts during the running of the algorithm. We can keep track of all new sub-experts by representing them with a single sub-expert at the start of the algorithm. Consider the group of all unseen sub-experts. If we can guarantee that this group of sub-experts always has the same weight then we only need to keep track of this single weight. Furthermore, if we guarantee these sub-experts have no effect on the global algorithm predictions, then we do not need to know how many sub-experts are in the group. (If they effected prediction, the effect would be proportional to the number of unseen experts, an unknown quantity.) Let $\mathbf{x}_\infty$ be the sub-expert that represents all the unseen sub-experts. When a sub-expert starts making its own predictions, we create a new sub-expert with weight w_∞ and add it to the group of normal sub-experts. This allows us to add a potentially infinite number of sub-experts.

To guarantee that the unseen sub-experts have no effect on predictions, we can stipulate that $\mathbf{x}_\infty$ must predict an equal amount on all classes. We can accomplish this by setting $x^j_\infty = 0$ for all j. To guarantee that the weights on all these sub-experts stays the same, we need to initialize all the unseen sub-experts with the same weight, and we need to add an extra assumption to the update function. The update function must make the same weight update on all unseen sub-experts. Since the update function comes from the linear-threshold algorithm, we will express the constraint in those terms. To keep the weights on the unseen sub-experts the same, it is necessary and sufficient to guarantee that for all attributes z_i and z_j, if, for all past trials, $z_i = z_j = 0$ then after the update $w_i = w_j$. All the linear-threshold algorithms in this paper satisfy this constraint.[6]

6.2 New classes

Next we consider adding new classes. The technique is similar to adding sub-experts. We want these unseen classes to have zero effect on prediction until the algorithm learns of them. Assume each sub-expert makes predictions on all the unseen classes. To make the presentation simpler, assume that each sub-expert makes the same prediction on all these unseen classes. (It is straightforward to extend this to different types of unseen classes with different predictions.) Let x_i^∞ be the prediction that sub-expert i makes for all unseen classes. By making these predictions low enough, we can guarantee that the unseen classes have no effect on the algorithm. This allows the algorithm to ignore the unseen classes until they are discovered. At that point the sub-experts can start making useful predictions on the new classes. To guarantee that the

[6]Useful updates functions that might not satisfy this constraint are randomized updates.

unseen classes don't effect the algorithm, we must guarantee they don't effect the global prediction. We accomplish this by forcing the unseen classes to always lose the global prediction vote. There are many ways to do this.

An easy way is to let all sub-experts give the same prediction to the unseen classes. Therefore we set $\mathbf{x}^{\infty}$ to a constant vector. This value can be chosen to give the unseen classes any value for ζ_w, the voting function. Have the algorithm calculate a value that assures that the unseen classes will lose the vote. We still have some flexibility in assigning a value to $\mathbf{x}^{\infty}$. This may be useful since this value can affect the updates. When a new class is returned as the correct label of an instance, we need to update the sub-expert weights based on $\mathbf{x}^{\infty} - \mathbf{x}^{\lambda}$. Therefore the value of $\mathbf{x}^{\infty}$ will have an effect on the new weights. The particular choice of $\mathbf{x}^{\infty}$ will most likely depend on details of the problem. After the update is completed, all sub-experts should be updated about the existence of a new class. Future predictions of sub-experts will include a prediction for this class.

6.3 Noise

One problem with both of these runtime algorithm modifications is that they make it impossible for a perfect classifier to exist at the start of the algorithm. If a perfect classifier exists with the starting set of sub-experts and classes, we have no need to add more sub-experts or classes. As we have seen, the standard mistake-bound results require a perfect classifier. A partial solution to this problem is to use the more robust mistake-bounds that allow some types of noise[Lit89, Lit91]. These noise models can give the algorithm some trials before a perfect classifier exists or even give bounds for situations where a perfect classifier never exists. A more satisfying solution is to devise new noise models and proofs that take into account the problems of a concept that gradually becomes representable.[7] This is an area we plan to pursue in the future.

7 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 main benefit of this transformation is that it preserves the on-line mistake-bound results for various linear-threshold algorithms to the linear-max setting. Linear-max algorithms learn target functions that are composed of sub-experts that make predictions on all classes. While this may seem restrictive, we have shown how to use these sub-experts to solve more traditional problems composed of attributes. This will allow linear-max algorithms to solve a wide range of multi-class problems including hybrid problems that are composed of a mix of attributes and sub-experts.

In the future, we plan on applying these results to Apobayesian algorithms[LM97] such as SASB. This is an interesting class of algorithms that contains useful linear-threshold functions. In particular, SASB is an algorithm related to WMA that shares many of its benefits for learning with few relevant attributes but does not require any problem specific parameters to set in order to guarantee good performance. Currently, the only thing preventing the transformation of SASB to a linear-max algorithm is that it only allows binary attributes. We are currently working on modifying SASB to allow attributes in an interval.[8]

[7]This also includes the problem of sub-experts that change over time by learning about the target function.

[8]Using a generalized form of the results in this paper, a limited type of sub-expert can be used with attributes that have at least three values.

Acknowledgements

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

References

[Blu92] Avrim Blum. Learning boolean functions in an infinite attribute space. *Journal of Machine Learning*, 9(4):373–386, 1992.

[Blu95] Avrim Blum. Empirical support for winnow and weighted-majority algorithms: results on a calendar scheduling domain. In *ML-95*, pages 64–72, 1995.

[CBFH$^+$97] Nicolò Cesa-Bianchi, Yoav Freund, David Haussler, David P. Helmbold, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. *Journal of the Association for Computing Machinery*, 44(3):427–485, 1997.

[DH73] R. O. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.

[DKR97] I. Dagan, Y. Karov, and D. Roth. Mistake-driven learning in text categorization. In *EMNLP-97*, pages 55–63, 1997.

[GLS97] Adam J. Grove, Nick Littlestone, and Dale Schuurmans. General convergence results for linear discriminant updates. In *COLT-97*, pages 171–183, 1997.

[GR96] A. R. Golding and D. Roth. Applying winnow to context-sensitive spelling correction. In *ML-96*, 1996.

[Lit89] Nick Littlestone. *Mistake bounds and linear-threshold learning algorithms*. PhD thesis, University of California, Santa Cruz, 1989. Technical Report UCSC-CRL-89-11.

[Lit91] Nick Littlestone. Redundant noisy attributes, attribute errors, and linear-threshold learning using winnow. In *COLT-91*, pages 147–156, 1991.

[Lit95] Nick Littlestone. Comparing several linear-threshold learning algorithms on tasks involving superfluous attributes. In *ML-95*, pages 353–361, 1995.

[LM97] Nick Littlestone and Chris Mesterharm. An apobayesian relative of winnow. In *NIPS-9*, pages 204–210. MIT Press, 1997.

[LW94] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108:212–261, 1994.

[Mes00] Chris Mesterharm. A multi-class linear learning algorithm related to winnow. In *NIPS-12*, pages 519–525. MIT Press, 2000.

[MP69] Marvin L. Minsky and Simon A. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[Ros62] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington, DC, 1962.