package com.twitter.simclusters\_v2.scalding

import com.twitter.algebird.Monoid

import com.twitter.logging.Logger

import com.twitter.scalding.{Execution, TypedPipe, TypedTsv}

import com.twitter.scalding\_internal.job.TwitterExecutionApp

import com.twitter.simclusters\_v2.hdfs\_sources.AdhocKeyValSources

import java.util

import no.uib.cipr.matrix.Matrix

import no.uib.cipr.matrix.sparse.{ArpackSym, LinkedSparseMatrix}

import scala.collection.JavaConverters.\_

object EigenVectorsForSparseSymmetric {

val log: Logger = Logger()

/\*\*

\* Construct matrix from the rows of the matrix, specified as a map. The outer map is indexed by rowId, and the inner maps are indexed by columnId.

\* Note that the input matrix is intended to be symmetric.

\*

\* @param map A map specifying the rows of the matrix. The outer map is indexed by rowId, and the inner maps are indexed by columnId. Both rows and columns are zero-indexed.

\* @param nRows number of rows in matrix

\* @param nCols number of columns in matrix

\*

\* @return the constructed matrix

\*/

def getMatrix(map: Map[Int, Map[Int, Double]], nRows: Int, nCols: Int): Matrix = {

val nonzeros = map.toSeq.flatMap {

case (i, subMap) =>

subMap.toSeq.map {

case (j, value) =>

(i, j, value)

}

}

getMatrix(nonzeros, nRows, nCols)

}

/\*\*

\* Construct matrix from iterable of the non-zero entries. Note that the input matrix is intended to be symmetric.

\*

\* @param nonzeros non-zeros in (i, j, v) format, where i is row, j is column, and v is value. Both rows and columns are zero-indexed.

\* @param nRows number of rows in matrix

\* @param nCols number of columns in matrix

\*

\* @return the constructed matrix

\*/

def getMatrix(nonzeros: Iterable[(Int, Int, Double)], nRows: Int, nCols: Int): Matrix = {

val matrix = new LinkedSparseMatrix(nRows, nCols)

var numEntries = 0

var maxRow = 0

var maxCol = 0

nonzeros.foreach {

case (i, j, v) =>

if (i > maxRow) {

maxRow = i

}

if (j > maxCol) {

maxCol = j

}

numEntries += 1

matrix.set(i, j, v)

}

log.info(

"Finished building matrix with %d entries and maxRow %d and maxCol %d"

.format(numEntries, maxRow, maxCol))

matrix

}

/\*\*

\* Prints out various diagnostics about how much the given matrix differs from a perfect

\* symmetric matrix. If (i,j) and (j,i) are different, it sets both of them to be the max of the two.

\* Call this function before invoking EVD.

\*

\* @param matrix Matrix which is modified (if need be) in place.

\*/

def ensureMatrixIsSymmetric(matrix: Matrix): Unit = {

var numUnequalEntries = 0

var numEntriesDifferentBy1Percent = 0

var numEqualEntries = 0

var numUnequalDueToZero = 0

var maxUnequal = (0, 0, 0.0, 0.0)

matrix.iterator().asScala.foreach { entry =>

val curr = entry.get()

val opp = matrix.get(entry.column(), entry.row())

if (curr == opp) {

numEqualEntries += 1

} else {

numUnequalEntries += 1

if (opp == 0) {

numUnequalDueToZero += 1

}

if (opp != 0 && (math.abs(curr - opp) / math.min(curr, opp)) > 0.01) {

numEntriesDifferentBy1Percent += 1

}

if (opp != 0 && math.abs(curr - opp) > maxUnequal.\_4) {

maxUnequal = (entry.row(), entry.column(), curr, math.abs(curr - opp))

}

val max = math.max(curr, opp)

matrix.set(entry.column(), entry.row(), max)

matrix.set(entry.row(), entry.column(), max)

}

}

var numUnEqualPrinted = 0

matrix.iterator().asScala.foreach { entry =>

val opp = matrix.get(entry.column(), entry.row())

if (numUnEqualPrinted < 10 && entry.get() != opp) {

numUnEqualPrinted += 1

log.info(

"Entries for (%d, %d) are %s and %s"

.format(entry.row(), entry.column(), entry.get(), opp))

}

}

log.info(

"Num unequal entries: %d, num unequal due to zero: %d, num unequal by 1percent or more: %d, num equal entries: %d, maxUnequal: %s"

.format(

numUnequalEntries,

numUnequalDueToZero,

numEntriesDifferentBy1Percent,

numEqualEntries,

maxUnequal))

}

/\*\*

\* Get the top-k eigenvalues (largest magnitude) and eigenvectors for an input matrix.

