Algebraic classifiers: a generic approach to fast cross-validation, online training, and parallel training

Michael Izbicki MIKE@IZBICKI.ME
UC Riverside, 900 University Ave., Riverside, CA 92521

Abstract

We use abstract algebra to derive new algorithms for fast cross-validation, online learning, and parallel learning. To use these algorithms on a classification model, we must show that the model has appropriate algebraic structure. It is easy to give algebraic structure to some models, and we do this explicitly for Bayesian classifiers and a novel variation of decision stumps called HomStumps. But not all classifiers have an obvious structure, so we introduce the Free HomTrainer. This can be used to give a "generic" algebraic structure to any classifier. We use the Free HomTrainer to give algebraic structure to bagging and boosting. In so doing, we derive novel online and parallel algorithms, and present the first fast cross-validation schemes for these classifiers.

1. Introduction

Abstract algebra is an increasingly popular tool for computer scientists. In machine learning, algebra provides an alternative approach to difficult problems involving harmonic analysis and the Fourier transform (Kondor et al., 2007; Kondor & Borgwardt, 2008; Pachauri et al., 2012). In statistics, it facilitates experimental design (Watanabe, 2009). And in cryptography, algebraic methods allow users to perform calculations on encrypted data without knowing the encryption key (Vaikuntanathan, 2011). But the main inspiration for this paper is the work by functional programmers. Their use of algebra promotes reuse of high level ideas and code in the most generic way possible (Yorgey, 2012). In this paper, we derive algebraic procedures for three common machine learning tasks. The algebraic formulation makes these procedures widely applicable to many learning models.

Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28. Copyright 2013 by the author(s).

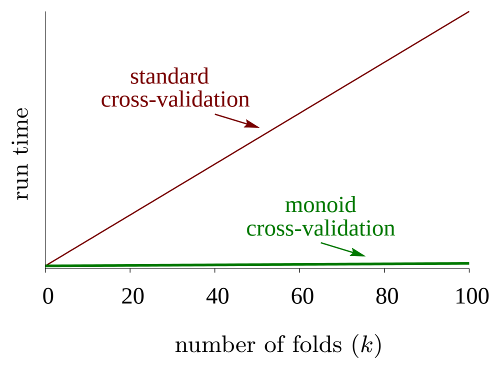

Figure 1. The monoid cross-validation algorithm runs asymptotically faster than the standard version. Run time is essentially independent of the number of folds.

First, *cross-validation* lets us find the optimal parameters for our model and estimate its generalization capability. Typically (although not always) increasing the number of folds makes for a better generalization estimate. On large data sets, however, doing many folds is expensive. To alleviate this probelem, a number of fast cross-validation algorithms have been developed for specific models (Arlot & Celisse, 2010). We present a new version of cross-validation that is more generic. It takes advantage of an algebraic structure called a monoid (defined in Section 2). Monoid cross-validation scales better than standard cross-validation, as shown in Figure 1. Furthermore, it is not an approximation—it gives the exact same results as standard cross-validation.

Second, *online training algorithms* let us process streams of data. In general, online training is much harder than batch training, and there is a large body of work exploring this difficulty (Littlestone, 1989; Bendavid et al., 1997; Kakade & Kalai, 2005; Dekel, 2008). Our contribution is to show that if a model has monoid structure, then it has a simple online training algorithm. Furthermore, the online and batch algorithms both produce exactly the same results.

Third, *parallel training algorithms* let us speed up batch training by simply adding more processors. We can scale the number of processors with the size of our data set to keep our training times manageable, as shown in Figure 2. Monoid structures are a simple abstraction for parallel computing. All computations in the "parallel complexity class" $\mathbb{NC}^1$ can be expressed as a computation over a monoid (Tesson & Thrien, 2004). For example, in Google's MapReduce framework, the reduce step is a monoid computation (Dean & Ghemawat, 2004). Many popular learning algorithms[1] have been parallelized in this way (Chu et al., 2006; Palit & Reddy, 2011). These authors gave learning algorithms an implicit monoid structure in order to parallelize them. By making this structure explicit, we can also use our online and fast cross-validation methods on these learning models.

The rest of this paper is organized as follows: Section 2 defines our basic notation and algebraic terms. Section 3 presents our fast cross-validation algorithms, and our online and parallel training algorithms. Section 4 shows that Bayesian classifiers and a new type of decision stump called HomStumps have the needed algebraic structure. Finally, section 5 presents a technique for giving algebraic structure to any learning model. In each section, we theoretically and empirical justify our claims as appropriate.

2. Notation

We focus on the supervised learning problem. The data points have type $\mathcal{L} \times \mathcal{A}$, where $\mathcal{L}$ is the type of class labels, $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2 \times ... \times \mathcal{A}_t$ is the type of the attributes, and t is the number of attributes. We call the space of our data sets $\mathcal{D}$, and every individual data set $d \in \mathcal{D}$ contains points drawn from $\mathcal{L} \times \mathcal{A}$. Every classifier has a space of possible models $\mathcal{M}$, a batch training function $T : \mathcal{D} \to \mathcal{M}$, and a classification function $C : \mathcal{M} \times \mathcal{A} \to \mathcal{L}$.

