package com.twitter.simclusters\_v2.scalding.common.matrix

import com.twitter.algebird.Semigroup

import com.twitter.bijection.Injection

import com.twitter.scalding.{TypedPipe, ValuePipe}

/\*\*

\* A case class that represents a sparse matrix backed by a TypedPipe[(R, C, V)].

\*

\* We assume the input does not have more than one value per (row, col), and all the input values

\* are non-zero.

\*

\* We do not except the input pipe are indexed from 0 to numRows or numCols.

\* The input can be any type (for example, userId/TweetId/Hashtag).

\* We do not convert them to indices, but just use the input as a key to represent the rowId/colId.

\*

\* Example:

\*

\* val a = SparseMatrix(TypedPipe.from(Seq((1,1,1.0), (2,2,2.0), (3,3,3.0))))

\*

\* val b = a.rowL2Normalize // get a new matrix that has unit-norm each row.

\*

\* val c = a.multiplySparseMatrix(b) // multiply another matrix

\*

\* val d = a.transpose // transpose the matrix

\*

\* @param pipe underlying pipe. We assume the input does not have more than one value per (row, col),

\* and all the values are non-zero.

\* @param rowOrd ordering function for row type

\* @param colOrd ordering function for col type

\* @param numericV numeric operations for value type

\* @param semigroupV semigroup for the value type

\* @param rowInj injection function for the row type

\* @param colInj injection function for the col type

\* @tparam R Type for rows

\* @tparam C Type for columns

\* @tparam V Type for elements of the matrix

\*/

case class SparseMatrix[R, C, V](

pipe: TypedPipe[(R, C, V)]

)(

implicit override val rowOrd: Ordering[R],

override val colOrd: Ordering[C],

override val numericV: Numeric[V],

override val semigroupV: Semigroup[V],

override val rowInj: Injection[R, Array[Byte]],

override val colInj: Injection[C, Array[Byte]])

extends TypedPipeMatrix[R, C, V] {

// number of non-zero values in the matrix

override lazy val nnz: ValuePipe[Long] = {

this.filter((\_, \_, v) => v != numericV.zero).pipe.map(\_ => 1L).sum

}

// number of non-zero values in each row

lazy val rowNnz: TypedPipe[(R, Long)] = {

this.pipe.collect {

case (row, \_, v) if v != numericV.zero =>

row -> 1L

}.sumByKey

}

// get the num of non-zero values for each col.

lazy val colNnz: TypedPipe[(C, Long)] = {

this.transpose.rowNnz

}

override lazy val uniqueRowIds: TypedPipe[R] = {

this.pipe.map(t => t.\_1).distinct

}

override lazy val uniqueColIds: TypedPipe[C] = {

this.pipe.map(t => t.\_2).distinct

}

override def getRow(rowId: R): TypedPipe[(C, V)] = {

this.pipe.collect {

case (i, j, value) if i == rowId =>

j -> value

}

}

override def getCol(colId: C): TypedPipe[(R, V)] = {

this.pipe.collect {

case (i, j, value) if j == colId =>

i -> value

}

}

override def get(rowId: R, colId: C): ValuePipe[V] = {

this.pipe.collect {

case (i, j, value) if i == rowId && j == colId =>

value

}.sum // this assumes the matrix does not have any duplicates

}

// filter the matrix based on (row, col, value)

def filter(fn: (R, C, V) => Boolean): SparseMatrix[R, C, V] = {

SparseMatrix(this.pipe.filter {

case (row, col, value) => fn(row, col, value)

})

}

// filter the matrix based on a subset of rows

def filterRows(rows: TypedPipe[R]): SparseMatrix[R, C, V] = {

SparseMatrix(this.rowAsKeys.join(rows.asKeys).map {

case (row, ((col, value), \_)) => (row, col, value)

})

}

// filter the matrix based on a subset of cols

def filterCols(cols: TypedPipe[C]): SparseMatrix[R, C, V] = {

this.transpose.filterRows(cols).transpose

}

// convert the triplet (row, col, value) to a new (row1, col1, value1)

def tripleApply[R1, C1, V1](

fn: (R, C, V) => (R1, C1, V1)

)(

implicit rowOrd1: Ordering[R1],

colOrd1: Ordering[C1],

numericV1: Numeric[V1],

semigroupV1: Semigroup[V1],

rowInj: Injection[R1, Array[Byte]],

colInj: Injection[C1, Array[Byte]]

): SparseMatrix[R1, C1, V1] = {

SparseMatrix(this.pipe.map {

case (row, col, value) => fn(row, col, value)

})

}

// get the l1 norms for all rows

lazy val rowL1Norms: TypedPipe[(R, Double)] = {

this.pipe.map {

case (row, \_, value) =>

row -> numericV.toDouble(value).abs

}.sumByKey

}

// get the l2 norms for all rows

lazy val rowL2Norms: TypedPipe[(R, Double)] = {

this.pipe

.map {

case (row, \_, value) =>

row -> numericV.toDouble(value) \* numericV.toDouble(value)

