OptiML Language Specification 0.1

Arvind K. Sujeeth
Stanford University
asujeeth@stanford.edu

1 Introduction

OptiML[**?**] is a domain-specific language for machine learning. It is embedded in Scala, and built on top of the Lightweight Modular Staging (LMS)[**?**] and Delite[**?**] frameworks. This document describes the structure, API and restrictions for OptiML[**?**] programs. It is organized into several catagories. Section 2 describes how an OptiML program is structured, and its basic operations, which are a subset of Scala. Section 4 and section 5 respectively describe the OptiML data and control structures and their associated operations and semantics. Finally, section **??** concludes with examples of simple OptiML programs.

Before diving in the details, it is important to clarify our design goals. OptiML is currently targeted at machine learning researchers and algorithm developers; it aims to provide a productive, high performance, MATLAB-like environment for linear algebra and supplemented with machine learning specific abstractions. Our primary goal is to allow machine learning practitioners to write code in a highly declarative manner and still achieve high performance on a variety of underlying parallel, heterogeneous devices. The same OptiML program should run well and scale on a CMP (chip multi-processor), a GPU, a combination of CMPs and GPUs, clusters of CMPs and GPUs, and eventually even FPGAs and other specialized accelerators.

In particular, OptiML is designed to allow statistical inference algorithms expressible by the Statistical Query Model[**?**] to be both easy to express and very fast to execute. These algorithms can be expressed in a *summation* form[**?**], and can be parallelized using fine-grained map-reduce operations. OptiML employs aggressive optimizations to reduce unnecessary memory allocations and fuse operations together to make these as fast as possible. OptiML also attempts to *specialize* implementations to particular hardware devices as much as possible to achieve the best performance.

Although nascent, OptiML also includes the beginning of a standard library for ML that incorporates common machine learning algorithms and classifiers. These functions are written in OptiML and allow machine learning application developers to easily make use of off-the-shelf machine learning algorithms.

2 Basics

2.1 Program structure

OptiML applications have the following basic structure:

```
  import ppl.dsl.optiml._

  object MyApplicationRunner extends OptiMLApplicationRunner with MyApplication
  trait MyApplication extends OptiMLApplication {
    def main() = { ... }
  }
}
```

The division into *Application* and *ApplicationRunner* is due to the fact that OptiML is embedded in Scala using LMS; for more details, about why this is necessary, see [**?**].

main() is the entry-point for OptiML applications. It implicitly imports an *args* variable into scope, that can be used to access parameters supplied to the program at run-time. For example, the command

```
scala MyApplicationRunner "hello"
```

would allow the parameter `"hello"` to be accessed using `args(0)`.

2.2 Declaring values and variables

The keywords **val** and **var** are used to declare constant and variable identifiers respectively. For example,

```
val x = 100
var y = 100
x = 7 // error; vals cannot be modified
y = 7 // ok
```

2.3 Conditionals

OptiML supports standard if/then/else statements. The syntax is:

```
if (cond) { ... }
else if { ... }
else { ... }
```

where any of the else clauses may be omitted.

2.4 Sequential iteration

OptiML supports **while** loops in the usual way. The semantics of OptiML's while loops are the same as Scala, Java, and C.[1] In particular, each iteration is guaranteed to be sequentially consistent. Note that OptiML will never attempt to parallelize *across* while loop iterations, but may parallelize independent tasks *within* an iteration. The result of a while loop is always deterministic. Example usage:

```
var i = 0
while (i < 10) {
  ...
}
```

[1] OptiML does not support Scala ranges, so standard Scala 'for loops' will generate a type error.

2.5 Methods

OptiML allows methods to be defined using the **def** keyword. For example,

```
  def main() = {
    val x = foo
    val y = bar()
    println(y) // prints "bar"
  }

  def foo() = {
    "foo"
  }

  def bar() = {
    "bar"
  }
}
```

This example defines two (pointless) methods that are invoked from inside *main*. Note that like Scala, parantheses may be omitted when calling a method. However, OptiML methods have two major differences from Scala methods:

- **return** is not supported. While this will eventually be rejected or implemented, currently it results in undefined behavior. The value on the last line of the method is the return value for the method, as in the previous examples.
- method parameters must be wrapped with the `Rep` type.[2]

Here is an example of using wrapped method parameters:

```
def foo(val x: Rep[Int], val y: Rep[Double], val z: Rep[Vector[Double]]) { ... }
```

2.6 Utility operations

OptiML provides a small set of utility operations automatically available in application scope:

- print, println
- exit

2.7 Scala subsetting

The astute reader familiar with Scala will notice that the set of operations listed here are a small subset of Scala operations and syntax. Any Scala operations or keywords not listed here are unsupported, and will result in undefined behavior (usually type errors). In the future, we intend to make this more robust, so that unsupported Scala operations are explicitly rejected with a sensible error message.