In this paper, we study classifiers whose model space $\mathcal{M}$ forms a monoid or a group and whose batch trainer T is a homomorphism. These are fundamental concepts in algebra, and we briefly define them here.

[1]Chu *et. al.* adapted these algorithms to MapReduce: locally weighted linear regression, naive bayes, gaussian discriminative analysis, k-means, logistic regression, neural networks, principal component analysis, independent component analysis, expectation maximization, and support vector machines. Palit and Reddy adopted AdaBoost.

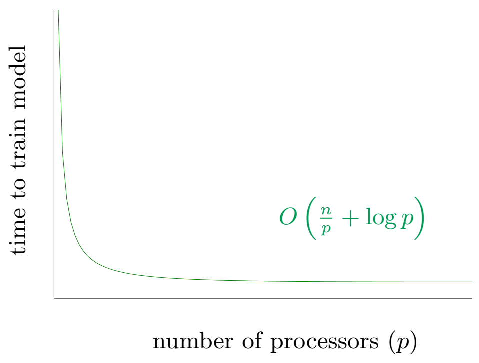

Figure 2. We can parallelize any batch trainer if the model is a monoid. Most of the potential speed up can be realized with only a small number of processors.

Definition 1. A **monoid** consists of a set $\mathcal{M}$, an associative binary operation $\diamond : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, and a special identity element ϵ such that for all $m \in \mathcal{M}$,

$$\epsilon \diamond m = m \diamond \epsilon = m$$

For example, our type of data sets $\mathcal{D}$ forms a monoid with concatenation $(+\!\!+)$ as the binary operation and the empty data set as ϵ. In this paper, we will use $(+\!\!+)$ to denote the monoid operation on data sets, and $(\diamond)$ for our classification models.

Definition 2. We call monoid $\mathcal{M}$ a **group** if every element m in $\mathcal{M}$ has a unique inverse m^{-1} such that:

$$m \diamond m^{-1} = m^{-1} \diamond m = \epsilon$$

We call the group **Abelian** if the binary operation is also commutative.

Definition 3. If the model $\mathcal{M}$ we are trying to learn forms a monoid, then our batch trainer $T : \mathcal{D} \to \mathcal{M}$ is a **homomorphism** if for all data sets $d_1, d_2 \in \mathcal{D}$:

$$T(d_1 +\!\!+ d_2) = T(d_1) \diamond T(d_2)$$

Homomorphisms are a generalization of linear functions—both are particularly nice to work with.

3. New Algorithms

In this section, we show how to use the algebraic properties of a model to derive a fast cross-validation algorithm, an online trainer, and a parallel batch trainer. To simplify run time analysis, we make three assumptions: (i) the batch trainer runs in time $O(n)$, where n is the number of elements in the data set; (ii) the monoid operation runs in constant time; (iii) classifying a data point takes constant time. These assumptions are not strictly necessary, but they hold for the examples in section 4 and the Free HomTrainer presented in section 5.

Table 1. Run times for cross-validation algorithms.

method	k-fold	leave-one-out ($k = n$)
standard	$O(kn)$	$O(n^2)$
monoid	$O(k+n)$	$O(n)$

3.1. Fast Cross-Validation

We can do fast cross-validation if our model forms a monoid. In standard k-fold cross-validation, we have a loop that iterates k times. Inside the loop we split the data set into a training set and a testing set. We train a model $m \in \mathcal{M}$ on the training set, and measure m's performance on the testing set. Because training a model takes time $O(n)$, the whole procedure takes time $O(kn)$.

In the standard procedure, each data point will be in the training set $k-1$ times. This results in repeated work as we train different models on this same data point. Monoid cross-validation avoids repeating this work using the well known prefix-sum procedure. Pseudocode is shown in Algorithm 1. We begin in the training loop by splitting the data set into k subsets and train a model m_i on each subset d_i. In the prefix loop we calculate p_i, which is a model trained on all of the data points in subsets $i-1$ and smaller. In the suffix loop we calculate s_i, which is a model trained on all of the data points in subsets $i+1$ and larger. Finally in the testing loop, we merge the appropriate prefix and suffix together. This merge operation gives a model m that has been trained on every data point except those in the test set d_i.

Our first loop has k iterations, and in each iteration we train a model on $\frac{n}{k}$ data points. Training takes time $O\left(\frac{n}{k}\right)$, so the loop takes time $O(n)$. The other three loops also have k iterations, but each iteration takes constant time. The overall run time is therefore $O(k+n)$.

3.2. Online Algorithms

We can construct an online trainer for any monoid learner. An online trainer $T^O : \mathcal{M} \times \mathcal{D} \to \mathcal{M}$ is a function that lets us add new data points to an already constructed model. If our model is a monoid, we can construct the online training function as

$$T^O(m, d) = m \diamond T(d)$$

where $m \in \mathcal{M}$ is the previously trained model, and $d \in \mathcal{D}$ is the set of data points we want to add. Often this set will contain only a single point, but it can be arbitrarily large.

