HLearn: A Machine Learning Library for Haskell

Michael Izbicki

UC Riverside

Abstract. HLearn is a Haskell-based library for machine learning. Its distinguishing feature is that it exploits the algebraic properties of learning models. Every model in the library is an instance of the `HomTrainer` type class, which ensures that the batch trainer is a *monoid homomorphism*. This is a restrictive condition that not all learning models satisfy; however, it is useful for two reasons. First, this property lets us easily derive three important functions for machine learning algorithms: online trainers, parallel trainers, and fast cross-validation algorithms. Second, many popular algorithms (or variants on them) satisfy the condition and are implemented in the library. For example, the HLearn library implements the standard version of many distribution estimators and Bayesian classification, as well as homomorphic variants of perceptrons, kd-trees, decision trees, ensemble algorithms, and clustering algorithms. Furthermore, many of these learning models have additional algebraic structure that the HLearn library exploits. In particular, if a model has Abelian group structure, then we can perform more efficient cross-validation; and if it has R-module structure, then we can use weighted data points. Hopefully, this algebraic framework makes it easier to incorporate machine learning into the average Haskell application.

1 Why another library for machine learning?

Machine learning libraries need to be fast. In order to get this speed, the most popular libraries are written in low level languages (see Table 1). Unfortunately, this emphasis on speed has meant that the libraries are often inconvenient to use. Current practice is to provide bindings to these low level libraries in higher level languages (e.g. R, Matlab, Python, and even Haskell); but this still leaves much to be desired. These interfaces are not standardized and require specialist knowledge to understand and use. In practice, they are rarely used by the average programmer writing an average program. The machine learning community is well aware of this deficiency, although there is relatively little effort to fix it [20].

The goal of the HLearn[1] library is to change this status quo and make machine learning techniques easily usable by non-specialists. We don't claim to have solved this problem, but only that we're aiming in that direction. We do this by characterizing learning models according to their algebraic structure. This is a

[1] The H in HLearn stands for both Haskell (because that is the language the library is written in) and homomorphism (because all batch trainers in the library must be homomorphisms; see section 2).

Table 1. Most popular machine learning libraries are written in non-functional languages. Weka is the most fully featured of these packages, and it is the easiest for a novice to use. It is no coincidence that it is written in the highest level language.

Library	Language
C4.5 Decision Trees	C
Fast Artificial Neural Networks (FANN)	C
Stuttgart Neural Network Simulator (SNNS)	C
Support Vector Machines Light (SVMLight)	C
Library for Support Vector Machines (LibSVM)	C++ and Java
Open Computer Vision (OpenCV)	C++
Weka	Java

powerful design pattern commonly used in functional programming libraries [21]. This pattern makes libraries easy to build and maintain by reducing the amount of boilerplate code. More importantly, it makes libraries easy to use—once a user understands an algebraic structure, then she automatically understands all instances of that structure. For example, the normal distribution forms a *vector space*, but Markov chains only form a *monoid*. As we shall see later, this means that they can use the same functions for online and parallel training, but that normal distributions support more efficient cross-validation and the weighting of data points. The user doesn't have to know anything about how these models actually work, just what algebraic structure they have.

Of course, there have been many other attempts to write machine learning algorithms in a functional language. The fact that probability distributions have a *monad* structure [9, 12, 18] has formed the foundation for a number of libraries for probabilistic programming [2, 7, 15]. The HLearn library is orthogonal to this work, and in principle both designs could exist side-by-side. Other attempts to integrate functional programming with machine learning have not used algebra [3, 14, 1]. Instead, they use the power of type classes, higher order functions, and pattern matching to express learning algorithms in a functional setting. The HLearn library incorporates much of these ideas. Finally, we note that all of this previous work has taken place within the Haskell language. Haskell is a good choice for this type of experimentation because it is both fast and was designed from the beginning to incorporate experimental language features [11, 17]. HLearn relies on a number of recent language extensions implemented in the Glasgow Haskell Compiler (GHC), such as `TypeFamilies`, `GADTs`, `DataKinds`, `ConstraintKinds`, and `TemplateHaskell`.

