Data centric programming with HLearn

Mike Izbicki, UC Riverside

HLearn is a machine learning library written in Haskell. It is an experiment in using advanced language features to simplify the process of analyzing data. This talk proposal first describes how the project takes advantage of algebraic structures and the Haskell language; then it covers three broad areas for future exploration. The descriptions in this paper are necessarily brief due to the lack of space, however more background information can be found at the project's github page: `http://github.com/mikeizbicki/hlearn`.

1 Current work

1.1 Functional machine learning

The Journal of Machine Learning Research (JMLR) maintains a repository of open source machine learning libraries at `mloss.org`, and HLearn is the only listed library written in a pure functional language. Most machine learning researchers believe that such languages are too slow for their applications. But this is not true. The right figure below demonstrates that nearest neighbor queries in HLearn are competitive with the popular MLPack library written in C++.

The main advantage of using Haskell is that HLearn can experiment with new language features. For example, HLearn uses the `DataKinds` extension to allow models to specify their parameters at the type level rather than the value level. The compiler uses this knowledge to generate an optimized model for the user's specific choice of parameters. Enabling this feature on nearest neighbor queries makes runtimes about 10% faster.

1.2 Algebraic structures

Algebraic structures are a classic design pattern in functional programming, and HLearn applies this pattern to learning models. All learning models in HLearn are implemented as *monoids*. The compiler uses this structure to automatically derive three algorithms for: online learning, parallel learning, and asymptotically faster cross-validation. The left figure below shows that the run time for HLearn's cross-validation procedure is independent of the number of folds! Other algebraic structures provide different benefits. For example, *modules* allow weighted data points, and *monads* allow fast transfer learning and data preprocessing. The project's github page provides a full description and tutorials of their use in practice.

This design philosophy simplifies the development of new algorithms. For example, HLearn uses the cover tree data structure to implement nearest neighbor searches. Because the cover tree is implemented as a monoid, we get parallel and fast cross-validation algorithms for free. Such algorithms were previously unknown to the machine learning community.

2 Future work

2.1 New algebras

There are two interesting problems with designing a learning library around algebraic structures. First, it can be hard to find algebraic structure for a particular model that we're interested in. For example, monoids are one of the simplest algebraic structures, but appropriate monoids may not even exist for many complex numerical learning models like support vector machines. One way to tackle this problem is through approximate algebraic structures. Much of the current research on parallel and fast cross-validation algorithms involves finding solutions that only approximate the original. These algorithms can be generalized by an "approximate monoid" structure that obeys the "approximately associative" law that:

$$a + (b + c) \approx (a + b) + c$$

There are a number of ways to formalize these structures, but all of them are considerably more complicated than standard monoids. Therefore it is not yet clear if these approximate structures will be useful in practice.

The second interesting problem is designing language interfaces that can handle more esoteric algebraic structures than currently allowed in Haskell. For example, the functor structure for most distributions cannot be implemented with Haskell's `Functor` type class. In Haskell the mapping operation must work for arbitrary functions (i.e. they are endofunctors over Hask), but many useful functor structures cannot handle arbitrary mappings (i.e. they are functors between much smaller categories). Current language extensions like `ConstraintKinds` only partially solve this problem.

2.2 Encryption

Homomorphic encryption is an important field of security research. The idea is that many encryption algorithms are actually monoid homomorphisms. That is, if e is our encryption function and x and y are plaintexts, then:

$$e(x)e(y) = e(xy)$$

This is useful for analyzing data when x and y are also learning models and the encryption function respects their monoid structure. For example, the normal distribution can be described with three fixed point real numbers (the 0th, 1st, and 2nd raw unnormalized moments); the monoid structure is componentwise addition on these sufficient statistics; then the ElGamal cryptosystem acts as a monoid homomorphism. This allows us to write systems where an untrusted third party can perform machine learning computations on encrypted data without knowing the decryption key.

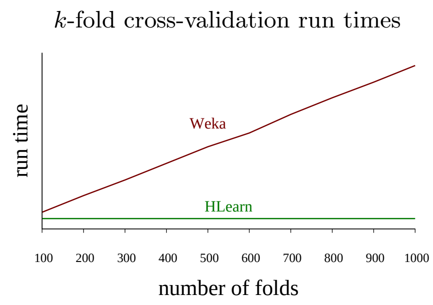
Current systems for this type of "privacy preserving" data analysis do not take advantage of algebraic structures. They are ad-hoc, difficult to develop and use, and often contain security vulnerabilities. HLearn is well suited to tackle this problem in a simple manner by taking advantage of the algebra. The goal is to develop a clean syntax to specify when to perform these computations in an encrypted manner, then have the type system guarantee that the computations are actually secure.

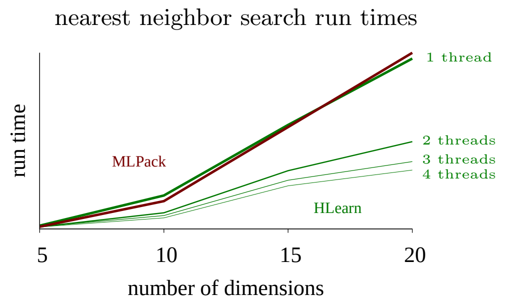
2.3 Programming by example

In programming by example, we give examples of inputs and outputs that a function should produce, and let the compiler infer an implementation. In theory, this could make it easier for novice programmers to write complex functions. But in practice, machine learning algorithms are not very good at inferring the correct function, and only experts can use these systems.

HLearn could be used to help develop these machine learning algorithms. This would be a two step process. First use template Haskell to generate the abstract syntax tree of example functions. Then learn a model over these syntax trees. Haskell's strong type system greatly reduces the number of possible implementations for any given function, which is an advantage that previous attempts at programming by example have not had.

Performance figures:





(*left*) HLearn performs k-fold cross-validation on Bayesian classifiers in time $O(n)$. This is independent of the number of folds, and faster than competing libraries which take time $O(kn)$. (*right*) HLearn performs a k-nearest neighbor search on a single core in time comparable time to other libraries. But HLearn also takes advantage of monoid structures and Haskell's purity to automatically parallelize these searches. (Each green line represents adding another thread of computation.) Results above were created on a laptop with two cores and two hyperthreads on randomly generated data.