Algorithm 1 Monoid cross-validation

```
**Input:** Batch trainer $T : \mathcal{D} \to \mathcal{M}$, data set $d \in \mathcal{D}$,
metric $P : \mathcal{M} \times \mathcal{D} \to \mathbb{R}$, number of folds $k$
**Output:** Mean ($\mu$) and variance ($\sigma^2$) of scores
// training loop
for $i \leftarrow 1..k$ do
  Let $d_i$ = select $i$th $\frac{n}{k}$ data points from $d$
  Let $m_i = T(d_i)$
end for
// prefix loop
Let $p_0 = \epsilon$
for $i \leftarrow 1..k$ do
  Let $p_i = m_i \diamond p_{i-1}$
end for
// suffix loop
Let $s_{k+1} = \epsilon$
for $i \leftarrow k..1$ do
  Let $s_i = m_i \diamond s_{i+1}$
end for
// testing loop
for $i \leftarrow 1..k$ do
  Let $m = p_{i-1} \diamond s_{i+1}$
  Update $\mu, \sigma^2$ with $P(m, d_i)$
end for
Return $(\mu, \sigma^2)$
```

This construction is exact. That is, we get the same model whether we train our model in online mode or batch mode. This exactness is a straightforward consequence of the definition of a homomorphism: We know that our model m was trained from some data set $d_m \in \mathcal{D}$; so $m = T(d_m)$. By substitution, we get that

$$T^O(m, d) = T(d_m) \diamond T(d) = T(d_m +\!\!+ d)$$

which is the model generated by the batch trainer.

3.3. Parallel Algorithms

We can train a model in parallel if it forms a monoid. Google's MapReduce is a popular platform for achieving this parallelism (Dean & Ghemawat, 2004), and our procedure is similarly divided into *map* and *reduce* steps. In the map step, we split the data set into p equal subsets $d_1 ... d_p$, where p is the number of processors. Each processor i trains a model $m_i = T(d_i)$. This takes parallel time $O(\frac{n}{p})$. In the reduce step, we use the monoid operation to combine the resulting models with a fan-in reduction (Parhami, 1999). If the monoid operation takes constant time, the reduction takes parallel time $O(\log p)$. The overall run time is plotted in Figure 2.

Because our training function is a monoid homomorphism, we have that

$$T(d_1) \diamond T(d_2) \diamond ... \diamond T(d_p) = T(d_1 + \!\!\!+ \, d_2 + \!\!\!+ \, ... + \!\!\!+ \, d_p)$$

Less formally, the model we get from training in parallel is the same exact model we would get from training with a single machine.

4. Examples

In this section, we show two examples of models with Abelian group structure. Abelian groups are a type of monoid, so this lets us apply the techniques above to create online, parallel, and fast cross-validation algorithms for these models. In the first example, we present a straightforward extension to the Bayesian classifier. In the second example, we derive algebraic structure for a new variant of decision stumps called HomStumps.

4.1. Bayesian Classification

Bayesian classifiers use Bayes theorem to calculate $P(L|A)$, where L is a random variable for the labels and A is a random variable for the attributes. To classify, we select the label with the highest probability. Formally,

$$C_{Bayes}(a) = \arg\max_{l \in \mathcal{L}} P(L = l)P(A = a|L = l)$$

The model for the Bayesian classifier is

$$\mathcal{M}_{Bayes} = (P(L), P(A|L))$$

where $P(L)$ and $P(A|L)$ are estimated from the data. Notice that this is an equation at the type level, not the value level. It says that any model $m \in \mathcal{M}_{Bayes}$ must have the structure of an ordered pair of two "base" probability distributions.

If these base distributions have Abelian group structure, then the Bayesian classifier will as well.[2] The Bayesian model's binary operation is defined as:

$$(P_a(L), P_a(A|L)) \diamond (P_b(L), P_b(A|L)) \\ = (P_a(L) \diamond P_b(L), P_a(A|L) \diamond P_b(A|L))$$

The empty element is:

$$\epsilon_{Bayes} = (\epsilon_{P(L)}, \epsilon_{P(A|L)})$$

And the inverse is:

$$(P(L), P(A|L))^{-1} = (P(L)^{-1}, P(A|L)^{-1})$$

[2]The method of moments is a simple way to give this structure to a continuous probability distribution. The supplemental material reviews how.

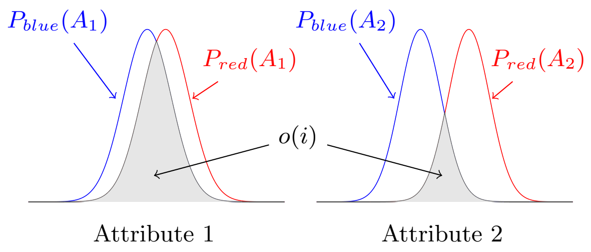

Figure 4. We plot $P_l(A_i)$ for two attributes from a classification problem with blue and red class labels. The HomStump algorithm would split on attribute 2 because $o(2) < o(1)$.

If the Abelian group laws hold for the underlying distribution, then they hold for the Bayesian classifier as well. Finally, note that we have not made the independence assumption of naive Bayes, so $P(A|L)$ can be a multivariate distribution.

