package com.twitter.simclusters\_v2.scalding.update\_known\_for

import com.twitter.algebird.Max

import com.twitter.hermit.candidate.thriftscala.Candidates

import com.twitter.sbf.core.AlgorithmConfig

import com.twitter.sbf.core.MHAlgorithm

import com.twitter.sbf.core.SparseBinaryMatrix

import com.twitter.sbf.core.SparseRealMatrix

import com.twitter.sbf.graph.Graph

import com.twitter.scalding.Days

import com.twitter.scalding.Execution

import com.twitter.scalding.Hdfs

import com.twitter.scalding.Mode

import com.twitter.scalding.Stat

import com.twitter.scalding.TypedTsv

import com.twitter.scalding.UniqueID

import com.twitter.scalding.commons.source.VersionedKeyValSource

import com.twitter.scalding.typed.TypedPipe

import com.twitter.scalding\_internal.dalv2.DAL

import com.twitter.scalding\_internal.dalv2.remote\_access.ExplicitLocation

import com.twitter.scalding\_internal.dalv2.remote\_access.ProcAtla

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

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

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

import com.twitter.simclusters\_v2.scalding.CompareClusters

import com.twitter.simclusters\_v2.scalding.KnownForSources

import com.twitter.simclusters\_v2.scalding.TopUser

import com.twitter.simclusters\_v2.scalding.TopUserWithMappedId

import com.twitter.simclusters\_v2.scalding.TopUsersSimilarityGraph

import com.twitter.simclusters\_v2.scalding.common.Util

import com.twitter.usersource.snapshot.flat.UsersourceFlatScalaDataset

import java.io.PrintWriter

import java.util.TimeZone

import org.apache.commons.math3.random.JDKRandomGenerator

import org.apache.commons.math3.random.RandomAdaptor

import org.apache.hadoop.fs.FileSystem

import org.apache.hadoop.fs.Path

import scala.collection.mutable

object UpdateKnownForSBFRunner {

/\*\*

\* The main logic of the job. It works as follows:

\*

\* 1. read the top 20M users, and convert their UserIds to an integer Id from 0 to 20M in order to use the clustering library

\* 2. read the user similarity graph from Sims, and convert their UserIds to the same mapped integer Id

\* 3. read the previous known\_for data set for initialization of the clustering algorithm;

\* for users without previous assignments, we randomly assign them to some unused clusters (if there are any).

\* 4. run the clustering algorithm for x iterations (x = 4 in the prod setting)

\* 5. output of the clustering result as the new known\_for.

\*

\*/

def runUpdateKnownFor(

simsGraph: TypedPipe[Candidates],

minActiveFollowers: Int,

topK: Int,

maxNeighbors: Int,

tempLocationPath: String,

previousKnownFor: TypedPipe[(UserId, Array[(ClusterId, Float)])],

maxEpochsForClustering: Int,

squareWeightsEnable: Boolean,

wtCoeff: Double,

mode: Mode

)(

implicit

uniqueId: UniqueID,

tz: TimeZone

): Execution[TypedPipe[(UserId, Array[(ClusterId, Float)])]] = {

val tempLocationPathSimsGraph = tempLocationPath + "/sims\_graph"

val tempLocationPathMappedIds = tempLocationPath + "/mapped\_user\_ids"

val tempLocationPathClustering = tempLocationPath + "/clustering\_output"

val mappedIdsToUserIds: TypedPipe[(Int, UserId)] =

getTopFollowedUsersWithMappedIds(minActiveFollowers, topK)

.map {

case (id, mappedId) =>

(mappedId, id)

}

.shard(partitions = topK / 1e5.toInt)

val mappedSimsGraphInput: TypedPipe[(Int, List[(Int, Float)])] =

getMappedSimsGraph(

mappedIdsToUserIds,

simsGraph,

maxNeighbors

) // The simsGraph here consists of the mapped Ids and mapped ngbr Ids and not the original userIds

val mappedSimsGraphVersionedKeyVal: VersionedKeyValSource[Int, List[(Int, Float)]] =

AdhocKeyValSources.intermediateSBFResultsDevelSource(tempLocationPathSimsGraph)

val mappedIdsToUserIdsVersionedKeyVal: VersionedKeyValSource[Int, UserId] =

AdhocKeyValSources.mappedIndicesDevelSource(tempLocationPathMappedIds)