}

.sumByKey

.mapValues(math.sqrt)

}

// normalize the matrix to make sure each row has unit norm

lazy val rowL2Normalize: SparseMatrix[R, C, Double] = {

val result = this.rowAsKeys

.join(this.rowL2Norms)

.collect {

case (row, ((col, value), l2norm)) if l2norm > 0.0 =>

(row, col, numericV.toDouble(value) / l2norm)

}

SparseMatrix(result)

}

// get the l2 norms for all cols

lazy val colL2Norms: TypedPipe[(C, Double)] = {

this.transpose.rowL2Norms

}

// normalize the matrix to make sure each column has unit norm

lazy val colL2Normalize: SparseMatrix[R, C, Double] = {

this.transpose.rowL2Normalize.transpose

}

/\*\*

\* Take topK non-zero elements from each row. Cols are ordered by the `ordering` function

\*/

def sortWithTakePerRow(k: Int)(ordering: Ordering[(C, V)]): TypedPipe[(R, Seq[(C, V)])] = {

this.rowAsKeys.group.sortedTake(k)(ordering)

}

/\*\*

\* Take topK non-zero elements from each column. Rows are ordered by the `ordering` function.

\*

\*/

def sortWithTakePerCol(k: Int)(ordering: Ordering[(R, V)]): TypedPipe[(C, Seq[(R, V)])] = {

this.transpose.sortWithTakePerRow(k)(ordering)

}

/\*\*

\* Multiply another SparseMatrix. The only requirement is that the col type of current matrix should

\* be same with the row type of the other matrix.

\*

\* @param sparseMatrix another matrix to multiply

\* @param numReducersOpt optional parameter to set number of reducers. It uses 1000 by default.

\* you can change it based on your applications.

\* @param ordering2 ordering function for the column type of another matrix

\* @param injection2 injection function for the column type of another matrix

\* @tparam C2 col type of another matrix

\*

\* @return

\*/

def multiplySparseMatrix[C2](

sparseMatrix: SparseMatrix[C, C2, V],

numReducersOpt: Option[Int] = None

)(

implicit ordering2: Ordering[C2],

injection2: Injection[C2, Array[Byte]]

): SparseMatrix[R, C2, V] = {

implicit val colInjectionFunction: C => Array[Byte] = colInj.toFunction

val result =

// 1000 is the reducer number used for sketchJoin; 1000 is a number that works well empirically.

// feel free to change this or make this as a param if you find this does not work for your case.

this.transpose.rowAsKeys

.sketch(numReducersOpt.getOrElse(1000))

.join(sparseMatrix.rowAsKeys)

.map {

case (\_, ((row1, value1), (col2, value2))) =>

(row1, col2) -> numericV.times(value1, value2)

}

.sumByKey

.map {

case ((row, col), value) =>

(row, col, value)

}

SparseMatrix(result)

}

/\*\*

\* Multiply a SparseRowMatrix. The implementation of this function assume the input SparseRowMatrix

\* is a skinny matrix, i.e., with a small number of unique columns. Based on our experience, you can

\* think 100K is a small number here.

\*

\* @param skinnyMatrix another matrix to multiply

\* @param numReducersOpt optional parameter to set number of reducers. It uses 1000 by default.

\* you can change it based on your applications.

\* @param ordering2 ordering function for the column type of another matrix

\* @param injection2 injection function for the column type of another matrix

\* @tparam C2 col type of another matrix

\*

\* @return

\*/

def multiplySkinnySparseRowMatrix[C2](

skinnyMatrix: SparseRowMatrix[C, C2, V],

numReducersOpt: Option[Int] = None

)(

implicit ordering2: Ordering[C2],

injection2: Injection[C2, Array[Byte]]

): SparseRowMatrix[R, C2, V] = {

assert(

skinnyMatrix.isSkinnyMatrix,

"this function only works for skinny sparse row matrix, otherwise you will get out-of-memory problem")

implicit val colInjectionFunction: C => Array[Byte] = colInj.toFunction

val result =

// 1000 is the reducer number used for sketchJoin; 1000 is a number that works well empirically.

// feel free to change this or make this as a param if you find this does not work for your case.

this.transpose.rowAsKeys

.sketch(numReducersOpt.getOrElse(1000))

.join(skinnyMatrix.pipe)

.map {

case (\_, ((row1, value1), colMap)) =>

row1 -> colMap.mapValues(v => numericV.times(value1, v))

}

.sumByKey

SparseRowMatrix(result, skinnyMatrix.isSkinnyMatrix)

}

/\*\*\*

\* Multiply a DenseRowMatrix. The result will be also a DenseRowMatrix.

\*

\* @param denseRowMatrix matrix to multiply

\* @param numReducersOpt optional parameter to set number of reducers. It uses 1000 by default.