[2]This is an example where details of the embedding implementation currently still leak into application code. We are working to make this unnecessary with newer versions of the virtualized Scala compiler and LMS library.

3 Restricted Functions

OptiML has a number of operations that accept user-defined functions. In order to effectively fuse and parallelize these operations, OptiML does not allow arbitrary functions, but instead usually requires *restricted functions*. An OptiML *restricted function* is a function that must be pure, i.e. contain no side-effects.If an operation expecting a restricted function is supplied a function with arbitrary side-effects, it will (eventually) be rejected by the OptiML compiler during staging (this is not currently implemented). Certain OptiML operations relax this restriction in structured ways, and allow mutations to certain elements within a data structure; these relaxations are specified on a per-operation basis.

4 Data Types

OptiML supports a small number of data types geared towards linear algebra and machine learning. While it is possible to use **`import`** statements to import other Scala types into scope, these are not legal to use inside OptiML programs, and will usually result in type errors.

The data types supported by OptiML can be organized into several categories: primitives, vectors, matrices, streams, and graphs. All non-primitive OptiML data types are *polymorphic* and can be used with any type parameter. In other words, you can instantiate a `DenseVector[Foo]` if the type `Foo` exists in the OptiML application. One practical use-case for this is vectors which contain other vectors.

All OptiML data types support the following methods[3]:

operation	description
IsInstanceOf	dynamic type test
AsInstanceOf	type cast
ToString	string representation

Table 1: Generic operations

4.1 Primitives

4.1.1 Numerics

`Double`, `Float`, `Int`, are supported with the following operations:

[3]Note that these are methods that are also built into all Scala objects. They are distinguished in OptiML by the "L" character suffix. This discrepenancy will be addressed in future versions of the Scala virtualized compiler.

operation	description
+	addition
-	subtraction
	multiplication
/	division
abs	absolute value
exp	base e exponentiation

Table 2: Numeric operations

4.1.2 Boolean

The `Boolean` type supports:

operation	description
!	unary negation
&&	logical and
||	logical or

Table 3: Boolean operations

4.1.3 String

The `String` type supports:

operation	description
+	string concatentation
trim	remove whitespace
split	returns an array of strings

Table 4: String operations

4.1.4 Tuple

`Tuple` can be used to pack multiple values into a single object, and arithmetic on Tuples is supported if all of its members also support arithmetic. Below we show an example of using a tuple:

```
// pack
val x = (DenseVector.zeros(10), DenseVector.ones(10))
// extract one member
val a = x._1 * 10
// member-wise arithmetic on the tuple
val b = x*2
// extract both of the new result
val (zero, twos) = b
```

4.1.5 Function

The `Function` type allows using functions as values. For example,

```
val newVector: Rep[Int] => Rep[DenseVector[Int]] = n => DenseVector[Int](n)
val vecFive = newVector(5)
```

4.2 Vector

`Vector` represents a generic, linear collection in OptiML. When a `Vector` is used with a type that supports arithmetic operations (e.g. Double), it is considered a mathematical vector, and possesses the corresponding arithmetic operations (see section **??**). OptiML distinguishes `Vector` from `Matrix` to maintain additional type safety, readability, and optimization opportunities.

There are a number of static operations available for `Vector`, listed below. These operations do not require a `Vector` instance, and are used by calling `Vector.<method>`.

operation	description
Vector(a,b,c,..)	construct a new vector containing elements a,b,c,..
Vector(len)	construct a new empty vector of initial size len
ones(len)	return a vector of all values 1.0
zeros(len)	return a vector of all values 0.0
rand(len)	return a vector of random double values
range(start,end,stride)	return a RangeVector containing values from start to end at stride intervals
uniform(start,end,stride)	return a Vector containing double values from start to end at stride intervals