4.1.1. Discussion

We do not test this method empirically because the online and parallel training algorithms will generate exactly the same classifier as the standard batch algorithm. Additionally, our fast cross-validation algorithm generates the same results as standard cross-validation.

4.2. Homomorphic Decision Stumps

A decision stump is a decision tree with a single branch (Iba & Langley, 1992). It is rarely a good classifier by itself, but it makes a good base classifier for ensemble algorithms. For example, the Viola-Jones face detection algorithm uses boosted decision stumps because they are faster than decision trees and have similar classification accuracy (Viola & Jones, 2004). In this section, we introduce a homomorphic variant of decision stumps called HomStumps and test them empirically. In section 5.2 we show how to use these HomStumps as a base classifier for homomorphic boosting.

Standard decision stumps use information gain to determine which attribute to split on, and where in the attribute's range to split. Unfortunately, it is difficult to define an algebraic structure for this splitting criterion. HomStumps use a different criterion. Instead of calculating the split point directly from the input data, we first estimate a distribution of the inputs. In particular, for every attribute A_i, we estimate the probability distribution for each label $l \in \mathcal{L}$. We call this distribution $P_l(A_i)$. If the set of distributions for a given attribute i has a lot of overlap, then attribute i carries little predictive information and is a

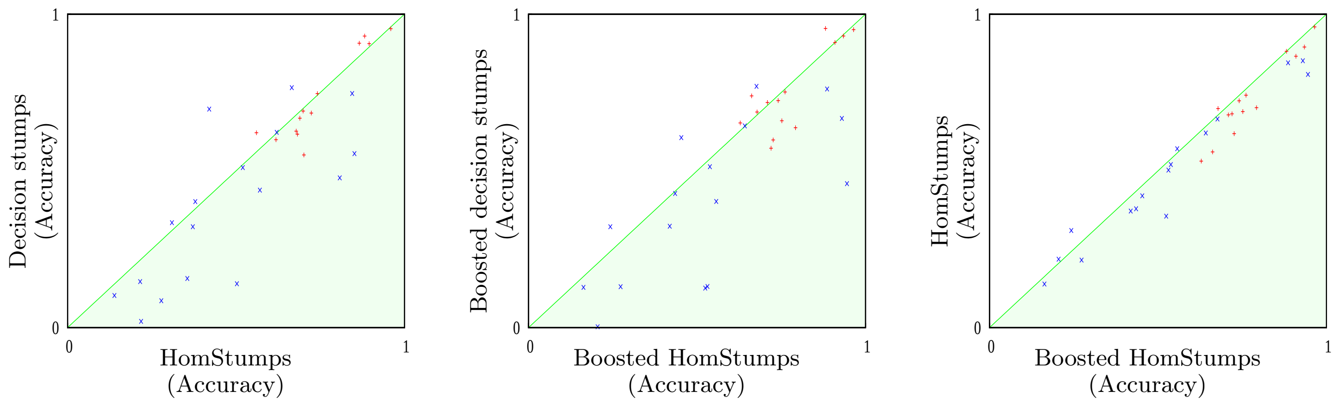

Figure 3. We empirically evaluate the HomStump's classification accuracy. We tested it on thirty binary (+) and multiclass (×) data sets. (*left*) HomStumps classify slightly better than standard decision stumps on most binary data sets and much better on multiclass data sets. (*center*) HomStumps still outperform standard decision stumps when boosted by AdaBoost. (*right*) Boosting almost always improves HomStump's accuracy. The supplemental material contains further details about this experiment.

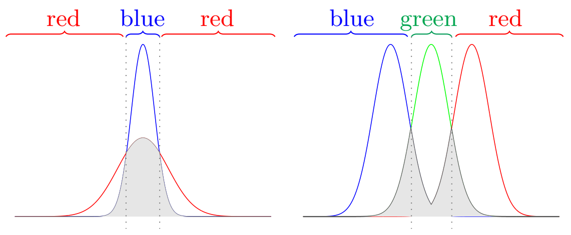

Figure 5. HomStumps outperform standard decision stumps when they create multiple decision boundaries. This can happen in two ways: (*left*) The densities cross at multiple points. (*right*) There are more than two classes in the dataset.

bad candidate for splitting. If the distributions have little overlap, then attribute i is a good predictor and a good candidate for splitting. This is shown graphically in Figure 4.

For a discrete distribution we calculate the overlap $o(i)$ on attribute i as

$$o(i) = \sum_{a \in \mathcal{A}_i} P_{l_2}(a)$$

where l_2 is the label with the second highest density when $A_i = a$. For a continuous distribution, this generalizes to

$$o(i) = \int_{-\infty}^{\infty} P_{l_2}(a)\,\mathrm{d}a$$

We use this overlap to calculate the split point s:

$$s = \operatorname*{arg\,min}_{i \in \{1..t\}} o(i)$$

Now, we can classify a data point with a modified Bayesian technique that only considers the attribute we chose to split on. Our classification function is:

$$\begin{aligned} &C_{HomStump}(\{a_1, a_2, ..., a_s, ...a_k\}) \\ &\qquad = \operatorname*{arg\,min}_{l \in \mathcal{L}} P(L = l) P(A_s = a_s | L = l) \end{aligned}$$

This uses exactly the same information as the Bayesian classifier, just in a different way. This modification makes the HomStump an "unstable classifier." That is, a single input is likely to have a large change in classification results if it changes the splitting attribute s. Unstable classifiers are particularly useful in ensemble methods, which we discuss in section 5.2.

