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

import com.twitter.algebird.Semigroup

import com.twitter.bijection.Injection

import com.twitter.scalding.TypedPipe

import com.twitter.scalding.ValuePipe

import org.apache.avro.SchemaBuilder.ArrayBuilder

import scala.util.Random

/\*\*

\* A class that represents a row-indexed matrix, backed by a TypedPipe[(R, Map(C, V)].

\* For each row of the TypedPipe, we save the rowId and a map consisting of colIds and their values.

\* Only use this class when the max number of non-zero values per row is small (say, <100K).

\*

\* Compared to SparseMatrix, this class has some optimizations to efficiently perform some row-wise

\* operations.

\*

\* Also, if the matrix is skinny (i.e., number of unique colIds is small), we have optimized solutions

\* for col-wise normalization as well as matrix multiplication (see SparseMatrix.multiplySkinnySparseRowMatrix).

\*

\* @param pipe underlying pipe

\* @param isSkinnyMatrix if the matrix is skinny (i.e., number of unique colIds is small)

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

\* @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 SparseRowMatrix[R, C, V](

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

isSkinnyMatrix: Boolean

)(

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

.values

.map(\_.size.toLong)

.sum

}

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

this.pipe

.collect {

case (i, values) if i == rowId =>

values.collect {

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

}

}

.flatten

.sum

}

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

this.pipe.flatMap {

case (i, values) if i == rowId =>

values.toSeq

case \_ =>

Nil

}

}

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

this.pipe.flatMap {

case (i, values) =>

values.collect {

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

i -> value

}

}

}

override lazy val uniqueRowIds: TypedPipe[R] = {

this.pipe.map(\_.\_1).distinct

}

override lazy val uniqueColIds: TypedPipe[C] = {

this.pipe.flatMapValues(\_.keys).values.distinct

}

// convert to a SparseMatrix

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

SparseMatrix(this.pipe.flatMap {

case (i, values) =>

values.map { case (j, value) => (i, j, value) }

})

}

// convert to a TypedPipe

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

this.pipe

}

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

SparseRowMatrix(

this.pipe

.map {

case (i, values) =>

i -> values.filter { case (j, v) => fn(i, j, v) }

}

.filter(\_.\_2.nonEmpty),

isSkinnyMatrix = this.isSkinnyMatrix

)

}

// sample the rows in the matrix as defined by samplingRatio

def sampleRows(samplingRatio: Double): SparseRowMatrix[R, C, V] = {

SparseRowMatrix(this.pipe.filter(\_ => Random.nextDouble < samplingRatio), this.isSkinnyMatrix)

}

// filter the matrix based on a subset of rows

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

SparseRowMatrix(this.pipe.join(rows.asKeys).mapValues(\_.\_1), this.isSkinnyMatrix)

}

// filter the matrix based on a subset of cols

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

this.toSparseMatrix.filterCols(cols).toSparseRowMatrix(this.isSkinnyMatrix)

}

// 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]]

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

SparseRowMatrix(

this.pipe.flatMap {

case (i, values) =>

values

.map {

case (j, v) => fn(i, j, v)

}

.groupBy(\_.\_1)

.mapValues { \_.map { case (\_, j1, v1) => (j1, v1) }.toMap }

},

isSkinnyMatrix = this.isSkinnyMatrix

)

}

// get the l2 norms for all rows. this does not trigger a shuffle.

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

this.pipe.map {

case (row, values) =>

row -> math.sqrt(

values.values

.map(a => numericV.toDouble(a) \* numericV.toDouble(a))

.sum)

}

}

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

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

val result = this.pipe.flatMap {

case (row, values) =>

val norm =

math.sqrt(

values.values

.map(v => numericV.toDouble(v) \* numericV.toDouble(v))

.sum)

if (norm == 0.0) {

None

} else {

Some(row -> values.mapValues(v => numericV.toDouble(v) / norm))

}

}

SparseRowMatrix(result, isSkinnyMatrix = this.isSkinnyMatrix)

}

// get the l2 norms for all cols

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

this.pipe

.flatMap {

case (\_, values) =>

values.map {

case (col, v) =>

col -> numericV.toDouble(v) \* numericV.toDouble(v)