Table 5: Vector static operations

A full description of the operations available on `Vector` instances follows:

operation	description
toBoolean	converts v to boolean if an element-wise conversion exists
toDouble	converts v to double if an element-wise conversion exists
toFloat	converts v to float if an element-wise conversion exists
toInt	converts v to int if an element-wise conversion exists
v(i)	returns elements with indices in i
v(i) = x	set elements in i to value x
length	returns the length of v
isEmpty	true if v is empty
first	returns the first element of v
last	returns the last element of v
indices	IndexVector from 0 to length
drop(c)	returns new vector without the first c elements of v
take(c)	returns new vector with only the first c elements of v
slice(s,e)	elements from s to e
contains(y)	true if y is in v
distinct	returns new vector with only unique elements in v
t	transpose
mt	in-place transpose (modifies v)
Clone	return an immutable deep-copy
mutable	return a mutable deep-copy
pprint	pretty-print v
replicate(i,j)	matrix with i,j tiles of v
mkString(sep)	return a string formed by joining all the elements in v with the 'sep' string
++	append element or vector
+=	point-wise addition in place
+	point-wise addition
*	point-wise multiplication
-	point-wise subtraction
/	point-wise division
**	outer product
.	dot product
sum	sum of all elements in v
abs	point-wise absolute value
exp	point-wise exponentiation
sort	sorted vector v
min	return minimum value in v
max	return maximum value in v
:>(v2)	return boolean vector representing point-wise > comparison of v and v2
:<(v2)	return boolean vector representing point-wise < comparison of v and v2
map(f)	transform v with a function $f^{\dagger}$
mmap(f)	transform v in place with a function $f^{\dagger}$
zip(v2,f)	return a new vector as the result of applying f pair-wise to every element in v and $v2^{\dagger}$
reduce(f)	flatten v with a reducing function $f^{\dagger}$
filter(f)	filter v with a comparator function $f^{\dagger}$
count(p)	return the number of elements in v matching the predicate $p^{\dagger}$
flatMap(f)	map each element in v to a Vector using f and concatenate the results†
partition(p)	split v into two vectors based on a predicate $p^{\dagger}$

Table 6: Vector class operations

†Must be restricted functions (see Section 3)

4.2.1 DenseVector

The default `Vector` in OptiML. The DenseVector object provides the following additional methods:

operation	description
flatten(vectors)	concatenates a vector of vectors into a single vector

Table 7: DenseVector static operations

4.2.2 RangeVector

A `Vector` that does not store any actual data, but represents a consecutive range of integers. `RangeVector` is read-only.

4.2.3 IndexVector

`IndexVector` is a `Vector` of non-negative integers representing indices. An `IndexVector` can represent either a contiguous range (in which case it can be instantiated using the syntax `(0::n)`, or a discrete set of indices (in which case it can be instantiated using the syntax `IndexVector(v)`, where `v` can represent either a real `Vector` or a list of non-negative integers, e.g. `1,7,32,5`. Certain `Vector` operations can also return an `IndexVector`, e.g. `find`.

An `IndexVector` can be used to bulk index into another `Vector` or `Matrix`, either returning multiple or updating multiple elements at a time.

`IndexVector` supports the key operations of `Vector` and `Matrix` *construction*, which we consider next.

Vector construction

The *Vector constructor* operation allows the user to construct a new `Vector` from a set of continuous indices by supplying a restricted function that maps an index to a new value. We illustrate the use of vector constructor through a simple example:

```
// returns a DenseVector[Int] containing values (0,2,4,6,..)
val x = (0::100) { i => i*2 }
// returns a DenseVector[String] containing values ("1","2",...)
val y = (0::100) { i => i.toStringL }
```

Matrix construction

Matrix constructor is the 2-dimensional version of *vector constructor*. The function supplied to it must also be a restricted function. Matrix constructor can be invoked by supplying either scalars (every value of the underlying `Matrix`), or vectors (every row of the underlying `Matrix`). We demonstrate each use case below:

```
/*
   Matrix construction from scalars
   returns m x n result
     [0, 0, 0, 0, ...]
     [0, 1, 2, 3, ...]
     [0, 2, 4, 6, ...]
           .
           .
*/

(0::m, 0::n) { (i,j) =>
  i*j
}

/*
   Matrix construction from row vectors
   returns m x 4 result
    [0, 1, 2, 3]
    [0, 1, 2, 3]
          .
          .
*/

(0::m, *) { i =>
  DenseVector(0,1,2,3)
}
```

4.2.4 VectorView

A `VectorView` represents a *view* of an underlying `Vector` or `Matrix`. It is obtained by using a *slice* operator and provides a view of the underlying data as a contiguous `Vector`. Updates to a `VectorView` propagate back to the underlying data structure the view is based on.