Besides the theoretical advantages of the HLearn library, there is also a practical advantage: a standardized interface for many learning tasks. While it's true that there are a number of excellent Haskell packages for machine learning,[2] unfortunately each of these packages uses a different interface. This makes it

[2] There are too many packages to describe them all here. For a complete list, visit the Hackage repository (http://hackage.haskell.org), and look under the sections: statistics, artificial intelligence, machine learning, and data mining.

difficult to compose learning routines, ruining one of the main advantages of functional programming. For example, the `statistics` package assumes that all input data points will be stored inside unboxed vectors, whereas the `KdTree` package requires data points stored in lists. The HLearn library has no such requirements—we can store our data points however is most convenient for our particular application. We only require that the container be a foldable functor. One neat trick this lets us do is work with data sets larger than memory by using containers that seamlessly swap to and from disk.

The remainder of this paper focuses on HLearn's internal mechanics. Section 2 describes the `HomTrainer` type class, and why it accurately captures our notions of what it means to be a learning model. The `HomTrainer` class gives us a simple method for defining new models, and useful bounds on their training time. Section 3 looks at other algorithms for manipulating algebraic models. These let us automatically parallelize our training procedures, perform asymptotically faster cross-validation, and apply weightings to our data points. Section 4 concludes with the future of the HLearn library. Finally, the haddock documentation contains tutorials and further details on practical usage of the library.[3]

2 The HomTrainer type class

Every learning model is represented by a data type, and that type must be an instance of `HomTrainer`. Table 3 lists all current instances. The wide selection of models—there are statistical distributions, classifiers, unsupervised learners, Markov chains, and even $\mathbb{NP}$-approximation algorithms—demonstrates the versatility of the `HomTrainer` class. In this section, we will show why the class is also powerful.

For each of these models, the `HomTrainer` class associates a unique `Datapoint` type and provides four training functions (see Code Snippet 1). The two most important training functions are the batch trainer `train` and the online trainer `add1dp`. The batch trainer takes a set of data points and returns the corresponding trained model. Typically, we would use it when analyzing historical data that was generated by some previous process. In contrast, the online trainer is used for analyzing data as it is generated. It takes an already trained model and "adds" a data point to the model. Which is more useful depends on our particular application.

There is a lot of interest in the machine learning community about the relationship between online and batch training. In general, online training is much harder [16, 4, 13, 6]. For our purposes, this means that not every learning model has a known online trainer that would satisfy the laws of the `HomTrainer` type class (discussed below). By limiting ourselves in this way, we gain a simpler, more powerful, easier to use interface. This tradeoff is reasonable because many popular learning algorithms have variants that do satisfy the `HomTrainer` laws.

The `HomTrainer` class also includes two other functions. First, the singleton trainer `train1dp` is included because it often makes defining new models easier.

[3] http://hackage.haskell.org/package/HLearn-algebra

Code Snippet 1 The `HomTrainer` type class

```
class (Monoid model) ⇒ HomTrainer model where
    type Datapoint model

    -- The singleton trainer
    train1dp :: Datapoint model → model

    -- The batch trainer
    train :: (Functor container, Foldable container) ⇒
        container (Datapoint model) → model

    -- The online trainer
    add1dp :: model → Datapoint model → model

    -- The online batch trainer
    addBatch :: (Functor container, Foldable container) ⇒
        model → container (Datapoint model) → model
```

As Section 2.1 shows, we only need to implement one of the training functions and the rest can be derived automatically. The singleton trainer can often be implemented in only a single line of code. Second, the online batch trainer `addBatch` is included for efficiency reasons. If we have a large list of data points to add to our model, it is more efficient to add them all at a single time than it is to add them one-by-one.

Every instance of `HomTrainer` must obey four laws. First, the batch trainer must be a monoid homomorphism. That is,

```
train (xs ++ ys) = (train xs) ◇ (train ys)
```

The next three laws ensure that no matter how we train our model, as long as we use the same data points we will get the same model:

```
add1dp (train xs) x = train (xs ++ [x])
train1dp x = train (point x)
addBatch (train xs) ys = train (xs ++ ys)
```

Next, we discuss how to define new `HomTrainer`s and the complexity of the resulting training functions.

2.1 Construction

In this section, we present four higher order functions that convert between the four types of training functions. The functions are shown in Code Snippet 2.

