Mahout Scala Bindings and Mahout Spark Bindings for Linear Algebra Subroutines

Working Notes and Manual

Dmitriy Lyubimov*

Abstract

In recent years significant effort was spent to produce semantically friendly environments for linear algebra. Working with vector, matrix and tensor data structures as a single data type offers essential qualities necessary for rapid prototyping of algebraically defined mathematical problems. The other side of the coin is convenience of the same environment as a programming language. Yet another one is doing things at scale. Yet another highly desirable capability of the same environment is plotting and visualization. Without bringing any detailing review of existing environments here, the author however offers an opinion that while a lot of environments succeed in one or more of these aspects, however none of them adequately addresses all of them at the same time and at reasonable cost.

Unlike many other environments, Mahout model was targeting both dense and sparse data structures from the very beginning both in type modeling and cost-based optimized computations.

In this work we are trying to bring semantic explicitness to Mahout's in-core and out-of-core linear algebra subroutines, while adding benefits of strong programming environments of scala, and extending great scalability benefits of Spark and GraphX.

1 Overview

The manual is mostly organized giving DSL features by example. That means that capabilities are wider than necessarily given here, and may change behind the scenes as the work develops. However, the author tries to facilitate and ecourage particular style given in this document as most explicit.

If a matrix or a vector are denoted by a single Latin letter, I use capital letters to denote matrices, and small letters to denote vectors, thus diverging somewhat from accepted camel case for reference variables in these few cases.

2 Mahout in-core Algebra Scala Bindings[1]

In-core DSL is hardly much more than just a syntactic sugar over `org.apache.mahout.math.Matrix`(`Vector`) trait implementations. As such, all originally implemented operation signatures of Mahout are also

*dlyubimov at apache dot org

[1]See link: original proposal.

retained.

2.1 Imports

The following two scala imports are typically used to enable Mahout Scala DSL bindings for Linear Algebra:

```
import org.apache.mahout.math.scalabindings._
import RLikeOps._
```

Another option is to use "matlab like" dialect by doing

```
import MatlabLikeOps._
```

However, Matlab-like DSL dialect adherence to original Matlab dialect is far less optimal that R dialect due to the specifics of operator support in scala, so we just will limit ourselves to R-like dialect here.

2.2 Inline initialization

Dense vectors

```
val denseVec1: Vector = (1.0, 1.1, 1.2)
val denseVec2 = dvec(1, 0, 1.1, 1.2)
```

Sparse vectors

```
val sparseVec = svec((5 -> 1) :: (10 -> 2.0) :: Nil)
val sparseVec2: Vector = (5 -> 1.0) :: (10 -> 2.0) :: Nil
```

matrix inline initialization, either dense or sparse, is always row-wise:
dense matrices :

```
val A = dense((1, 2, 3), (3, 4, 5))
```

sparse matrices

```
val A = sparse(
    (1, 3) :: Nil,
    (0, 2) :: (1, 2.5) :: Nil
)
```

diagonal matrix with constant diagonal elements

```
diag(10, 3.5)
```

diagonal matrix with main diagonal backed by a vector

```
diagv((1, 2, 3, 4, 5))
```

Identity matrix

```
eye(10)
```

Obviously, direct initialization of any vector or matrix type in Mahout is still available with regular oeration `new`.

2.3 Slicing and Assigning

geting vector element

```
val d = vec(5)
```

setting vector element

```
vec(5) = 3.0
```

getting matrix element

```
val d = m(3,5)
```

setting matrix element (setQuick() behind the scenes)

```
M(3,5) = 3.0
```

Getting matrix row or column

```
val rowVec = M(3, ::)
val colVec = M(::, 3)
```

Setting matrix row or column

```
M(3, ::) = (1, 2, 3)
M(::, 3) = (1, 2, 3)
```

thru vector assignment also works

```
M(3, ::) := (1, 2, 3)
M(::, 3) := (1, 2, 3)
```

subslices of row or vector work too

```
a(0, 0 to 1) = (3, 5)
```

or with vector assignment

```
a(0, 0 to 1) := (3, 5)
```

matrix contiguous block as matrix, with assignment
// block

```
val B = A(2 to 3, 3 to 4)
```