\* Top eigenvalues means they're the largest in magnitude.

\* Input matrix needs to be perfectly symmetric; if it's not, this function will fail.

\*

\* Many of the eigenvectors will have very small values along most of the dimensions. This method also

\* only retains the bigger entries in an eigenvector.

\*

\* @param matrix symmetric input matrix.

\* @param k how many of the top eigenvectors to get.

\* @param ratioToLargestCutoff An entry needs to be at least 1/ratioToLargestCutoff of the biggest entry in that vector to be retained.

\*

\* @return seq of (eigenvalue, eigenvector) pairs.

\*/

def getTruncatedEVD(

matrix: Matrix,

k: Int,

ratioToLargestCutoff: Float

): Seq[(Double, Seq[(Int, Double)])] = {

val solver = new ArpackSym(matrix)

val resultsMap = solver.solve(k, ArpackSym.Ritz.LM).asScala.toMap

val results = resultsMap.toIndexedSeq.sortBy { case (eigValue, \_) => -eigValue }

results.zipWithIndex.map {

case ((eigValue, denseVectorJava), index) =>

val denseVector = new Array[Double](denseVectorJava.size())

denseVector.indices.foreach { index => denseVector(index) = denseVectorJava.get(index) }

val denseVectorMax = denseVector.maxBy { entry => math.abs(entry) }

val cutOff = math.abs(denseVectorMax) / ratioToLargestCutoff

val significantEntries = denseVector.zipWithIndex

.filter { case (vectorEntry, \_) => math.abs(vectorEntry) >= cutOff }

.sortBy { case (vectorEntry, \_) => -1 \* math.abs(vectorEntry) }

(eigValue.toDouble, significantEntries.toSeq.map(\_.swap))

}

}

/\*\*

\* Compute U\*Diag\*Ut - where Diag is a diagonal matrix, and U is a sparse matrix.

\* This is primarily for testing - to make sure that the computed eigenvectors can be used to

\* reconstruct the original matrix up to some reasonable approximation.

\*

\* @param diagToUColumns seq of (diagonal entries, associated column in U)

\* @param cutoff cutoff for including a value in the result.

\*

\* @return result of multiplication, returned as a map of the rows in the results.

\*/

def uTimesDiagTimesUT(

diagToUColumns: Seq[(Double, Seq[(Int, Double)])],

cutoff: Double

): Map[Int, Map[Int, Double]] = {

val result = new util.HashMap[Int, util.HashMap[Int, Double]]()

diagToUColumns.foreach {

case (diag, uColumn) =>

uColumn.foreach {

case (i, iVal) =>

uColumn.foreach {

case (j, jVal) =>

val prod = diag \* iVal \* jVal

if (result.containsKey(i)) {

val newVal = if (result.get(i).containsKey(j)) {

result.get(i).get(j) + prod

} else prod

result.get(i).put(j, newVal)

} else {

result.put(i, new util.HashMap[Int, Double])

result.get(i).put(j, prod)

}

}

}

}

val unfiltered = result.asScala.toMap.mapValues(\_.asScala.toMap)

unfiltered

.mapValues { m => m.filter { case (\_, value) => math.abs(value) >= cutoff } }

.filter { case (\_, vector) => vector.nonEmpty }

}

/\*\* Note: This requires a full EVD to correctly compute the inverse! :-( \*/

def getInverseFromEVD(

evd: Seq[(Double, Seq[(Int, Double)])],

cutoff: Double

): Map[Int, Map[Int, Double]] = {

val evdInverse = evd.map {

case (eigValue, eigVector) =>

(1.0 / eigValue, eigVector)

}

uTimesDiagTimesUT(evdInverse, cutoff)

}

}

object PCAProjectionMatrixAdhoc extends TwitterExecutionApp {

val log = Logger()