Finally, showing an algebraic structure for the HomStump model is easy. The model itself is the same as the model for a Bayesian classifier:

$$\mathcal{M}_{HomStump} = (P(L), P(L|A))$$

Therefore, HomStumps have the same monoid structure.

4.2.1. Discussion

It was hard to give the standard decision stumps algebraic structure. We therefore created a variant of HomStumps based on a classifier that does have algebraic structure. This technique can potentially be used to give algebraic structure to other models as well. These variants can have similar or better performance to the classifiers that inspire them. For example, HomStumps tend to outperform standard decision stumps in practice. As Figure 5 shows, HomStumps naturally split an attribute in multiple locations when this is appropriate. Traditional decision stumps only split once for ease of computation. In practice, this greatly improves the HomStump's classification accuracy. Figure 3 shows HomStumps outperforming decision stumps both as stand alone classifiers and as the base model to AdaBoost.

5. Homomorphisms for Any Classifier

In this section we present the Free HomTrainer. This technique is a generalization of bagging, and is useful for discovering algebraic structure in a base model. We will use it to develop a variant of AdaBoost with Abelian group structure.

5.1. Training and Classification

The Free HomTrainer relies on "free objects" from the branch of mathematics called category theory.[3] Free objects have two important properties for our purposes. First, they are the most generic way to give algebraic structure to a set. We will use this fact to imbue a model space $\mathcal{M}$ with Abelian group structure. The resulting structure is denoted $\mathcal{F}(\mathcal{M})$. This free Abalian group has the structure:

$$\mathcal{F}(\mathcal{M}) = \{(\mathbb{Z}, \mathcal{M})\}$$

This is a type level equation. It says that an element $m \in \mathcal{F}(\mathcal{M})$ will be a set of pairs of base models (having type $\mathcal{M}$) and the (possibly negative) number of times that model appears.

The second important property of free objects is that if we have a model $\mathcal{M}$with training function $T : \mathcal{D} \to \mathcal{M}$ then there is a natural way to construct the function $\mathcal{F}(T) : \mathcal{F}(\mathcal{D}) \to \mathcal{F}(\mathcal{M})$. This natural construction for $\mathcal{F}(T)$ is guaranteed to be a homomorphism. Therefore, if we select $\mathcal{F}(\mathcal{M})$ to be our model, and $\mathcal{F}(T)$ to be our training function, we satisfy the algebraic conditions necessary for the parallel, online, and fast cross-validation algorithms of Section 3. Unfortunately, we cannot use the function $\mathcal{F}(T)$ directly. We must first convert our data set into a free data set using a function $\delta : \mathcal{D} \to \mathcal{F}(\mathcal{D})$. After applying this function to our data set, we can train $\mathcal{F}(\mathcal{M})$ like normal.

For Abelian groups, this is a simple procedure. The function δ must somehow split our data set d into a set of multiple smaller data sets $\{d_1, ..., d_k\}$. Each of these smaller data *sets* is a data *point* for $\mathcal{F}(\mathcal{M})$. Our modified training function is:

$$\mathcal{F}(T) = \{(1, d_1),(1, d_2), ..., (1, d_k)\} \mapsto \{(1, T(d_0)), (1, T(d_1)), ..., (1, T(d_k))\}$$

To classify, we would like to provide another function $\mu : \mathcal{F}(\mathcal{M}) \to \mathcal{M}$, so that the following diagram commutes:

$$\begin{tikzcd}
\mathcal{D} \arrow[r, "T"] \arrow[d, "\delta"'] & \mathcal{M} \\
\mathcal{F}(\mathcal{D}) \arrow[r, "\mathcal{F}(T)"] & \mathcal{F}(\mathcal{M}) \arrow[u, "\mu"']
\end{tikzcd}$$

In other words, we want $T = \mu \cdot \mathcal{F}(T) \cdot \delta$. This would let us construct the free model's classification function $C_{\mathcal{F}(\mathcal{M})} : \mathcal{F}(\mathcal{M}) \times \mathcal{A} \to \mathcal{L}$ using the base model's classification function $C_\mathcal{M} : \mathcal{M} \times \mathcal{A} \to \mathcal{L}$ as:

$$C_{\mathcal{F}(\mathcal{M})}(m, a) = C_\mathcal{M}(\mu(m), a)$$

It is easy to make the diagram commute if our base model $\mathcal{M}$ already has an Abelian group structure. We take:

$$\delta = \{x_1, x_2, ..., x_n\} \mapsto \{\{(1, x_1)\}, \{(1, x_2)\}, ..., \{(1, x_n)\}\}$$
$$\mu = \{(1, m_1), (1, m_2), ..., (1, m_k)\} \mapsto m_1 \diamond m_2 \diamond ... \diamond m_k$$

In general, however, it is impossible to construct this μ without any knowledge of the underlying model $\mathcal{M}$. A suitable map is not even guaranteed to exist.