4.2.5 Vertices

A `DenseVector` of `Vertex` instances. Supports a structured form of accessing neighbors during a parallel iteration, as described in section **??**. Returned when accessing the `vertices` field of a `Graph` instance.

4.3 Matrix

OptiML's `Matrix` is a standard row-major, 2-dimensional Matrix. Like `Vector`, it is also polymorphic and can contain elements of any type, but provide arithmetic operations when used with arithmetic types. Below are the operations supported by `Matrix`[4]:

There are also a number of static operations available for `Matrix`, listed below. These operations do not require a `Matrix` instance, and are used by calling `Matrix.<method>`.

[4]Because OptiML's Matrix is row-major, some operations of Matrix are only available on rows for efficiency. This forces the user to transpose, rather than inadvertantly suffer major performance losses.

operation	description
Matrix(Vector(a,b,c), Vector(d,e,f), ..)	construct a new m x 3 matrix containing elements ((a,b,c),(d,e,f),..)
Matrix(m, n)	construct a new m x n matrix
diag(w, vals)	construct a new w x w diagonal matrix with values from vals on the diagonal
identity(w)	construct a w x w identity matrix
zeros(m,n)	return a matrix of all values 0.0
ones(m,n)	return a matrix of all values 1.0
rand(len)	return a matrix of random double values

Table 8: Matrix static operations

Operations available on a `Matrix` instance are:

operation	description
toBoolean	converts m to boolean if an element-wise conversion exists
toDouble	converts m to double if an element-wise conversion exists
toFloat	converts m to float if an element-wise conversion exists
toInt	converts m to int if an element-wise conversion exists
m(i,j)	return element at (i,j)
m(i,j) = x	set element at (i,j)
m(i)	return row vector at i
m(i) = x	set row vector at i
getRow(i)	return row vector at i
getCol(j)	return col vector at j
vview(start,stride,len)	return a vector view of the matrix containing values every stride from start to length
slice(i0,i1,j0,j1)	submatrix from (i0-i1),(j0-j1)
sliceRows(s,e)	submatrix containing rows from s to e
numRows	return number of rows of m
numCols	return number of cols of m
t	transpose
Clone	return an immutable deep-copy
mutable	return a mutable deep-copy
pprint	pretty-print m
replicate(i,j)	matrix with i,j tiles of m
+=	append vector
++=	append matrix
insertRow(n,x)	insert vector x at row n
insertAllRows(n,x)	insert matrix x at row n
insertCol(n,x)	insert vector x at col n
insertAllCols(n,x)	insert matrix x at col n
removeRow(n)	remove nth row
removeRows(n,len)	remove len rows starting at n
removeCol(n)	remove nth col
removeCols(n,len)	remove len cols starting at n
+	point-wise addition
-	point-wise substraction
:	point-wise multiplication
/	point-wise division
*	matrix multiply
inv	matrix inverse
sum	sum of all elements in m
abs	point-wise absolute value
exp	point-wise exponentiation
sigmoid	point-wise sigmoid function
min	return min element
minRow	return row with minimum total value
max	return max element
maxRow	return row with maximum total value
:>(m2)	return boolean matrix representing point-wise > comparison of m and m2
:<(m2)	return boolean matrix representing point-wise < comparison of m and m2
map(f)	transform each element in m†
mmap(f)	transform m in place with a function f†
zip(m2,f)	return a new matrix as the result of applying f pair-wise to every element in m and m2†
mapRows(f)	transform each row in m†
reduceRows(f)	flatten m with a reducing function†
filterRows(f)	filter m with a comparator function†
count(p)	return the number of elements in m matching the predicate p†

Table 9: Matrix class operations

†Must be restricted functions (see Section 3)

4.3.1 DenseMatrix

The default `Matrix` in OptiML.