// exec to write intermediate results for mapped Sims Graph and mappedIds

val mappedSimsGraphAndMappedIdsWriteExec: Execution[Unit] = Execution

.zip(

mappedSimsGraphInput.writeExecution(mappedSimsGraphVersionedKeyVal),

mappedIdsToUserIds.writeExecution(mappedIdsToUserIdsVersionedKeyVal)

).unit

mappedSimsGraphAndMappedIdsWriteExec.flatMap { \_ =>

// The simsGraph and the mappedIds from userId(long) -> mappedIds are

// having to be written to a temporary location and read again before running

// the clustering algorithm.

Execution

.zip(

readIntermediateExec(

TypedPipe.from(mappedSimsGraphVersionedKeyVal),

mode,

tempLocationPathSimsGraph),

readIntermediateExec(

TypedPipe.from(mappedIdsToUserIdsVersionedKeyVal),

mode,

tempLocationPathMappedIds)

)

.flatMap {

case (mappedSimsGraphInputReadAgain, mappedIdsToUserIdsReadAgain) =>

val previousKnownForMappedIdsAssignments: TypedPipe[(Int, List[(ClusterId, Float)])] =

getKnownForWithMappedIds(

previousKnownFor,

mappedIdsToUserIdsReadAgain,

)

val clusteringResults = getClusteringAssignments(

mappedSimsGraphInputReadAgain,

previousKnownForMappedIdsAssignments,

maxEpochsForClustering,

squareWeightsEnable,

wtCoeff

)

clusteringResults

.flatMap { updatedKnownFor =>

// convert the list of updated KnownFor to a TypedPipe

convertKnownForListToTypedPipe(

updatedKnownFor,

mode,

tempLocationPathClustering

)

}

.flatMap { updatedKnownForTypedPipe =>

// convert the mapped integer id to raw user ids

val updatedKnownFor =

updatedKnownForTypedPipe

.join(mappedIdsToUserIdsReadAgain)

.values

.swap

.mapValues(\_.toArray)

Execution.from(updatedKnownFor)

}

}

}

}

/\*\*

\* Helper function to compare newKnownFor with the previous week knownFor assignments

\*/

def evaluateUpdatedKnownFor(

newKnownFor: TypedPipe[(UserId, Array[(ClusterId, Float)])],

inputKnownFor: TypedPipe[(UserId, Array[(ClusterId, Float)])]

)(

implicit uniqueId: UniqueID

): Execution[String] = {

val minSizeOfBiggerClusterForComparison = 10

val compareClusterExec = CompareClusters.summarize(

CompareClusters.compare(

KnownForSources.transpose(inputKnownFor),

KnownForSources.transpose(newKnownFor),

minSizeOfBiggerCluster = minSizeOfBiggerClusterForComparison

))

val compareProducerExec = CompareClusters.compareClusterAssignments(

newKnownFor.mapValues(\_.toList),

inputKnownFor.mapValues(\_.toList)

)

Execution

.zip(compareClusterExec, compareProducerExec)

.map {

case (compareClusterResults, compareProducerResult) =>

s"Cosine similarity distribution between cluster membership vectors for " +

s"clusters with at least $minSizeOfBiggerClusterForComparison members\n" +

Util.prettyJsonMapper

.writeValueAsString(compareClusterResults) +

"\n\n-------------------\n\n" +

"Custom counters:\n" + compareProducerResult +

"\n\n-------------------\n\n"

}

}

/\*\*

\*

\* Convert the list of updated KnownFor to a TypedPipe

\*

\* This step should have been done using TypedPipe.from(updatedKnownForList), however, due to the

\* large size of the list, TypedPipe would throw out-of-memory exceptions. So we have to first

\* dump it to a temp file on HDFS and using a customized read function to load to TypedPipe

