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

import com.twitter.scalding.{Args, DateRange, Execution, TypedPipe, UniqueID}

import com.twitter.simclusters\_v2.common.ModelVersions

import com.twitter.simclusters\_v2.scalding.common.matrix.{SparseMatrix, SparseRowMatrix}

import com.twitter.simclusters\_v2.scalding.embedding.common.EmbeddingUtil.\_

import com.twitter.simclusters\_v2.thriftscala.\_

import java.util.TimeZone

/\*\*

\* This is the base job for computing SimClusters Embedding for any Noun Type on Twitter, such as

\* Users, Tweets, Topics, Entities, Channels, etc.

\*

\* The most straightforward way to understand the SimClusters Embeddings for a Noun is that it is

\* a weighted sum of SimClusters InterestedIn vectors from users who are interested in the Noun.

\* So for a noun type, you only need to define `prepareNounToUserMatrix` to pass in a matrix which

\* represents how much each user is interested in this noun.

\*/

trait SimClustersEmbeddingBaseJob[NounType] {

def numClustersPerNoun: Int

def numNounsPerClusters: Int

def thresholdForEmbeddingScores: Double

def numReducersOpt: Option[Int] = None

def prepareNounToUserMatrix(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): SparseMatrix[NounType, UserId, Double]

def prepareUserToClusterMatrix(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): SparseRowMatrix[UserId, ClusterId, Double]

def writeNounToClustersIndex(

output: TypedPipe[(NounType, Seq[(ClusterId, Double)])]

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit]

def writeClusterToNounsIndex(

output: TypedPipe[(ClusterId, Seq[(NounType, Double)])]

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit]

def runOnDateRange(

args: Args

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

val embeddingMatrix: SparseRowMatrix[NounType, ClusterId, Double] =

prepareNounToUserMatrix.rowL2Normalize

.multiplySkinnySparseRowMatrix(

prepareUserToClusterMatrix.colL2Normalize,

numReducersOpt

)

.filter((\_, \_, v) => v > thresholdForEmbeddingScores)

Execution

.zip(

writeNounToClustersIndex(

embeddingMatrix.sortWithTakePerRow(numClustersPerNoun)(Ordering.by(-\_.\_2))

),

writeClusterToNounsIndex(

embeddingMatrix.sortWithTakePerCol(numNounsPerClusters)(

Ordering.by(-\_.\_2)

)

)

)

.unit

}

}

object SimClustersEmbeddingJob {

/\*\*

\* Multiply the [user, cluster] and [user, T] matrices, and return the cross product.

\*/

def computeEmbeddings[T](

simClustersSource: TypedPipe[(UserId, ClustersUserIsInterestedIn)],

normalizedInputMatrix: TypedPipe[(UserId, (T, Double))],

scoreExtractors: Seq[UserToInterestedInClusterScores => (Double, ScoreType.ScoreType)],

modelVersion: ModelVersion,

toSimClustersEmbeddingId: (T, ScoreType.ScoreType) => SimClustersEmbeddingId,

numReducers: Option[Int] = None

): TypedPipe[(SimClustersEmbeddingId, (ClusterId, Double))] = {

val userSimClustersMatrix =

getUserSimClustersMatrix(simClustersSource, scoreExtractors, modelVersion)

multiplyMatrices(

normalizedInputMatrix,

userSimClustersMatrix,

toSimClustersEmbeddingId,

numReducers)

}

def getL2Norm[T](

inputMatrix: TypedPipe[(T, (UserId, Double))],

numReducers: Option[Int] = None

)(

implicit ordering: Ordering[T]

): TypedPipe[(T, Double)] = {

val l2Norm = inputMatrix

.mapValues {

case (\_, score) => score \* score

}

.sumByKey

.mapValues(math.sqrt)

numReducers match {

case Some(reducers) => l2Norm.withReducers(reducers)

case \_ => l2Norm

}

}

def getNormalizedTransposeInputMatrix[T](

inputMatrix: TypedPipe[(T, (UserId, Double))],

numReducers: Option[Int] = None

)(

implicit ordering: Ordering[T]

): TypedPipe[(UserId, (T, Double))] = {

val inputWithNorm = inputMatrix.join(getL2Norm(inputMatrix, numReducers))