Code Snippet 2 Higher order functions for constructing training functions

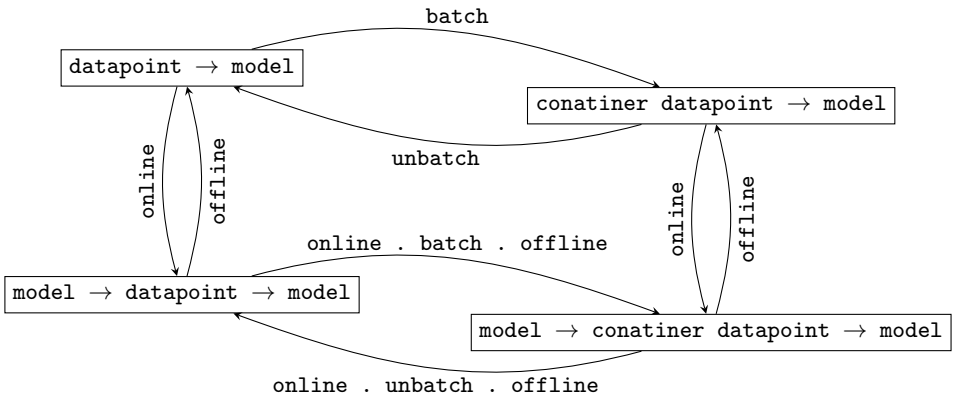
```
-- convert singleton trainer into batch trainer
batch :: (Functor container, Foldable container) ⇒
    (datapoint → model) → (container datapoint → model)
batch f = λ xs → reduce (map f xs)

-- convert batch trainer into singleton trainer
unbatch :: (Functor container) ⇒
    (container datapoint → model) → (datapoint → model)
unbatch f = λ x → f [x]

-- convert singleton trainer into online trainer
online :: (datapoint → model) → (model → datapoint → model)
online f = λ m x → m ◇ f x

-- convert online trainer into singleton trainer
offline :: (model → datapoint → model) → (datapoint → model)
offline f = λ x → f mempty x
```

When implementing the `HomTrainer` instance for a particular model, we need to implement only one of the training functions. Then, we can use these higher order functions to derive the other trainers. We can do this because the following diagram commutes:



Next, we must prove that the resulting training functions obey the `HomTrainer` laws. We show the proof for the first law, and the other three laws are similar:

Theorem 1. *Given a singleton trainer* `f :: datapoint → model`*, the batch trainer* `batch f :: container datapoint → model` *is a monoid homomorphism.*

Proof. Let `xs` and `ys` be arbitrary data sets with type

```
(Functor container, Foldable container) ⇒ container datapoint
```

Then, the following statements are equivalent:

```
(batch f) (xs ++ ys)
(λ xs → reduce (map f xs)) (xs ++ ys)
reduce (map f (xs ++ ys))
reduce ((map f xs) ++ (map f ys))
(reduce (map f xs)) ◇ (reduce (map f ys))
((batch f) xs) ◇ ((batch f) ys)
```

The first and last lines above are the definition of a monoid homomorphism. □

2.2 Computational Complexity

Now that we have seen a simple way to construct `HomTrainer`s, we must investigate the complexity of these constructions. In particular, we will compare the batch trainer's running time with the monoid operation's running time.

First, we need some definitions. We denote the batch trainer's running time as $\alpha(n)$, where n is the number of elements in the input data set. In order to compare this complexity to that of the monoid operation, we must write the monoid operations's run time in terms of n. We do this by defining the model's size as the number of elements it took to train it. We can therefore write the running time of `m1 ◇ m2` as a function $\beta(n_1, n_2)$, where n_i is the number of elements it took to train model `mi`.

This is not a standard technique in computational algebra. Most previous work assumes that the algebraic operations take constant time. That is, the complexity of an algorithm is determined only by the number of times the algebraic operations get called. This assumption does not make sense for us, however, because there are many algebraic operations in Haskell that do not run in constant time. Haskell's immutable lists provide a familiar example. Using our notation above, the function `(++) :: [a] → [a] → [a]` has time complexity $\beta(n_1, n_2) = n_1$. Table 3 shows the run times for the batch trainer and monoid operations of all the learning methods currently implemented in HLearn, many of which also have non-constant run time.