\*

\*/

def convertKnownForListToTypedPipe(

updatedKnownForList: List[(Int, List[(ClusterId, Float)])],

mode: Mode,

temporaryOutputStringPath: String

): Execution[TypedPipe[(Int, List[(ClusterId, Float)])]] = {

val stringOutput = updatedKnownForList.map {

case (mappedUserId, clusterArray) =>

assert(clusterArray.isEmpty || clusterArray.length == 1)

val str = if (clusterArray.nonEmpty) {

clusterArray.head.\_1 + " " + clusterArray.head.\_2 // each user is known for at most 1 cluster

} else {

""

}

if (mappedUserId % 100000 == 0)

println(s"MappedIds:$mappedUserId ClusterAssigned$str")

s"$mappedUserId $str"

}

// using Execution to enforce the order of the following 3 steps:

// 1. write the list of strings to a temp file on HDFS

// 2. read the strings to TypedPipe

// 3. delete the temp file

Execution

.from(

// write the output to HDFS; the data will be loaded to Typedpipe later;

// the reason of doing this is that we can not just do TypePipe.from(stringOutput) which

// results in OOM.

TopUsersSimilarityGraph.writeToHDFSIfHDFS(

stringOutput.toIterator,

mode,

temporaryOutputStringPath

)

)

.flatMap { \_ =>

println(s"Start loading the data from $temporaryOutputStringPath")

val clustersWithScores = TypedPipe.from(TypedTsv[String](temporaryOutputStringPath)).map {

mappedIdsWithArrays =>

val strArray = mappedIdsWithArrays.trim().split("\\s+")

assert(strArray.length == 3 || strArray.length == 1)

val rowId = strArray(0).toInt

val clusterAssignment: List[(ClusterId, Float)] =

if (strArray.length > 1) {

List((strArray(1).toInt, strArray(2).toFloat))

} else {

// the knownFors will have users with Array.empty as their assignment if

// the clustering step have empty results for that user.

Nil

}

if (rowId % 100000 == 0)

println(s"rowId:$rowId ClusterAssigned: $clusterAssignment")

(rowId, clusterAssignment)

}

// return the dataset as an execution and delete the temp location

readIntermediateExec(clustersWithScores, mode, temporaryOutputStringPath)

}

}

/\*\*

\* Helper function to read the dataset as execution and delete the temporary

\* location on HDFS for PDP compliance

\*/

def readIntermediateExec[K, V](

dataset: TypedPipe[(K, V)],

mode: Mode,

tempLocationPath: String

): Execution[TypedPipe[(K, V)]] = {

Execution

.from(dataset)

.flatMap { output =>

// delete the temporary outputs for PDP compliance

mode match {

case Hdfs(\_, conf) =>

val fs = FileSystem.newInstance(conf)

if (fs.deleteOnExit(new Path(tempLocationPath))) {

println(s"Successfully deleted the temporary folder $tempLocationPath!")

} else {

println(s"Failed to delete the temporary folder $tempLocationPath!")

}

case \_ => ()

}

Execution.from(output)

}

}

/\*\*

\* Converts the userIDs in the sims graph to their mapped integer indices.

\* All the users who donot have a mapping are filtered out from the sims graph input

\*

\* @param mappedUsers mapping of long userIDs to their integer indices

\* @param allEdges sims graph

\* @param maxNeighborsPerNode number of neighbors for each user

\*

\* @return simsGraph of users and neighbors with their mapped interger ids

\*/

def getMappedSimsGraph(

mappedUsers: TypedPipe[(Int, UserId)],

allEdges: TypedPipe[Candidates],

maxNeighborsPerNode: Int

)(

implicit uniqueId: UniqueID

): TypedPipe[(Int, List[(Int, Float)])] = {

val numEdgesAfterFirstJoin = Stat("num\_edges\_after\_first\_join")

val numEdgesAfterSecondJoin = Stat("num\_edges\_after\_second\_join")

val numEdgesLostTopKTruncated = Stat("num\_edges\_lost\_topk\_truncated")

val finalNumEdges = Stat("final\_num\_edges")

val mappedUserIdsToIds: TypedPipe[(UserId, Int)] = mappedUsers.swap

allEdges

.map { cs => (cs.userId, cs.candidates) }

// filter the users not present in the mapped userIDs list

.join(mappedUserIdsToIds)

.withReducers(6000)