Fortunately, we can take advantage of the classifier structure of $\mathcal{M}$. That is, if we fix the data point a that we are trying to classify, then there exists a function $C_a : \mathcal{M} \to \mathcal{L}$. We can use this map to extend the commutative diagram like so:

$$\begin{tikzcd}
\mathcal{D} \arrow[r, "T"] \arrow[d, "\delta"'] & \mathcal{M} \arrow[r, "C_a"] & \mathcal{L} \\
\mathcal{F}(\mathcal{D}) \arrow[r, "\mathcal{F}(T)"] & \mathcal{F}(\mathcal{M}) \arrow[r, "\mathcal{F}(C_a)"] & \mathcal{F}(\mathcal{L}) \arrow[u, "\lambda"']
\end{tikzcd}$$

Our newly introduced function $\lambda : \mathcal{L} \to \mathcal{F}(\mathcal{L})$ is independent of the structure of the model. Unfortunately, it is not possible to select λ so that the diagram commutes; however, we *can* select λ so that the diagram *almost* commutes. One easy way to do this is by letting λ take the majority vote.

[3]Category theory is an extremely abstract branch of mathematics. It has been called both the "unifying theory of mathematics" and "general abstract nonsense."

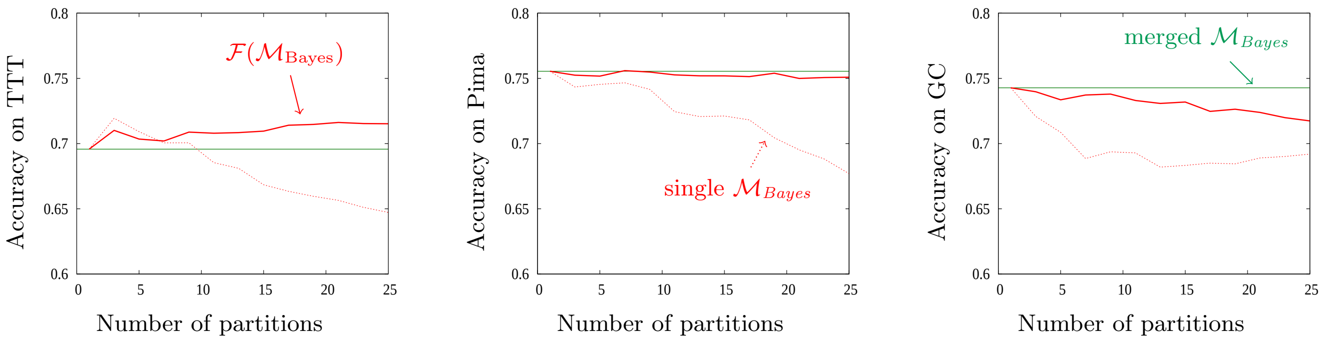

Figure 6. As in the bagging classifier, the accuracy of the Free HomTrainer depends on the combined characteristics of our base model, data set, and the number of partitions created by δ. (*left*) On the Tic-Tac-Toe data set, the Bayesian classifier over fits the data; therefore, $\mathcal{F}(\mathcal{M}_{\text{Bayes}})$ is more accurate. (*center*) On the Pima Indian Diabetes data set, the Bayesian classifier and $\mathcal{F}(\mathcal{M}_{\text{Bayes}})$ have similar classification accuracies. (*right*) On the German Credit data set, the Bayesian classifier outperforms $\mathcal{F}(\mathcal{M}_{\text{Bayes}})$. (*all data sets*) If we train a Bayesian classifier on only a single partition, classification accuracy falls as the number of data points decreases. If we train a Bayesian classifier on each partition and merge them with the monoid operation, then performance is constant because the batch trainer is a homomorphism.

5.1.1. Discussion

Figure 6 empirically demonstrates that the Free HomTrainer $\mathcal{F}(\mathcal{M})$ classifies similarly to the base model $\mathcal{M}$ for at least some model/data set combinations. In particular, if the base model is stable and doesn't over fit the data set, the Free HomTrainer will be a good approximation. This is a well known property of the bagging classifier (Breiman, 1996), which supports the notion that the Free HomTrainer is a generalization of the bagging classifier to arbitrary algebraic structures.

The real usefulness of this categorical construction is that it will help us derive specialized homomorphic learning algorithms for other classifiers. We now consider this procedure for boosting.

5.2. Specializing to Boosting

AdaBoost is a popular and highly effective classification algorithm (Freund & Schapire, 1996), but finding an algebraic structure for it is nontrivial. It has a number of published variants for online (Oza, 2001; Grabner et al., 2008) and parallel training (Yu & b. Skillicorn, 2001; Merler et al., 2007; Chen et al., 2008; Palit & Reddy, 2011), but no previously published method for fast cross-validation. We derive novel variants of these algorithms by applying the Free HomTrainer. In particular, we show three definitions of μ that take advantage of the boosting model's structure in different ways.