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, \_) =>

Execution.withId { \_ =>

val args = config.getArgs

val k = args.int("k", 100)

val ratioToLargestEntryInVectorCutoff = args.int("ratioToLargestEntryInVectorCutoff", 100)

val minClusterFavers = args.int("minClusterFavers", 1000)

val input = TypedPipe.from(AdhocKeyValSources.clusterDetailsSource(args("inputDir")))

val outputDir = args("outputDir")

val filteredClustersExec =

input

.collect {

case ((\_, clusterId), details)

if details.numUsersWithNonZeroFavScore > minClusterFavers =>

clusterId

}

.toIterableExecution

.map { fc =>

val fcSet = fc.toSet

log.info("Number of clusters with favers more than %d is %d"

.format(minClusterFavers, fcSet.size))

fcSet

}

filteredClustersExec

.flatMap { filteredClusters =>

input.flatMap {

case ((\_, clusterId), details) =>

if (filteredClusters(clusterId)) {

details.neighborClusters.getOrElse(Nil).collect {

case neighbor

if filteredClusters(

neighbor.clusterId) && neighbor.favCosineSimilarity.isDefined =>

(clusterId, neighbor.clusterId, neighbor.favCosineSimilarity.get)

}

} else Nil

}.toIterableExecution

}

.flatMap { edgesIter =>

val edges = edgesIter.toSeq

val oldIdToNewId = edges

.flatMap { case (i, j, \_) => Seq(i, j) }

.distinct

.zipWithIndex

.toMap

val mapString = oldIdToNewId.toList

.take(5).map {

case (old, nw) =>

Seq(old, nw).mkString(" ")

}.mkString("\n")

log.info("A few entries of OldId to NewId map is")

log.info(mapString)

val newIdToOldId = oldIdToNewId.map(\_.swap)

log.info(

"Num clusters after filtering out those with no neighbors with favers more than %d is %d"

.format(minClusterFavers, oldIdToNewId.size))

val newEdges = edges.map {

case (oldI, oldJ, value) =>

(oldIdToNewId(oldI), oldIdToNewId(oldJ), value)

}

log.info("Going to build matrix")

val matrix = EigenVectorsForSparseSymmetric.getMatrix(

newEdges,

oldIdToNewId.size,

oldIdToNewId.size)

EigenVectorsForSparseSymmetric.ensureMatrixIsSymmetric(matrix)

log.info("Going to solve now for %d eigenvalues".format(k))

val tic = System.currentTimeMillis()

val results = EigenVectorsForSparseSymmetric.getTruncatedEVD(

matrix,

k,

ratioToLargestEntryInVectorCutoff)

val toc = System.currentTimeMillis()

log.info("Finished solving in %.2f minutes".format((toc - tic) / 1000 / 60.0))

val eigValues = results.map(\_.\_1).map { x => "%.3g".format(x) }.mkString(" ")

val eigValueNorm = math.sqrt(results.map(\_.\_1).map(x => x \* x).sum)

val matrixNorm = math.sqrt(matrix.iterator().asScala.map(\_.get()).map(x => x \* x).sum)

println(

"matrixNorm %s, eigValueNorm %s, explained fraction %s"

.format(matrixNorm, eigValueNorm, eigValueNorm / matrixNorm))

log.info("The eigenvalues are:")

log.info(eigValues)

val nnzInEigenVectors = results.map(\_.\_2.size).sum

log.info("Average nnz per eigenvector using ratioToLargestCutoff %d is %.2g"

.format(ratioToLargestEntryInVectorCutoff, nnzInEigenVectors \* 1.0 / results.size))

val transposedRaw = results.zipWithIndex.flatMap {

case ((\_, eigVector), eigIndex) =>

eigVector.map {

case (index, vectorEntry) =>

val clusterId = newIdToOldId(index)

Map(clusterId -> List((eigIndex, vectorEntry)))

}

}

val transposed = Monoid.sum(transposedRaw).mapValues { rowForCluster =>

rowForCluster

.map {

case (dimId, weight) =>

"%d:%.2g".format(dimId, weight)

}.mkString(" ")

}

TypedPipe.from(transposed.toSeq).writeExecution(TypedTsv(outputDir))

}

}

}

}