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 detailed 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, none of them however adequately addresses all of them at the same time and at a 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 environment of scala, and captializing on great scalability benefits of Spark and GraphX.

Overview

The manual is mostly origanised by giving DSL features by example. That means that capabilities are wider than those shown, and may change behind the scenes as the work develops. However, the authors try to facilitate and ecourage particular style given, and retain behind-the-scenes compatibility with the examples given.

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.

[*]dlyubimov at apache dot org

Contents

1 Mahout in-core algebraic 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 retained.

1.1 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.

1.2 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
)
```

[1]See link: original proposal.

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`.

1.3 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))
```

1.4 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.

1.5 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)
```

1.6 Misc

vector cardinality

```
a.length
```

matrix cardinality

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

a copy-by-value (vector or matrix)

```
val b = a cloned
```

1.7 Bringing it all together: in-core SSVD

Just to illustrate semantic clarity, we will adduce a source for in-core SSVD code.

```
/**
 * In-core SSVD algorithm.
 *
 * @param a input matrix A
 * @param k request SSVD rank
 * @param p oversampling parameter
 * @param q number of power iterations
 * @return (U,V,s)
 */
def ssvd(a: Matrix, k: Int, p: Int = 15, q: Int = 0) = {
  val m = a.nrow
  val n = a.ncol
  if (k > min(m, n))
    throw new IllegalArgumentException(
      "k cannot be greater than smaller of m,n")
  val pfxed = min(p, min(m, n) - k)
  // actual decomposition rank
  val r = k + pfxed
  val rnd = RandomUtils.getRandom
  val omega = Matrices.symmetricUniformView(n, r, rnd.nextInt)
  var y = a %*% omega
  var yty = y.t %*% y
  val at = a.t
  var ch = chol(yty)
```

```
    var bt = ch.solveRight(at %*% y)
    // power iterations
    for (i <- 0 until q) {
      y = a %*% bt
      yty = y.t %*% y
      ch = chol(yty)
      bt = ch.solveRight(at %*% y)
    }
    val bbt = bt.t %*% bt
    val (uhat, d) = eigen(bbt)
    val s = d.sqrt
    val u = ch.solveRight(y) %*% uhat
    val v = bt %*% (uhat %*%: diagv(1 /: s))
    (u(::, 0 until k), v(::, 0 until k), s(0 until k))
  }
```

1.8 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
```

2 Out-of-core linalg bindings

The subject of this section are solely operations applicable to Mahout's DRM (distributed row matrix). Once loaded into spark, DRM is represented by Spark partitions initially consisting of handful of row vectors.

Here and on, I will denote spark-backed DRM references as `A`, whereas in-core matrices as `inCoreA`.

2.1 Initializing Mahout/Spark context

Many (if not all) operations will require a Spark context. Spark context can be passed in two ways: (1) as an implicit value; and as passed down from a parent source (DRM's backing RDD).

To initialize Mahout/Spark session, just create an implicit value of a specifically prepped Spark context:

```
import org.apache.mahout.sparkbindings._
implicit val mahoutCtx = mahoutSparkContext(

    masterUrl = "local",
    appName = "MahoutLocalContext"
    // [,...]

)
```

Parameter `masterUrl` points to Spark's master. Note that Mahout expects either MAHOUT_HOME environment or -Dmahout.home=... java system variable to point to Mahout home directory in order to collect relevant jars for the Spark sessions.

From there on, as long as Mahout-initialized Spark context is exposed thru implicit variable, attribute or paremeter, there's no need to specify it explicitly for any of the successive operations.

Note that as of the time of this writing Spark sessions cannot coexist in the same jvm, even though a single spark session is reentrant and can handle requests from more than one thread.