// asignment to a block

```
A(0 to 1, 1 to 2) = dense((3, 2), (2, 3))
```

or thru the matrix assignment operator

```
A(0 to 1, 1 to 2) := dense((3, 2), (2, 3))
```

Assignment operator by copying between vectors or matrix

```
vec1 := vec2
M1 := M2
```

also works for matrix subindexing notations as per above

Assignment thru a function literal (matrix)

```
M := ((row, col, x) => if (row == col) 1 else 0)
```

for a vector, the same:

```
vec := ((index, x) => sqrt(x))
```

2.4 BLAS-like operations

plus/minus, either vector or matrix or numeric, with assignment or not

```
a + b
a - b
a + 5.0
a - 5.0
```

Hadamard (elementwise) product, either vector or matrix or numeric operands

```
a * b
a * 5
```

same things with assignment, matrix, vector or numeric operands

```
a += b
a -= b
a += 5.0
a -= 5.0
a *= b
a *= 5
```

One nuance here is associativity rules in scala. E.g. `1/x` in R (where x is vector or matrix) is elementwise inverse operation and in scala realm would be expressed as

```
val xInv = 1 /: x
```

and R's `5.0 - x` would be

```
val x1 = 5.0 -: x
```

Even trickier and really probably not so obvious stuff :

```
a -=: b
```

assigns `a - b` to `b` (in-place) and returns `b`. Similarly for `a /=: b` or `1 /=: v`.

(all assignment operations, including :=, return the assignee argument just like in C++)

dot product (vector operands)

```
a dot b
```

matrix /vector equivalency (or non-equivalency). Dangerous, exact equivalence is rarely useful, better use norm comparisons with admission of small errors

```
a === b
a !== b
```

Matrix multiplication (matrix operands)

```
a %*% b
```

for matrices that explicitly support optimized right and left muliply (currently, diagonal matrices)

right-multiply (for symmetry, in fact same as %*%)

```
diag(5,5) :%*% b
```

optimized left multiply with a diagonal matrix:

```
A %*%: diag(5,5) # i.e. same as (diag(5,5) :%*% A.t) t
```

Second norm, vector or matrix argument:

```
a.norm
```

Finally, transpose

```
val Mt = M.t
```

Note: Transposition currently is handled via *view*, i.e. updating a transposed matrix will be updating the original. Also computing something like $\mathbf{X}^\top\mathbf{X}$

```
val XtX = X.t %*% X
```

will not therefore incur any additional data copying.

2.5 Decompositions

All arguments in the following are matrices.

Cholesky decompositon (as an object of a CholeskyDecomposition class with all its operations)

```
val ch = chol(M)
```

SVD

```
val (U, V, s) = svd(M)
```

EigenDecomposition

```
val (V, d) = eigen(M)
```

QR decomposition

```
val (Q, R) = qr(M)
```

Rank Check for rank deficiency (runs rank-revealing QR)

```
M.isFullRank
```

In-core SSVD

```
val (U, V, s) = ssvd(A, k=50, p=15, q=1)
```

2.6 Misc

vector cardinality

```
a.length
```

matrix cardinality

```
m.nrow
m.ncol
```

a copy-by-value (vector or matrix)

```
val b = a cloned
```

2.7 Pitfalls

This one the people who are accustomed to writing R linear algebra will probably find quite easy to fall into. R has a nice property, a copy-on-write, so all variables actually appear to act as no-side-effects scalar-like values and all assignment appear to be by value. Since scala always assigns by reference (except for AnyVal types which are assigned by value), it is easy to fall prey to the following side effect modifications

```
val m1 = m
m1 += 5.0 // modifies m as well
```

A fix is as follows:

```
val m1 = m cloned
m1 += 5.0 // now m is intact
```

3 MAHOUT-1346: Spark Bindings (DRM)