Upper bound on the batch trainer We can always use the function `batch` to construct a batch trainer from the singleton trainer and the monoid operation; therefore, this construction creates an upper bound on the batch trainer's run time. The batch trainer created by `batch` proceeds in two steps. First, it `map`s the singleton trainer onto each of the n data points. This takes time $O(n)$ because the singleton trainer always takes constant time. Then, we `reduce` the result using the fan-in reduction strategy (see Code Snippet 3). Fan-in reduction is an iterative procedure. On each iteration i, we group pairs of elements of the list and apply the monoid operation. This results in $O(\frac{n}{2^i})$ monoid operations per iteration. The number of data elements in each monoid element on iteration i is

Code Snippet 3 Fan-in reduction on lists

```
reduceL :: (Monoid m) ⇒ [m] → m
reduceL []  = mempty
reduceL [x] = x
reduceL xs  = reduceL $ itr xs
    where
        itr :: (Monoid m) ⇒ [m] → [m]
        itr []            = []
        itr [x]           = [x]
        itr (x1:x2:xs)    = (x1⋄x2):(itr xs)
```

$O(2^i)$, so the cost of the monoid operation is $\beta(2^i, 2^i)$. After $\lceil \log_2 n \rceil$ iterations we will be left with only a single element and be done. The total run time of the reduction is the sum of each iteration's run time, which is:

$$\sum_{i=0}^{\lceil \log_2 n \rceil} \frac{n}{2^{i+1}} \beta(2^i, 2^i)$$

The batch trainer's run time $\alpha(n)$ is upper bounded by the sum of the map and reduce steps:

$$\alpha(x) \leq n + \sum_{i=0}^{\lceil \log_2 n \rceil} \frac{n}{2^{i+1}} \beta(2^i, 2^i)$$

which simplifies to:

$$\alpha(x) \leq O\left(n \sum_{i=0}^{\log n} \frac{\beta(2^i, 2^i)}{2^i} \right)$$

This equation is difficult to understand intuitively, so Table 2 precomputes limits on α in terms of β. Notice that under the standard assumption that the monoid operation takes time $\beta(n_1, n_2) = O(1)$, the fan-in reduction has the same running time as a right or left fold, i.e. $O(n)$. But in the case where $\beta(n_1, n_2) > O(1)$, the fan-in reduction will be asymptotically faster.

Table 2. Upper bound of the batch trainer's run time $\alpha(n)$ in terms of the monoid operation's run time $\beta(n_1, n_2)$, with $n = n_1 + n_2$. The parallelization procedure is described in Section 3.1.

$\beta(n_1, n_2)$	$\alpha(n)$	parallel batch trainer with p processors
$O(1)$	$O(n)$	$O\left(\frac{n}{p} + \log p\right)$
$O(\log^a n)$, $a > 0$	$O(n)$	$O\left(\frac{n}{p} + (\log^a n)(\log p)\right)$
$O(n)$	$O(n \log n)$	$O\left(\frac{n}{p} \log \frac{n}{p} + n\right)$
$O(n^b)$, $b > 1$	$O(n^b)$	no improvement

The results in Table 2 give us two useful tools for implementing new `HomTrainer` instances. First, many real-world learning problems have monoid operations with non-constant run time, and Table 2 simplifies our analysis of their batch trainers. For example, finger trees form a monoid whose binary operation takes time $O(\log n)$ [10]. The k-Centers clustering algorithm implemented in HLearn uses finger trees (as implemented in `Data.Sequence`) to keep track of which points are in each cluster. This dominates the running time of the monoid operation, so by Table 2 we can prove that our batch trainer still must have a linear run time. As a second example, binary search trees form a monoid whose binary operation takes time $O(n)$. In the HLearn library, we implement kd-trees using binary search trees. This means that their monoid operation takes time $O(n)$, and so their batch trainer takes time $O(n \log n)$.

Second, if our monoid operation takes time greater than $O(n)$, then it is not useful. The resulting batch trainer would take the exact same time as the monoid operation. Therefore, the monoid operation can never save us any work. We are only interested in monoid operations with run time less than or equal to $O(n)$.

Upper bound on the monoid operation's complexity We can trivially bound the monoid operations's complexity in terms of the batch trainer. Because we know that for all data sets `xs` and `ys`:

```
train xs ◇ train ys = train (xs ++ ys)
```

We can always use the batch trainer to directly perform the monoid operation. Therefore, the monoid operation must take time less than or equal to the batch trainer on a similar input. That is,

$$\beta(n_1, n_2) \leq \alpha(n_1 + n_2)$$

Of course, if $\beta(n_1, n_2) = \alpha(n_1 + n_2)$, then using our monoid operation will not save us any work and so it is essentially useless. We are only interested in models where β is strictly less than α.