Boosting is an iterative procedure. At each iteration i, a base model $m_i \in \mathcal{M}_{Base}$ is generated and assigned a weight $w_i \in \mathbb{R}$. The model for a boosting algorithm is a set of these weighted base models:

$$\mathcal{M}_{Boost} = \{(\mathbb{R}, \mathcal{M}_{Base})\}$$

Classification is the weighted vote of the base models in the set. For boosting, our μ function has type $\mu : \mathcal{F}(\mathcal{M}_{\text{Boost}}) \to \mathcal{M}_{Boost}$. We will compare three alternative definitions for μ that take advantage of the boosting model's internal structure. Their empirical performance is discussed in Figure 7.

5.2.1. Concatenation

Our first candidate μ is the simplest. Observe that since $\mathcal{M}_{Boost}$ is a set of other models, $\mathcal{F}(\mathcal{M}_{\text{Boost}})$ is a set of a set of models:

$$\mathcal{F}(\mathcal{M}_{\text{Boost}}) = \{(\mathbb{Z}, \mathcal{M}_{Boost})\} = \{(\mathbb{Z}, \{(\mathbb{R}, \mathcal{M}_{Base})\})\}$$

Classification on $\mathcal{F}(\mathcal{M}_{\text{Boost}})$ happens as in a representative democracy—each AdaBoost model uses its base classifiers to elect a label; then these labels vote in the λ function to determine the final classification. The concatenation method "flattens" this representative democracy into a direct democracy. It uses the mapping:

$$\{(c_i, \{(w_{ij}, m_{ij})\})\} \mapsto \{(c_i w_{ij}, m_{ij})\}$$

where i ranges over each element in the free HomTrainer, and j ranges over each element in the boosting model.

5.2.2. Sorting+Voting

We can derive a slightly more complicated method from the work of Palit and Reddy (2011). They

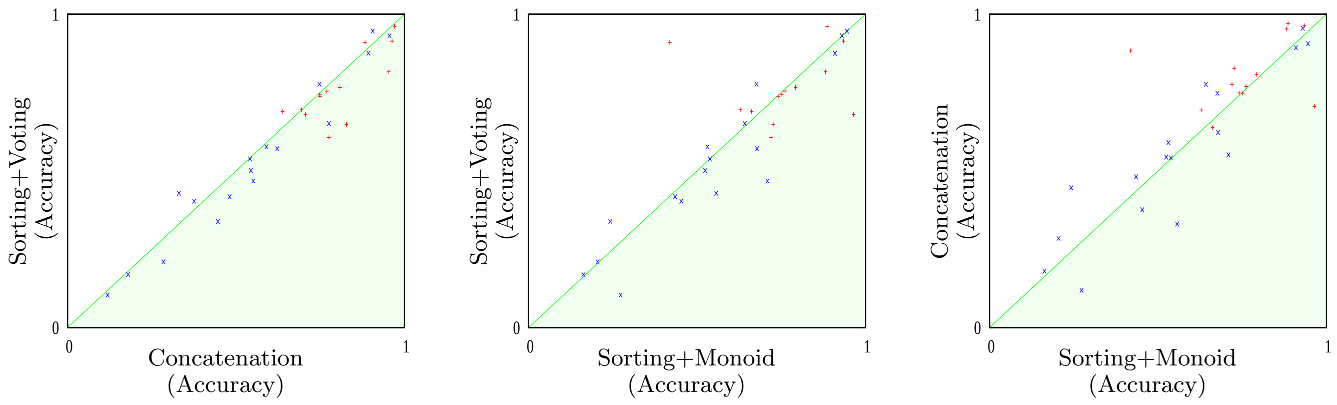

Figure 7. We compare the effectiveness of our different μ functions (Concatenation, Sorting+Voting and Sorting+Monoid). In each experiment we use AdaBoost with HomStumps as the base classifier. The supplemental material contains further details. (*left*) The concatenation method dramatically outperforms the sort method. (*center*) The monoid method tends to outperform the sort method, but much less dramatically. (*right*) The concat method and monoid method perform well on different data sets. With an unstable base classifier like HomStumps, the concat method tends to do better.

adapted the AdaBoost algorithm to run on the MapReduce architecture. In so doing, they implicitly define a monoid structure for $\mathcal{M}_{Boost}$. By adapting their algorithm to the Free HomTrainer, we make this monoid structure explicit and get online and fast cross-validation algorithms "for free."

The key to Palit and Reddy's algorithm is a reducing function that takes a set of boosting models and merges them into a single boosting model. This function is our second candidate for μ. We briefly review their technique here, but full details are provided in their paper. First, the components of each boosting model are sorted according to their weights. Then, the smallest weighted models get associated together into a single voting classifier; the next smallest get associated; and so on. Classification proceeds as another 2-tiered weighted vote, just as for $\mathcal{F}(\mathcal{M}_{\text{Boost}})$. The difference is that in this new model, we have rearranged the voting districts so that base models with similar weights are voting together. This gerrymandering tends to improve classification accuracy because base models with similar weights tend to vote in a similar manner.

As shown in Figure 7 (*left*), the direct democracy of our concatenation method outperforms this modified representative democracy. However, this method of sorting and voting is the inspiration for our next method.