\* you can change it based on your applications

\* @return

\*/

def multiplyDenseRowMatrix(

denseRowMatrix: DenseRowMatrix[C],

numReducersOpt: Option[Int] = None

): DenseRowMatrix[R] = {

implicit val colInjectionFunction: C => Array[Byte] = colInj.toFunction

implicit val arrayVSemiGroup: Semigroup[Array[Double]] = denseRowMatrix.semigroupArrayV

val result =

// 1000 is the reducer number used for sketchJoin; 1000 is a number that works well empirically.

// feel free to change this or make this as a param if you find this does not work for your case.

this.transpose.rowAsKeys

.sketch(numReducersOpt.getOrElse(1000))

.join(denseRowMatrix.pipe)

.map {

case (\_, ((row1, value1), array)) =>

row1 -> array.map(v => numericV.toDouble(value1) \* v)

}

.sumByKey

DenseRowMatrix(result)

}

// Transpose the matrix.

lazy val transpose: SparseMatrix[C, R, V] = {

SparseMatrix(

this.pipe

.map {

case (row, col, value) =>

(col, row, value)

})

}

// Create a Key-Val TypedPipe for .join() and other use cases.

lazy val rowAsKeys: TypedPipe[(R, (C, V))] = {

this.pipe

.map {

case (row, col, value) =>

(row, (col, value))

}

}

// convert to a TypedPipe

lazy val toTypedPipe: TypedPipe[(R, C, V)] = {

this.pipe

}

lazy val forceToDisk: SparseMatrix[R, C, V] = {

SparseMatrix(this.pipe.forceToDisk)

}

/\*\*

\* Convert the matrix to a SparseRowMatrix. Do this only when the max number of non-zero values per row is

\* small (say, not more than 200K).

\*

\* @isSkinnyMatrix is the resulted matrix skinny, i.e., number of unique colIds is small (<200K).

\* Note the difference between `number of unique colIds` and `max number of non-zero values per row`.

\* @return

\*/

def toSparseRowMatrix(isSkinnyMatrix: Boolean = false): SparseRowMatrix[R, C, V] = {

SparseRowMatrix(

this.pipe.map {

case (i, j, v) =>

i -> Map(j -> v)

}.sumByKey,

isSkinnyMatrix)

}

/\*\*

\* Convert the matrix to a DenseRowMatrix

\*

\* @param numCols the number of columns in the DenseRowMatrix.

\* @param colToIndexFunction the function to convert colId to the column index in the dense matrix

\* @return

\*/

def toDenseRowMatrix(numCols: Int, colToIndexFunction: C => Int): DenseRowMatrix[R] = {

this.toSparseRowMatrix(isSkinnyMatrix = true).toDenseRowMatrix(numCols, colToIndexFunction)

}

/\*\*

\* Determines whether we should return a given Iterator given a threshold for the sum of values

\* across a row and whether we are looking to stay under or above that value.

\* Note that Iterators are mutable/destructive, and even calling .size on it will 'use it up'

\* i.e. it no longer hasNext and we no longer have any reference to the head of the collection.

\*

\* @param columnValueIterator Iterator over column-value pairs.

\* @param threshold The threshold for the sum of values

\* @param ifMin True if we want to stay at least above that given value

\* @return A new SparseMatrix after we have filtered the ineligible rows

\*/

private[this] def filterIter(

columnValueIterator: Iterator[(C, V)],

threshold: V,

ifMin: Boolean

): Iterator[(C, V)] = {

var sum: V = numericV.zero

var it: Iterator[(C, V)] = Iterator.empty

var exceeded = false

while (columnValueIterator.hasNext && !exceeded) {

val (c, v) = columnValueIterator.next

val nextSum = semigroupV.plus(sum, v)

val cmp = numericV.compare(nextSum, threshold)

if ((ifMin && cmp < 0) || (!ifMin && cmp <= 0)) {

it = it ++ Iterator((c, v))

sum = nextSum

} else {

it = it ++ Iterator((c, v))

exceeded = true

}

}

(ifMin, exceeded) match {

case (true, true) => it ++ columnValueIterator

case (true, false) => Iterator.empty

case (false, true) => Iterator.empty

case (false, false) => it ++ columnValueIterator

}

}

/\*\*

\* removes entries whose sum over rows do not meet the minimum sum (minSum)

\* @param minSum minimum sum for which we want to enforce across all rows

\*/

def filterRowsByMinSum(minSum: V): SparseMatrix[R, C, V] = {

val filteredPipe = this.rowAsKeys.group

.mapValueStream(filterIter(\_, threshold = minSum, ifMin = true)).map {

case (r, (c, v)) =>

(r, c, v)

}

SparseMatrix(filteredPipe)

}

/\*\*

\* removes entries whose sum over rows exceed the maximum sum (maxSum)

\* @param maxSum maximum sum for which we want to enforce across all rows

\*/

def filterRowsByMaxSum(maxSum: V): SparseMatrix[R, C, V] = {

val filteredPipe = this.rowAsKeys.group

.mapValueStream(filterIter(\_, threshold = maxSum, ifMin = false)).map {

case (r, (c, v)) =>

(r, c, v)

}

SparseMatrix(filteredPipe)

}

}