3 More fun with algebra

HLearn uses algebra for more than just deriving `HomTrainer`s. The beauty of algebra is that if we can write an algorithm for any particular algebraic structure, then it will apply to all instances of that structure. This makes our algorithms as generic as possible. In this section, we will see three of these algorithms. First, we'll create a higher order function called `parallel` that makes any monoid homomorphism run efficiently on multiple cores. This function will let us easily parallelize the training of any model in the HLearn library. Second, we'll see two algorithms for fast cross-validation that take advantage of monoid and Abelian group structures. Finally, we'll see that R-Module structures give us the ability to weight the importance of our data points.

3.1 Parallelization

In this section, we show how to parallelize monoid homomorphisms. Monoids have a strong theoretical connection with parallel computing. In particular, the "parallel complexity class" $\mathbb{NC}^1$ can be defined in terms of programs over finite monoids [19]. For a more concrete example, the reduce step in Google's MapReduce framework is a monoid computation [5]. Our contribution is that we do not make the assumption that the monoid operation takes constant time. This generalization will let us train any `HomTrainer` instance in parallel.

The parallelization procedure has three steps. Given p processors: (i) partition the data set into p subsets; (ii) `map` a batch trainer on the list of subsets; then (iii) `reduce` the results. The parallel execution time is the sum of each of these steps. Step (i) takes constant time for each partition, and each partition can be created in parallel. Step (ii) takes time $\alpha(\frac{n}{p})$ for each subset, and each subset can be processed in parallel. Step (iii) takes $\lceil \log_2 p \rceil$ reduction iterations. In each iteration, every reduction can be done in parallel; therefore, iteration i only takes as long as a single monoid operation $\beta(\frac{n}{p}2^i, \frac{n}{p}2^i)$. The overall execution time is the sum of these steps:

$$\alpha\left(\frac{n}{p}\right) + \sum_{i=0}^{\lceil \log_2 p \rceil} \beta\left(\frac{n}{p}2^i, \frac{n}{p}2^i\right)$$

Table 2 precalculates the parallel executation time for a number of cases, and Table 3 shows the parallel run time for all implemented models.

HLearn currently implements local parallelism using the higher order function:

```
parallel :: (Semigroup model, NFData model, Partitionable container) ⇒
    (container datapoint → model) → (container datapoint → model)
```

This function takes a sequential batch trainer, and returns a parallel one. The resulting function will use all available cores on the machine efficiently. Those readers familiar with Google's MapReduce framework [5] should see that our procedure can be easily implemented using MapReduce. It would be nice to create a similar higher order function that would perform distributed parallelism on a Haskell-based MapReduce cluster using, for example, the Holumbus project.[4]

3.2 Algebraic algorithms for faster cross-validation

Cross-validation is a procedure for estimating the accuracy of a learning model. We use cross-validation to avoid one of the most common mistakes in data analysis: over fitting. Over fitting happens when our trained model performs well on the input data points, but generalizes poorly to other data points it hasn't yet seen. When analyzing data, we must run cross-validation procedures every time we train a new model to ensure we are not over fitting. Faster cross-validation therefore makes the practicing data analyst more efficient. HLearn's

[4] http://holumbus.fh-wedel.de/trac

contribution is to make it clear when we can and cannot use fast cross-validation. In this section we will present the standard method for k-fold cross-validation, then compare it to the faster Abelian group cross-validation. Both methods calculate the exact same answer; the only difference is their run time.

Haskell code for k-fold cross-validation is shown in Code Snippet 4. In the standard algorithm, we first divide the data set into k subsets. The `crossvalidate` function assumes this has already been done for us, and accepts the subsets as the input variable `xs`. The `do` notation is for the list monad. In each iteration, we select one of the subsets. This is called the `testset`. The union of the remaining subsets is the `trainingset`. We train our model on the `trainingset`, and measure its performance on the `testset` using a `LossFunction`. Loss functions are how we measure just how well our algorithm performs. For classification tasks, error rate and log-loss are two popular functions. The final result of cross-validation is the mean and variance of the losses from each iteration. We calculate this by training a `Normal` distribution. If we make the simplifying assumption that our loss function takes constant time and our training function takes time $O(n)$, then our overall run time is $O(kn)$.