2.2 DRM Persistence operators

```
import org.apache.mahout.sparkbindings.drm._
import RLikeDrmOps._
```

2.2.1 Loading DRM off HDFS

```
val A = drmFromHDFS(path = hdfsPath)
```

2.2.2 Parallelizing from an in-core matrix

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

2.2.3 Empty DRM

```
val A = drmParallelizeEmpty(100, 50)
```

2.2.4 Collecting to driver's jvm in-core

```
val inCoreA = A.collect()
```

2.2.5 Collecting to HDFS

Collect Spark-backed DRM to HDFS in Mahout's DRM format files:[2]

```
A.writeDRM(path = hdfsPath)
```

2.3 Logical algebraic operators on DRM matrices

We will define a logical set of operators that are familiar to users of environments such as R, which are elementwise +, -, *, / as well as matrix multiplication %*% and transposition. General rule is that we try to do a subset of those enabled for in-core DSL. In particular, since all distributed matrices are immutable, there are no assignment versions (e.g. `A += B`). We also probably will have trouble to do efficient slicing.

Logical operators comprised into expression do not however mean that concrete physical plan is materialized until the expression is "checkpointed" – directly or indirectly. In terms of Spark, this is called "action".

Unlike with Spark, we want to discern two types of "actions": optimizer action and computational action.

Optimizer actions. Optimizer action triggers materialization of a physical plan (concrete RDD graph with result marked for Spark caching), backed by CheckpointedDRM. CheckpointedDRM servies as a cut-off boundary for optmizer action. Optimizer action does not trigger actual computation of result data set. Right now optimizer action is triggered explicitly by DRMLike#checkpoint().

Let consider two examples:

```
val A = drmParallelize (...)
val B = drmParallelize (...)
val C = A %*% B
val D = C.t
val E = C.t %*% C
```

In this example, optimizer optimizes separately 2 pipelines: $\mathbf{D} = \mathbf{AB}^\top$ and $\mathbf{E} = \left(\mathbf{AB}^\top\right)^\top \left(\mathbf{AB}^\top\right)$ using same matrices $\mathbf{A}$ and $\mathbf{B}$ as root of both computations. Now let's consider the following modified example:

```
val A = drmParallelize (...)
val B = drmParallelize (...)
val C = (A %*% B).checkpoint
val D = C.t
val E = C.t %*% C
```

[2]if you see an error here along the lines "no implicit view available from A => org.apache.hadoop.io.Writable" most likely you need just to import SparkContext._.

In this case, optimizer considers 3 separate pipelines: $\mathbf{C}=\mathbf{AB}$, $\mathbf{D}=\mathbf{C}^{\top}$ and $\mathbf{E}=\mathbf{C}^{\top}\mathbf{C}$ while caching optimized plan and intermediate result for $\mathbf{C}$ into the Spark cache. Introducing checkpoints may improve "wall time" (since matrices $\mathbf{D}$ and $\mathbf{E}$ will be triggered for action at different time and optimizer wouldn't be able to consider computational graph that includes both at the same time). However, in this particular case the checkpoint might be better considered for $\mathbf{AB}^{\top}$ since at this point optimizer doesn't compute $\mathbf{AB}$ directly but rather as $\mathbf{A}\left(\mathbf{B}^{\top}\right)^{\top}$, i.e. this is an example when checkpointing will result in one extra transposition of arguments. However, even in the first example optimizer will be able to figure to optimize $\mathbf{E}=\left(\mathbf{AB}^{\top}\right)^{\top}\left(\mathbf{AB}^{\top}\right)$ as $\text{t\_square}\left(\text{product}\left(\mathbf{A},\mathbf{B}^{\top}\right)\right)$ pipeline, i.e. into only two sequential physical operators.

In either of the examples, nothing happens in the backend until a computational action is triggered for either of $\mathbf{E}$ or $\mathbf{D}$.

Computational actions. Computational action leads to result being computed and (optionally?) placed into Spark cache. Such actions will also lazily and implicitly trigger linalg optimizer checkpointing. Currently, computational actions include writeDrm(), collect(), blockify() and sometimes could also be triggered implicitly by optimizer activity beyond current checkpoint cut-off (if checkpointed but not computed and cached yet) to run some cost estimates necessary for the optimizer beyond checkpointing, potentially future actions associated with DRM sub-blocking.

E.g. in the second example, running

```
E.writeDrm(path)
```

will trigger computational actions for $\mathbf{E}$ and, implicitly, for $\mathbf{C}$.

All these rules follow the same patterns as for the in-core arguments.