4.3.2 Image

`Image` extends `DenseMatrix` to provide common operators over images:

operation	description
downsample	return a new image by converting blocks of the original image to a single value
windowedFilter	return a new image computed by running a rectangular windowed filter over the original image
convolve(kernel)	computes a new image where each pixel is the sum of the pointwise multiplication of the given kernel and a window of the original image

Table 10: Image operations

4.3.3 TrainingSet

`TrainingSet` is a wrapper around `Matrix`. `TrainingSet` is immutable, and is the expected input for most algorithms in the OptiML standard library. `SupervisedTrainingSet` stores the data matrix along with a vector containing training labels, while `UnsupervisedTrainingSet` stores only the data matrix.

Although currently `TrainingSet` is mainly a convenience data structure in OptiML, the goal is to eventually have it provide more useful built-in machine learning primitives (e.g. ensembles), and to use the type information to inform device scheduling and data decomposition decisions.

4.4 Stream

`Streams` are meant to hold or compute a very large amount of data (more than can fit in memory). They are immutable, and only allow streaming access (i.e. next, prev) as well as bulk operators (e.g. foreach). `Stream` only keeps a chunk of the actual underlying data in memory at a time, and transparently computes new chunks when required. `Streams` currently support the following operations:

operation	description
Stream(m, n, func)	construct a new m x n stream, where func computes the (ith,jth) element

Table 11: Stream static operations

operation	description
numRows	return number of rows of m
numCols	return number of cols of m
isPure	true if the stream initialization function is pure

Table 12: Stream class operations

4.5 Graph

`Graph` allows statistical inference problems on graphical models to be naturally expressed in OptiML. It provides a typical representation containing a set of `Vertices` and `Edges`. `Vertices` and `Edges` can be iterated over with built-in synchronization, as described in section 5. `Graph` is designed for message passing; `Vertices` can store data, and `Edges` can store both incoming and outgoing data.

operation	description
Graph()	construct a new graph instance

Table 13: Graph static operations

operation	description
vertices	vector containing vertices of g
edges	vector containing edges of g
neighborsOf(v)	vector containing neighbors of vertex v
edgesOf(v)	vector containing edges of vertex v
addVertex(v)	add a vertex v to g
addEdge(e)	add an edge e to g
freeze	finalize the graph structure

Table 14: Graph class operations

4.5.1 Vertex

`Vertex` is represents a node in a `Graph`. A `Vertex` can store data of any OptiML type (or user-defined struct containing OptiML types).

operation	description
Vertex(g, d)	construct a new Vertex on graph g storing data d

Table 15: Vertex static operations

operation	description
edges	return the set of edges connected to this vertex
neighbors	return the set of neighboring vertices
data	return the data stored at this vertex
graph	return the graph associated with this vertex

Table 16: Vertex class operations

4.5.2 Edge

`Edge` represents an edge in a `Graph`. An `Edge` can store both ingoing and outgoing data of any OptiML type (or user-defined struct containing OptiML types).

operation	description
Stream(m, n, func)	construct a new m x n stream, where func computes the (ith,jth) element

Table 17: Edge static operations

operation	description
inData	return the data stored as an incoming message
outData	return the data stored as an outgoing message
v1	return the first vertex connected to this edge
v2	return the second vertex connected to this edge
graph	return the graph associated with this edge

Table 18: Edge class operations

4.6 User-defined types

OptiML allows *struct-like* data structures to be defined by users. These data structures must only contain fields (cannot contain any methods). A user-defined type can be declared and used by defining a new `Record`:

```
  val MyType = new Record {
    val x: Int
    val y: String
    val z: DenseVector[Int]
  }
  type RT = Record{x: Int, y: String, z: DenseVector[Int]}

  val v = DenseVector[RT](100)
}
```

5 Control structures

5.1 Iterating with 'for'

It is possible to iterate over the elements of an OptiML data structure using the the special *for* iterator. Each iterator accepts any restricted function, but in certain cases, the restrictions are relaxed to allow structured communication. The access patterns are described below:

operation	access pattern
for (e <- Vector)	cannot write to any other elements in Vector
for (v <- Vertices)	can only write to neighboring vertices, e.g. v.neighbors
for (e <- Matrix)	cannot write to any other elements in Matrix
for (r <- Matrix.rows)	cannot write to any other rows in Matrix
for (p <- Image)	can only write to pixels in a local window, e.g. p.window
for (r <- Stream.rows)	cannot access any other rows in Stream

Table 19: For operations

Note that each of these operations are syntactic sugar for the corresponding `foreach` methods defined on each data type.