Code Snippet 4 Standard cross-validation

```
type LossFunction model = model → [Datapoint model] → Double

crossvalidate :: (HomTrainer model, Eq (Datapoint model)) ⇒
    [[Datapoint model]] → LossFunction model → Normal Double
crossvalidate xs f = train $ do
    testset ← xs
    let trainingset = concat $ filter (/=testset) xs
    let model = train trainingset
    return $ f model testset
```

We can speed up this procedure using Abelian groups. Abelian groups are monoids with two extra properties. Type classes for these properties are shown in Code Snippet 5.

Code Snippet 5 Abelian and Group type classes

```
class (Monoid m) ⇒ Abelian m
class (Monoid m) ⇒ Group m where
    inverse :: m → m
```

The `Abelian` type class has no member functions, but all instances must have a commutative monoid operation. That is, they must obey the law that:

```
m1 ◇ m2 == m2 ◇ m1
```

Intuitively, if a learning model is Abelian, then the order in which we train data points does not matter. For example, when estimating a normal distribution from a list of numbers, we don't care what order the numbers are in. We'll get the same answer for all possible orderings. Therefore, the normal distribution is Abelian.

Second, groups are monoids that provide a unary operation `inverse`. Groups must obey the law that:

```
m ◇ inverse m == inverse m ◇ m == mempty
```

Intuitively, this `inverse` function let's us subtract models from each other.

Haskell code for Abelian group cross-validation is shown in Code Snippet 6. First, we train a model for each subset of data points. In the code, this gets zipped with the points themselves to create the variable `modelL`. We can `reduce` this list of models to get the model trained on the entire data set, called `fullmodel`. In each iteration of `do`, we then use the `inverse` function to subtract those data points from `fullmodel` to generate the model we will test. Because of the laws of the `Abelian` and `Group` classes, we are guaranteed to get the same exact answer as with standard cross-validation. But our run time is now much less. With the same simplifying assumptions as above, we get a run time of $O(k + n)$ instead of $O(kn)$.

Code Snippet 6 Abelian group fast cross-validation

```
crossValidate_group :: (HomTrainer model, Group model) ⇒
    [[Datapoint model]] → LossFunction model → Normal Double
crossValidate_group xs f = train $ do
    (testset,testModel) ← modelL
    let model = fullmodel ◇ inverse testModel
    return $ f model testset
    where
        modelL = zip xs $ map train xs
        fullmodel = reduce $ map snd modelL
```

3.3 R-Modules and weighted data

Intuitively, the weight of a data point specifies "how many times" the data point appears in the data set. For example, if our data points are of type `Char`, and our weighted data points are of type `(Int,Char)`, then the data sets `['a','a','a','a','a']` and `[(5,'a')]` are equivalent—they will train exactly the same model. Handling these weighted data points is a common task in data anal-

ysis. In this section, we show that models that have the algebraic structure of an R-module allow data points weighted by elements in the ring R.[5]

An R-module is an Abelian group with operations for scalar multiplication. The HLearn library uses the `Module` type class to capture this structure, as shown in Code Snippet 7.

Code Snippet 7 R-Module type class

```
class (Num (Ring m), Abelian m, Group m) ⇒ Module m where
    type Ring m
    (.∗) :: Ring m → m → m
    (∗.) :: m → Ring m → m
```

All instances must obey the standard R-module laws. That is, if `r ~ Ring m`, then:

```
r .∗ m = m ∗. r
r .∗ (m1 ◇ m2) = (r .∗ m1) ◇ (r .∗ m2)
(r1 + r2) .∗ m = (r1 .∗ m) ◇ (r2 .∗ m)
(r1 ∗ r2) .∗ m = r1 .∗ (r2 .∗ m)
1 .∗ m = m
```

These laws imply the following classic theorem from algebra:

Theorem 2. *If* `f :: (Module m1, Module m2) ⇒ m1 → m2` *is a monoid homomorphism, then* `f` *is also an R-module homomorphism (more commonly called a linear transformation).*

We use this theorem to allow weighted data points with the `WeightedHomTrainer` type class (see Code Snippet As seen in the code, the single instance of the `WeightedHomTrainer` is all we need. To make a learning model handle weighted data points, we simply make it an instance of `Module`. 8).