.flatMap {

case (id, (neighbors, mappedId)) =>

val before = neighbors.size

val topKNeighbors = neighbors.sortBy(-\_.score).take(maxNeighborsPerNode)

val after = topKNeighbors.size

numEdgesLostTopKTruncated.incBy(before - after)

topKNeighbors.map { candidate =>

numEdgesAfterFirstJoin.inc()

(candidate.userId, (mappedId, candidate.score.toFloat))

}

}

.join(mappedUserIdsToIds)

.withReducers(9000)

.flatMap {

case (id, ((mappedNeighborId, score), mappedId)) =>

numEdgesAfterSecondJoin.inc()

// to make the graph symmetric, add those edges back that might have been filtered

// due to maxNeighborsPerNodefor a user but not for its neighbors

List(

(mappedId, Map(mappedNeighborId -> Max(score))),

(mappedNeighborId, Map(mappedId -> Max(score)))

)

}

.sumByKey

.withReducers(9100)

.map {

case (id, nbrMap) =>

// Graph initialization expects neighbors to be sorted in ascending order of ids

val sorted = nbrMap.mapValues(\_.get).toList.sortBy(\_.\_1)

finalNumEdges.incBy(sorted.size)

(id, sorted)

}

}

def getTopFollowedUsersWithMappedIds(

minActiveFollowers: Int,

topK: Int

)(

implicit uniqueId: UniqueID,

timeZone: TimeZone

): TypedPipe[(Long, Int)] = {

val numTopUsersMappings = Stat("num\_top\_users\_with\_mapped\_ids")

println("Going to include mappedIds in output")

TopUsersSimilarityGraph

.topUsersWithMappedIdsTopK(

DAL

.readMostRecentSnapshotNoOlderThan(

UsersourceFlatScalaDataset,

Days(30)).withRemoteReadPolicy(ExplicitLocation(ProcAtla)).toTypedPipe,

minActiveFollowers,

topK

)

.map {

case TopUserWithMappedId(TopUser(id, activeFollowerCount, screenName), mappedId) =>

numTopUsersMappings.inc()

(id, mappedId)

}

}

/\*\*

\* Map the userIds in the knownFor dataset to their integer Ids .

\*/

def getKnownForWithMappedIds(

knownForDataset: TypedPipe[(UserId, Array[(ClusterId, Float)])], //original userId as the key

mappedIdsWithUserId: TypedPipe[(Int, UserId)] //mapped userId as the key

): TypedPipe[(Int, List[(ClusterId, Float)])] = {

val userIdsAndTheirMappedIndices = mappedIdsWithUserId.map {

case (mappedId, originalId) => (originalId, mappedId)

}

knownForDataset.join(userIdsAndTheirMappedIndices).map {

case (userId, (userClusterArray, mappedUserId)) =>

(mappedUserId, userClusterArray.toList)

}

}

/\*\*

\* Attach the cluster assignments from knownFor dataset to the users in mapped Sims graph .

\*/

def attachClusterAssignments(

mappedSimsGraph: TypedPipe[(Int, List[(Int, Float)])],

knownForAssignments: TypedPipe[(Int, List[(ClusterId, Float)])],

squareWeights: Boolean

)(

implicit uniqueId: UniqueID

): TypedPipe[(Int, Array[Int], Array[Float], List[(ClusterId, Float)])] = {

val numPopularUsersWithNoKnownForBefore = Stat(

"num\_popular\_users\_with\_no\_knownfor\_before\_but\_popular\_now")

val input = mappedSimsGraph.map {

case (id, nbrsList) =>

val ngbrIds = nbrsList.map(\_.\_1).toArray

val ngbrWts = if (squareWeights) {

nbrsList.map(\_.\_2).map(currWt => currWt \* currWt \* 10).toArray

} else {

nbrsList.map(\_.\_2).toArray

}

(id, ngbrIds, ngbrWts)

}

// input simsGraph consists of popular ppl with most followed users, who might not have been

// a knownFor user in the previous week. So left join with the knownFor dataset, and these

// new popular users will not have any prior cluster assignments while clustering this time

input

.groupBy(\_.\_1)

.leftJoin(knownForAssignments.groupBy(\_.\_1))

.toTypedPipe

.map {

case (mappedUserId, ((mappedId, ngbrIds, ngbrWts), knownForResult)) =>

val clustersList: List[(Int, Float)] = knownForResult match {

case Some(values) => values.\_2

case None =>

numPopularUsersWithNoKnownForBefore.inc()

List.empty

}

(mappedUserId, ngbrIds, ngbrWts, clustersList)

}

}

/\*\*

\* Initialize graph with users and neighbors with edge weights .