(numReducers match {

case Some(reducers) => inputWithNorm.withReducers(reducers)

case \_ => inputWithNorm

}).map {

case (inputId, ((userId, favScore), norm)) =>

(userId, (inputId, favScore / norm))

}

}

/\*\*

\* Matrix multiplication with the ability to tune the reducer size for better performance

\*/

@Deprecated

def legacyMultiplyMatrices[T](

normalizedTransposeInputMatrix: TypedPipe[(UserId, (T, Double))],

userSimClustersMatrix: TypedPipe[(UserId, Seq[(ClusterId, Double)])],

numReducers: Int // Matrix multiplication is expensive. Use this to tune performance

)(

implicit ordering: Ordering[T]

): TypedPipe[((ClusterId, T), Double)] = {

normalizedTransposeInputMatrix

.join(userSimClustersMatrix)

.withReducers(numReducers)

.flatMap {

case (\_, ((inputId, score), clustersWithScores)) =>

clustersWithScores.map {

case (clusterId, clusterScore) =>

((clusterId, inputId), score \* clusterScore)

}

}

.sumByKey

.withReducers(numReducers + 1) // +1 to distinguish this step from above in Dr. Scalding

}

def multiplyMatrices[T](

normalizedTransposeInputMatrix: TypedPipe[(UserId, (T, Double))],

userSimClustersMatrix: TypedPipe[(UserId, Seq[((ClusterId, ScoreType.ScoreType), Double)])],

toSimClustersEmbeddingId: (T, ScoreType.ScoreType) => SimClustersEmbeddingId,

numReducers: Option[Int] = None

): TypedPipe[(SimClustersEmbeddingId, (ClusterId, Double))] = {

val inputJoinedWithSimClusters = numReducers match {

case Some(reducers) =>

normalizedTransposeInputMatrix

.join(userSimClustersMatrix)

.withReducers(reducers)

case \_ =>

normalizedTransposeInputMatrix.join(userSimClustersMatrix)

}

val matrixMultiplicationResult = inputJoinedWithSimClusters.flatMap {

case (\_, ((inputId, inputScore), clustersWithScores)) =>

clustersWithScores.map {

case ((clusterId, scoreType), clusterScore) =>

((clusterId, toSimClustersEmbeddingId(inputId, scoreType)), inputScore \* clusterScore)

}

}.sumByKey

(numReducers match {

case Some(reducers) =>

matrixMultiplicationResult.withReducers(reducers + 1)

case \_ => matrixMultiplicationResult

}).map {

case ((clusterId, embeddingId), score) =>

(embeddingId, (clusterId, score))

}

}

def getUserSimClustersMatrix(

simClustersSource: TypedPipe[(UserId, ClustersUserIsInterestedIn)],

scoreExtractors: Seq[UserToInterestedInClusterScores => (Double, ScoreType.ScoreType)],

modelVersion: ModelVersion

): TypedPipe[(UserId, Seq[((ClusterId, ScoreType.ScoreType), Double)])] = {

simClustersSource.map {

case (userId, clusters)

if ModelVersions.toModelVersion(clusters.knownForModelVersion) == modelVersion =>

userId -> clusters.clusterIdToScores.flatMap {

case (clusterId, clusterScores) =>

scoreExtractors.map { scoreExtractor =>

scoreExtractor(clusterScores) match {

case (score, scoreType) => ((clusterId, scoreType), score)

}

}

}.toSeq

case (userId, \_) => userId -> Nil

}

}

def toReverseIndexSimClusterEmbedding(

embeddings: TypedPipe[(SimClustersEmbeddingId, (ClusterId, EmbeddingScore))],

topK: Int

): TypedPipe[(SimClustersEmbeddingId, InternalIdEmbedding)] = {

embeddings

.map {

case (embeddingId, (clusterId, score)) =>

(

SimClustersEmbeddingId(

embeddingId.embeddingType,

embeddingId.modelVersion,

InternalId.ClusterId(clusterId)),

(embeddingId.internalId, score))

}

.group

.sortedReverseTake(topK)(Ordering.by(\_.\_2))

.mapValues { topInternalIdsWithScore =>

val internalIdsWithScore = topInternalIdsWithScore.map {

case (internalId, score) => InternalIdWithScore(internalId, score)

}

InternalIdEmbedding(internalIdsWithScore)

}

}

}