}

}

.sumByKey

.mapValues(math.sqrt)

}

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

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

val result = if (this.isSkinnyMatrix) {

// if this is a skinny matrix, we first put the norm of all columns into a Map, and then use

// this Map inside the mappers without shuffling the whole matrix (which is expensive, see the

// `else` part of this function).

val colL2NormsValuePipe = this.colL2Norms.map {

case (col, norm) => Map(col -> norm)

}.sum

this.pipe.flatMapWithValue(colL2NormsValuePipe) {

case ((row, values), Some(colNorms)) =>

Some(row -> values.flatMap {

case (col, value) =>

val colNorm = colNorms.getOrElse(col, 0.0)

if (colNorm == 0.0) {

None

} else {

Some(col -> numericV.toDouble(value) / colNorm)

}

})

case \_ =>

None

}

} else {

this.toSparseMatrix.transpose.rowAsKeys

.join(this.colL2Norms)

.collect {

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

row -> Map(col -> numericV.toDouble(value) / colNorm)

}

.sumByKey

.toTypedPipe

}

SparseRowMatrix(result, isSkinnyMatrix = this.isSkinnyMatrix)

}

/\*\*

\* 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.pipe.map {

case (row, values) =>

row -> values.toSeq.sorted(ordering).take(k)

}

}

/\*\*

\* 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.toSparseMatrix.sortWithTakePerCol(k)(ordering)

}

/\*\*

\* Similar to .forceToDisk function in TypedPipe, but with an option to specify how many partitions

\* to save, which is useful if you want to consolidate the data set or want to tune the number

\* of mappers for the next step.

\*

\* @param numShardsOpt number of shards to save the data.

\*

\* @return

\*/

def forceToDisk(

numShardsOpt: Option[Int] = None

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

numShardsOpt

.map { numShards =>

SparseRowMatrix(this.pipe.shard(numShards), this.isSkinnyMatrix)

}

.getOrElse {

SparseRowMatrix(this.pipe.forceToDisk, this.isSkinnyMatrix)

}

}

/\*\*

\* transpose current matrix and multiple another Skinny SparseRowMatrix.

\* The difference between this and .transpose.multiplySkinnySparseRowMatrix(anotherSparseRowMatrix),

\* is that we do not need to do flatten and group again.

\*

\* One use case is to when we need to compute the column-wise covariance matrix, then we only need

\* a.transposeAndMultiplySkinnySparseRowMatrix(a) to get it.

\*

\* @param anotherSparseRowMatrix it needs to be a skinny SparseRowMatrix

\* @numReducersOpt Number of reducers.

\*/

def transposeAndMultiplySkinnySparseRowMatrix[C2](

anotherSparseRowMatrix: SparseRowMatrix[R, C2, V],

numReducersOpt: Option[Int] = None

)(

implicit ordering2: Ordering[C2],

injection2: Injection[C2, Array[Byte]]

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

// it needs to be a skinny SparseRowMatrix, otherwise we will have out-of-memory issue

require(anotherSparseRowMatrix.isSkinnyMatrix)

SparseRowMatrix(

numReducersOpt

.map { numReducers =>

this.pipe

.join(anotherSparseRowMatrix.pipe).withReducers(numReducers)

}.getOrElse(this.pipe

.join(anotherSparseRowMatrix.pipe))

.flatMap {

case (\_, (row1, row2)) =>

row1.map {

case (col1, val1) =>

col1 -> row2.mapValues(val2 => numericV.times(val1, val2))

}

}

.sumByKey,

isSkinnyMatrix = true

)

}

/\*\*\*

\* 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] = {

this.toSparseMatrix.multiplyDenseRowMatrix(denseRowMatrix, numReducersOpt)

}

/\*\*

\* 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] = {

DenseRowMatrix(this.pipe.map {

case (row, colMap) =>

val array = new Array[Double](numCols)

colMap.foreach {

case (col, value) =>

val index = colToIndexFunction(col)

assert(index < numCols && index >= 0, "The converted index is out of range!")

array(index) = numericV.toDouble(value)

}

row -> array

})

}

}