Finally, we note that if our ring is actually a field (i.e. implements `Fractional`), then our model forms a vector space and can be weighted with fractional data points. This means that `(0.5,'a')` would be a valid data point in our earlier example. It doesn't necessarily "make sense" to have only half a data point in our data set; however, this generalization is useful in practice—the resulting weighted training methods are equivalent to the standard methods for weighted training.

4 Future directions

The HLearn library is still a work in progress, and it does not yet meet it's goal of making machine learning techniques accessible to the lay-programmer.

[5] We use the `Num` type class to represent rings.

Code Snippet 8 The `WeightedHomTrainer` type class

```
class (Module ring model, HomTrainer datapoint model) ⇒
    WeightedHomTrainer ring datapoint model
        where

    train1dpW :: (ring,datapoint) → model
    train1dpW (r,dp) = r .* train1dp dp

    trainW :: (Foldable container, Functor container) ⇒
        container (ring,datapoint) → model
    trainW = batch train1dpW

    add1dpW :: model → (ring,datapoint) → model
    add1dpW = online $ unbatch $ offline addBatchW

    addBatchW :: (Foldable container, Functor container) ⇒
        model → container (ring,datapoint) → model
    addBatchW = online trainW

instance (Module ring model, HomTrainer datapoint model) ⇒
    WeightedHomTrainer ring datapoint model
```

It needs three practical improvements to get there. First, the interface needs to be stabalized. It is currently under heavy development, which results in interface changes that would break client programs. Second, numerical stability is a major concern in libraries that involve statistics. Currently, it requires specialist knowledge to understand when numerical stability breaks down on many of HLearn's models. Finally, HLearn still needs to be faster. Asymptotically, all the implemented routines should be correct, but there is still a relatively large constant factor for many models. The library has seen almost no optimization work yet, and so speed improvements should be possible.

HLearn also presents exciting directions for future research. The machine learning community has done essentially no work on the algebraic structures of learning models, and is worse off for the neglect. We are currently working on a homomorphic variant of the AdaBoost algorithm [8]. This is a popular method in machine learning. It has a number of online and parallel variants, but these are all approximations, and none of these satisfy the `HomTrainer` laws. By developing an algebraic variant of AdaBoost we would get *exact* online and parallel algorithms, as well as the first method for fast cross-validation.

There is also still a lot of interesting work to be done from the functional programing perspective as well. In particular, the `HomTrainer` class should be easy to use with reactive programming and streaming IO libraries. But more of the practical issues with the library will need to be taken care of before it makes sense to explore in this direction.

References

1. Darko Aleksovski, Martin Erwig, and Saso Dzeroski. A Functional Programming Approach to Distance-based Machine Learning. 2008.
2. Lloyd Allison. Types and Classes of Machine Learning and Data Mining. In *ACM International Conference Proceeding Series*, pages 207–215, 2003.
3. Lloyd Allison. Models for machine learning and data mining in functional programming. *Journal of Functional Programming*, 15:15–32, 2005.
4. Shai Ben-david, Eyal Kushilevitz, and Yishay Mansour. Online Learning versus Offline Learning. *Machine Learning*, 29:45–63, 1997.
5. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplied Data Processing on Large Clusters. In *Operating Systems Design and Implementation*, pages 137–150, 2004.
6. Ofer Dekel. From Online to Batch Learning with Cutoff-Averaging. In *Neural Information Processing Systems*, pages 377–384, 2008.
7. Martin Erwig and Steve Kollmansberger. Functional Pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16:21–34, 2006.
8. Yoav Freund and Robert E. Schapire. Experiments with a New Boosting Algorithm. In *International Conference on Machine Learning*, pages 148–156, 1996.
9. Michèle Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer Berlin Heidelberg, 1982.
10. Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16:197–217, 2006.
11. Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *History of Programming Languages*, pages 1–55, 2007.
12. Claire Jones and Gordon D Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 186–195. IEEE, 1989.
13. Sham M. Kakade and Adam Kalai. From Batch to Transductive Online Learning. In *Neural Information Processing Systems*, 2005.
14. Nittaya Kerdprasop and Kittisak Kerdprasop. Mining Frequent Patterns with Functional Programming. 2007.
15. Eric Kidd. Build your own probability monads. 2007.
16. Nick Littlestone. From On-Line to Batch Learning. In *Computational Learning Theory*, pages 269–284, 1989.
17. Simon Marlow. Haskell 2010 language report.
18. Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):325–353, 1996.
19. Pascal Tesson and Denis Thrien. Monoids and Computations. *International Journal of Algebra and Computation*, 14:801–816, 2004.
20. Kiri Wagstaff. Machine learning that matters. *International Conference on Machine Learning (ICML)*, 2012.
21. Brent A. Yorgey. Monoids: theme and variations (functional pearl). In *Proceedings of the 2012 symposium on Haskell symposium*, Haskell '12, pages 105–116, New York, NY, USA, 2012. ACM.