5.2.3. Sorting+Monoid

If our base model has a monoid structure, we can create a third μ function. The procedure is the same as Palit and Reddy's, with one minor difference. When we are associating similarly weighted models together, we don't construct a voting classifier. Instead, we merge those moddels together into a single model with the monoid operation.

This modified algorithm gives us several advantages. First, it improves classification accuracy in many cases as shown empirically in Figure 7. Second, it requires asymptotically less space—the resulting model is simply a set of base classifiers whose size is the same as a single AdaBoost model's size.

6. Conclusion

The algebraic structure of our models is important. In the future, we hope to derive algebraic structures for other learning models, and develop new algorithms for operating over these structures. Finally, this algebraic approach need not be limited to supervised learning. We plan to apply it to other areas of machine learning as well.

Acknowledgements

Thank you to an anonymous reviewer who improved the run time of the monoid cross-validation algorithm from $O(n+k^2)$ to $O(n+k)$. Also, thank you to the Haskell community for encouraging programmers to explore the relationship between algebra and programming.

References

Arlot, Sylvain and Celisse, Alain. A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4:40–79, 2010.

Ben-david, Shai, Kushilevitz, Eyal, and Mansour, Yishay. Online Learning versus Offline Learning. *Machine Learning*, 29:45–63, 1997.

Breiman, Leo. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

Chen, Yen-kuang, Li, Wenlong, and Tong, Xiaofeng. Parallelization of AdaBoost algorithm on multi-core processors. In *IEEE Workshop on Signal Processing Systems*, pp. 275–280, 2008.

Chu, Cheng-tao, Kim, Sang Kyun, Lin, Yi-an, Yu, Yuanyuan, Bradski, Gary, Ng, Andrew, and Olukotun, Kunle. Map-Reduce for Machine Learning on Multicore. In *Neural Information Processing Systems*, pp. 281–288, 2006.

Dean, Jeffrey and Ghemawat, Sanjay. MapReduce: Simplied Data Processing on Large Clusters. In *Operating Systems Design and Implementation*, pp. 137–150, 2004.

Dekel, Ofer. From Online to Batch Learning with Cutoff-Averaging. In *Neural Information Processing Systems*, pp. 377–384, 2008.

Freund, Yoav and Schapire, Robert E. Experiments with a New Boosting Algorithm. In *International Conference on Machine Learning*, pp. 148–156, 1996.

Grabner, Helmut, Leistner, Christian, and Bischof, Horst. *Semi-Supervised On-Line Boosting for Robust Tracking.* 2008.

Iba, Wayne and Langley, Pat. Induction of One-Level Decision Trees. In *International Conference on Machine Learning*, pp. 233–240, 1992.

Kakade, Sham M. and Kalai, Adam. From Batch to Transductive Online Learning. In *Neural Information Processing Systems*, 2005.

Kondor, Risi Imre and Borgwardt, Karsten M. The skew spectrum of graphs. In *International Conference on Machine Learning (ICML)*, pp. 496–503, 2008.

Kondor, Risi Imre, Howard, Andrew, and Jebara, Tony. Multi-object tracking with representations of the symmetric group. 2:211–218, 2007.

Littlestone, Nick. From On-Line to Batch Learning. In *Computational Learning Theory*, pp. 269–284, 1989.

Merler, Stefano, Caprile, Bruno, and Furlanello, Cesare. Parallelizing AdaBoost by weights dynamics. *Computational Statistics & Data Analysis*, 51:2487–2498, 2007.

Oza, Nikunj C. Online Bagging and Boosting. 2001.

Pachauri, Deepti, Collins, Maxwell, Kondor, Risi Imre, and Singh, Vikas. Incorporating domain knowledge in matching problems via harmonic analysis. In *International Conference on Machine Learning (ICML)*, 2012.

Palit, I. and Reddy, C.K. Scalable and parallel boosting with mapreduce. *Knowledge and Data Engineering, IEEE Transactions on*, PP(99):1, 2011. ISSN 1041-4347.

Parhami, B. *Introduction to Parallel Processing: Algorithms and Architectures.* Plenum Series in Computer Science. Springer, 1999.

Tesson, Pascal and Thrien, Denis. Monoids and Computations. *International Journal of Algebra and Computation*, 14:801–816, 2004. doi: 10.1142/S0218196704001979.

Vaikuntanathan, Vinod. Computing blindfolded: New developments in fully homomorphic encryption. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, FOCS '11, pp. 5–16, Washington, DC, USA, 2011. IEEE Computer Society.

Viola, Paul A. and Jones, Michael J. Robust Real-Time Face Detection. *International Journal of Computer Vision*, 57:137–154, 2004.

Watanabe, Sumio. *Algebraic geometry and statistical learning theory.* Cambridge University Press, 2009.

Yorgey, Brent A. Monoids: theme and variations (functional pearl). In *Proceedings of the 2012 symposium on Haskell symposium*, Haskell '12, pp. 105–116, New York, NY, USA, 2012. ACM.

Yu, C. and b. Skillicorn, D. Parallelizing Boosting and Bagging. 2001.