2.3.1 Transposition

```
A.t
```

2.3.2 Elementwise +, -, *, /

$$\mathbf{M}=\mathbf{A}+\mathbf{B}$$

$$\mathbf{M}=\mathbf{A}-\mathbf{B}$$

$$\mathbf{M}=\mathbf{A}\circ\mathbf{B}\ \ \text{(Hadamard)}$$

$$\mathbf{M}=\begin{pmatrix}\frac{a_{11}}{b_{11}} & \frac{a_{12}}{b_{12}} & \cdots & \frac{a_{1n}}{b_{1n}}\\ \vdots & \vdots & \ddots & \vdots\\ \frac{a_{m1}}{b_{m1}} & \frac{a_{m2}}{b_{m2}} & \cdots & \frac{a_{mn}}{b_{mn}}\end{pmatrix}\ \text{(elementwise deletion)}$$

All this operations require identical geometry of operands and row keying types that will be asserted at optmizer checkpointing time.

```
A + B
A - B
A * B
A / B
```

Binary operators involving in-core argument (only on int-keyed DRMs)

```
A + inCoreB
A - inCoreB
A * inCoreB
A / inCoreB
A :+ inCoreB
A :- inCoreB
A :* inCoreB
A :/ inCoreB
inCoreA +: B
inCoreA -: B
inCoreA *: B
inCoreA /: B
```

Note spark associativity change (e.g. inCoreA +: B means B.leftMultiply(A), just like with both in-core arguments). Important thing here is that whenever operator arguments include both in-core and out-of-core arguments, operator can only be associated with the out-of-core argument to support distributed implementation.

2.3.3 Matrix-matrix multiplication %*%

$\mathbf{M} = \mathbf{AB}$

```
A %*% B
A %*% inCoreB
A :%*% inCoreB
inCoreA %*%: B
```

Same as above, when both in-core and out-of-core argumetns used, associativity of operation must follow the out-of-core (DRM) argument in the expression.

2.3.4 Matrix-scalar +,-,*,/

In this context, matrix-scalar operations mean element-wise operatios of every matrix element and a scalar.

```
A + 5.0
A - 5.0
A :- 5.0
5.0 -: A
A * 5.0
A / 5.0
5.0 /: A
```

Note that `5.0 -: A` means $m_{ij} = 5 - a_{ij}$ and `5.0 /: A` means $m_{ij} = \frac{5}{a_{ij}}$ for all elements of the result.

2.4 Slicing

Slicing (without assigning) is supported mostly identically to in-core slicing. Slicing row or range is of `Range` scala type, which typically can be inlined as `x to y` or `x until y`. All-range is given by `::`.

General slice

```
A(100 to 200, 100 to 200)
```

Vertical block

```
A(::, 100 to 200)
```

Horizontal block

```
A(100 to 200, ::)
```

Note: if row range is not all-range (`::`) then the DRM must be `Int`-keyed. Row slicing in general case is not supported for key types other than `Int`.

2.5 Custom pipelines on blocks

Pretty often there's a need to do something with the matrix expressed as blocks. Some physical operators are also more effective once working with matrix blocks rather than individual rows. Internally, Mahout's matrix pipeline (lazily) blockifies every data partition into `BlockifiedDrmTuple` blocks whenever first physical operator requiring blocking is encountered. After that, any row-wise physical operators work on row vector *views* of the blocks.

Here is definition for DRM block tuple type:

```
/** Drm block-wise tuple:
  Array of row keys and the matrix block. */
type BlockifiedDrmTuple[K] = (Array[K], _ <: Matrix)
```

DRM operator `mapBlock` provides transformational access to the vertical blockified tuples of the matrix. (Current implementation also guarantees that there's exactly one block per map task).