Table 3. Learning models currently implemented or under development in the HLearn library. More details on each model can be found in the haddock documentation.

	Model	Algebraic Structure				Running Times			
		Monoid	Group	Abelian	Vector Space	batch trainer (sequential)	batch trainer (parallel)	monoid operation	model use*
Distributions	Exponential	X	X	X	X	$O(n)$	$O\left(\frac{n}{p} + \log p\right)$	$O(1)$	$O(1)$
	LogNormal	X	X	X	X	$O(n)$	$O\left(\frac{n}{p} + \log p\right)$	$O(1)$	$O(1)$
	Normal	X	X	X	X	$O(n)$	$O\left(\frac{n}{p} + \log p\right)$	$O(1)$	$O(1)$
	Kernel Density Estimator†	X	X	X	X	$O(nk)$	$O\left(k\frac{n}{p} + k\log p\right)$	$O(k)$	$O(1)$
	Binomial	X	X	X	X	$O(n)$	$O\left(\frac{n}{p} + \log p\right)$	$O(1)$	$O(1)$
	Categorical	X	X	X	X	$O(nc)$	$O\left(\frac{nc}{p} + c\log p\right)$	$O(c)$	$O(\log c)$
	Geometric	X	X	X	X	$O(n)$	$O\left(\frac{n}{p} + \log p\right)$	$O(1)$	$O(1)$
	Poisson	X	X	X	X	$O(n)$	$O\left(\frac{n}{p} + \log p\right)$	$O(1)$	$O(1)$
	Multivariate Normal	X	X	X	X	$O(nd^2)$	$O\left(d^2\frac{n}{p} + d^2\log p\right)$	$O(d^2)$	$O(d^2)$
	Multivariate Categorical	X	X	X	X	$O(nc^d)$	$O\left(c^d\frac{n}{p} + c^d\log p\right)$	$O(c^d)$	$O(d\log c)$
Classifiers	Naive Bayes	X	X	X	X	$O(nd)$	$O\left(d\frac{n}{p} + d\log p\right)$	$O(d)$	$O(d)$
	Full Bayes	X	X	X	X	*variable*	*variable*	*variable*	*variable*
	Decision Stumps†	X	X	X	X	$O(nd)$	$O\left(d\frac{n}{p} + d\log p\right)$	$O(d)$	$O(1)$
	Decision Trees†	X	X	X	X	*variable*	*variable*	*variable*	*variable*
	k-Nearest Neighbor (naive)	X	X	X	X	$O(1)$	$O(1)$	$O(n)$	$O(n^2)$
	k-Nearest Neighbor (kd-trees)†	X	X	X	X	$O(n\log n)$	$O\left(\frac{n}{p}\log\frac{n}{p} + n\right)$	$O(n)$	$O(\log n)$
	Perceptron†	X	X	X	X	$O(nd)$	$O\left(d\frac{n}{p} + d\log p\right)$	$O(d)$	$O(d)$
	Bagging†	X	X	X	X	*variable*	*variable*	*variable*	*variable*
	Free HomTrainer	X	X	X	X	*variable*	*variable*	*variable*	*variable*
Other	Markov Chains	X	X	-	-	$O(nk)$	$O\left(k\frac{n}{p} + k\log p\right)$	$O(k)$	$O(d)$
	k-Centers	X	-	-	-	$O(n)$	$O\left(\frac{n}{p} + (\log n)(\log p)\right)$	$O(\log n)$	N/A
	Partition	X	X	X	X	$O(n\log n)$	$O\left(\frac{n}{p}\log\frac{n}{p} + n\right)$	$O(n)$	N/A

key

c : number of distinct categories in categorical data

d : dimension of a data point

k : a parameter specific to each training algorithm

n : number of data points

p : number of processors available

* : for distributions, this means calling `pdf`; for classifiers, `classify`

† : these algorithms have been significantly modified from their standard published versions in order to obey the `HomTrainer` laws