5.2 untilconverged

`untilconverged` is an iterative control structure that iterates until reaching a convergence criterion. Each iteration of `untilconverged` produces a value, and the loop converges when the difference between values in consecutive iterations falls below a supplied threshold parameter, or a maximum number of iterations is reached.

An example of using `untilconverged` is shown below:

```
val x = Matrix.ones(100,100) * 1000
// default matrix difference is absolute difference
untilconverged(x, .01) { m =>
  m / 100
}
```

5.3 gradient

`gradient` is a specialized version of `untilconverged` that implements the gradient descent algorith for exponential family models. Specifically, it provides `batch` and `stochastic` variants (the default is `batch`).

The `batch` variant has the following signature:

```
batch(x: Rep[TrainingSet[Double]], alpha: Rep[Double], thresh: Rep[Double], maxIter: Rep[Int])
     (hyp: Rep[Vector[Double]] => Rep[Double]): Rep[Double]
```

where `alpha` is the learning rate, `thresh` is the convergence threshold, `maxIter` is the maximum number of iterations, and `hyp` is the hypothesis function that maps a training sample to a scalar value. `batch` implements the following pseudocode:

```
block() updates each jth parameter from the sum of all ith training samples
while not converged{
  for j from 0 until n
  j_update = sum((y(i) - h(x(i))*x(j,i)
  updatej(j_update)
```

The `stochastic` variant has the same signature as `batch`, but implements the following pseudocode:

```
block() updates every jth parameter for every ith training sample
while not converged{
  for i from 0 until m
    for j from 0 until n
      updatej(i,j)
```

6 Built-in functions

OptiML supplies a number of built-in functions that are useful for statistics and machine learning.

In addition to the functions listed below, the following standard mathematical operators are available:

- sqrt, ceil, floor, exp, log, pow, sin, cos, acos, atan, atan2

as well as the following standard mathematical constants:

- Pi, E

6.1 sum

6.2 aggregate

6.3 min

6.4 max

6.5 abs

6.6 dist

6.7 sample

6.8 nearestNeighborIndex

6.9 tic/toc

6.10 random

7 I/O

OptiML currently supports reading and writing files as a tab-delimited set of values, where matrices are represented by multiple lines (this is the same default format as MATLAB).

The functions to read and write files are:

- `readMatrix(filename)`
- `readVector(filename)`
- `readImage(filename)` (values must be integers currently)

8 Advanced Topics

8.1 Type system

OptiML inherits its type system from Scala, and any Scala type annotations may be used. However, to use these features, the user must keep in mind that all OptiML types are wrapped in the *Rep* type. For example, one could specify two illustrative type-bounds on methods as follows:

```
def foo[A <: Double](x: Rep[Vector[A]]) { ... }
def bar[B <: Rep[Vector[Double]]](x: B) { ... }
```

Of course, using advanced type signatures is not required. In most cases, Scala's type inference allows OptiML result types to be inferred with no type declaration necessary.

A detailed description of Scala's type system can be found at **??**.

8.2 Pass by value

OptiML also follows the Scala/Java pass by value semantics, where objects are *passed by the value of their reference*, i.e., an immutable reference to the object. Therefore, accessing an object element in OptiML data structures returns a the reference to the object itself (not a copy); in a `Vector[Foo]`, returning `v(0)` will return a reference to the `Foo` object at element 0. Similarly, accessing row `i` of a Matrix using `m(i)` returns a reference to the `RowVector` at that position. Accessing a primitive object, e.g. `v(0)` in a `Vector[Double]` will return the value of the double of at `v(0)`.

8.3 Mutable/Immutable operators

Almost all OptiML operations on a data structure return an entirely new data structure. For example, `v + 5` if `v` is a `Vector` will not modify `v`, but will return an entirely new `Vector`. There are a small number of operations that *do* make in-place modifications which are noted in section 4.

Although a naive implementation of these semantics could result in a large number of unnecessary temporary allocations, the fusion optimization described in section **??** aggressively eliminates these when possible.

9 Library

The OptiML standard library is small, but growing. The goal is to eventually contain a number of off-the-shelf implementations of common machine learning algorithms, written in OptiML, that are useful for

someone writing applications that need to leverage machine learning algorithms (as opposed to designing their own algorithm).

Currently available library functions are `linear regression` and `k-means clustering`. These functions can be found in the `ppl.dsl.optiml.library` package, and example applications using them can be found in the `ppl.apps.ml` package.

References