Here is unit test that demonstrates use of `mapBlock` operator by producing A + 1.0:

```
val inCoreA = dense((1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6))
val A = drmParallelize(m = inCoreA, numPartitions = 2)
val B = A.mapBlock(/* Inherit width */)({
  case (keys, block) => keys -> (block += 1.0)
})
val inCoreB = B.collect
val inCoreBControl = inCoreA + 1.0
println(inCoreB)
// Assert they are the same
(inCoreB - inCoreBControl).norm should be < 1E-10
```

The constrain is that operator mapBlock should not attempt to change the height of the block, in order to provide correct total matrix row count estimate to optimizer after application of the operator. MapBlock operator may change *width* (i.e. column count) of the matrix; if it does so, it needs to supply it to first `ncol` parameter of the mapBlock call. Otherwise, it is assumed operator has inherited the width of the original matrix. The geometry of the block returned is asserted at run time, as geometry is vitally important for the coherence of linear operators.

Another note is that it is ok to return a reference to a modified same in-core block. This is actually recommended whenever possible (note the `+=` operator in the example) to avoid matrix copying.

2.6 Doing something completely custom

If flexibility of Drm api is not enough, it is always possible to exit out of opimizer-based algebra pipeline into pure spark RDD environment. The exit is possible at optimizer checkpoints, which are presented by `CheckpointedDrmBase[K]` trait. This trait has an `rdd:DrmRdd[K]` method, which returns a row-wise RDD of `DrmTuple[K]` type.

The row-wise tuple types and RDDs are defined as following:

```
    /** Drm row-wise tuple */
    type DrmTuple[K] = (K, Vector)
    /** Row-wise organized DRM rdd type */
    type DrmRdd[K] = RDD[DrmTuple[K]]
```

(type `Vector` here is `org.apache.mahout.math.Vector`).

E.g.:

```
val myRdd = (A %*% B).checkpoint().rdd
...
```

Similarly, an Rdd conforming to a type of DrmRdd, can be re-wrapped into optimizer checkpoint via

```
val rdd:DrmRdd[K] = ... //
val A = drmWrap(rdd = rdd, nrow = 50000, ncol = 100)
... // use A in a DRM pipeline
```

Parameters ncol and nrow (geometry) are optional. If not supplied, they will be recomputed off cached dataset. But if supplied, they *must* be accurate!

A note about serialization: the Spark bindings for Mahout support serialization of Vector and Matrix types (including their views and slices) via Kryo serialization. Hence, Spark context for Mahout is initialized with kryo serializer for all objects. This is something to keep in mind (Vector and Matrix objects can be broadcasted/collected, but there's no way to revert to java-serialized-only support in spark session and use Mahout objects at the same time). This generally should not be a problem in Spark 0.9 since there's a kryo serialization back for practically anything of interest in the twitter/chill that is used by Spark since 0.8.

2.7 Distributed Decompositions

2.7.1 Distributed thin QR.

For the classic QR decomposition of the form $\mathbf{A}=\mathbf{QR}$, $\mathbf{A}\in\mathbb{R}^{m\times n}$, a distributed version is fairly easily achieved if $\mathbf{A}$ is tall and thin such that $\mathbf{A}^{\top}\mathbf{A}$ fits in memory, i.e. m is large, but $n\leq\sim 5000$. Under such circumstances, only $\mathbf{A}$ and $\mathbf{Q}$ are distributed matrices, and $\mathbf{A}^{\top}\mathbf{A}$ and $\mathbf{R}$ are in-core products. We just compute in-core version of Cholesky decomposition in the form of $\mathbf{LL}^{\top}=\mathbf{A}^{\top}\mathbf{A}$. After that we take $\mathbf{R}=\mathbf{L}^{\top}$ and $\mathbf{Q}=\mathbf{A}\left(\mathbf{L}^{\top}\right)^{-1}$. The latter is easily achieved by multiplying each vertical block of $\mathbf{A}$ by $\left(\mathbf{L}^{\top}\right)^{-1}$. (There's no actual matrix inversion happening).

Corollary to this design are two facts: (1) rows of $\mathbf{Q}$ retain the same indexing type as rows of $\mathbf{A}$ (not necessarily int-keyed); and (2) $\mathbf{A}$ and $\mathbf{Q}$ are identically partitioned. Therefore, $\mathbf{A}$ and $\mathbf{Q}$ subsequently be trivially zipped together if join of rows is needed (used in d-ssvd).

```
val (drmQ, inCoreR) = dqrThin(drmA)
```

3 Notations

COMMENT

TENTATIVE