\*/

def getGraphFromSimsInput(

mappedSimsIter: Iterable[

(Int, Array[Int], Array[Float], List[(ClusterId, Float)])

],

numUsers: Int

): Graph = {

val nbrsIds: Array[Array[Int]] = new Array[Array[Int]](numUsers)

val nbrsWts: Array[Array[Float]] = new Array[Array[Float]](numUsers)

var numEdges = 0L

var numVertices = 0

var numVerticesWithNoNgbrs = 0

mappedSimsIter.foreach {

case (id, nbrArrayIds, nbArrayScores, \_) =>

nbrsIds(id) = nbrArrayIds

nbrsWts(id) = nbArrayScores

numEdges += nbrArrayIds.length

numVertices += 1

if (numVertices % 100000 == 0) {

println(s"Done loading $numVertices many vertices. Edges so far: $numEdges")

}

}

(0 until numUsers).foreach { i =>

if (nbrsIds(i) == null) {

numVerticesWithNoNgbrs += 1

nbrsIds(i) = Array[Int]()

nbrsWts(i) = Array[Float]()

}

}

println(

s"Done loading graph with $numUsers nodes and $numEdges edges (counting each edge twice)")

println("Number of nodes with at least one neighbor is " + numVertices)

println("Number of nodes with at no neighbors is " + numVerticesWithNoNgbrs)

new Graph(numUsers, numEdges / 2, nbrsIds, nbrsWts)

}

/\*\*

\* Helper function that initializes users to clusters based on previous knownFor assignments

\* and for users with no previous assignments, assign them randomly to any of the empty clusters

\*/

def initializeSparseBinaryMatrix(

graph: Graph,

mappedSimsGraphIter: Iterable[

(Int, Array[Int], Array[Float], List[(ClusterId, Float)])

], // user with neighbors, neighbor wts and previous knownfor assignments

numUsers: Int,

numClusters: Int,

algoConfig: AlgorithmConfig,

): SparseBinaryMatrix = {

var clustersSeenFromPreviousWeek: Set[Int] = Set.empty

var emptyClustersFromPreviousWeek: Set[Int] = Set.empty

var usersWithNoAssignmentsFromPreviousWeek: Set[Int] = Set.empty

mappedSimsGraphIter.foreach {

case (id, \_, \_, knownFor) =>

if (knownFor.isEmpty) {

usersWithNoAssignmentsFromPreviousWeek += id

}

knownFor.foreach {

case (clusterId, \_) =>

clustersSeenFromPreviousWeek += clusterId

}

}

(1 to numClusters).foreach { i =>

if (!clustersSeenFromPreviousWeek.contains(i)) emptyClustersFromPreviousWeek += i

}

var z = new SparseBinaryMatrix(numUsers, numClusters)

println("Going to initialize from previous KnownFor")

var zeroIndexedClusterIdsFromPreviousWeek: Set[Int] = Set.empty

for (clusterIdOneIndexed <- emptyClustersFromPreviousWeek) {

zeroIndexedClusterIdsFromPreviousWeek += (clusterIdOneIndexed - 1)

}

// Initialize z - users with no previous assignments are assigned to empty clusters

z.initFromSubsetOfRowsForSpecifiedColumns(

graph,

(gr: Graph, i: Integer) => algoConfig.rng.nextDouble,

zeroIndexedClusterIdsFromPreviousWeek.toArray,

usersWithNoAssignmentsFromPreviousWeek.toArray,

new PrintWriter(System.err)

)

println("Initialized the empty clusters")

mappedSimsGraphIter.foreach {

case (id, \_, \_, knownFor) =>

val currClustersForUserZeroIndexed = knownFor.map(\_.\_1).map(x => x - 1)

// Users who have a previous cluster assignment are initialized with the same cluster

if (currClustersForUserZeroIndexed.nonEmpty) {

z.updateRow(id, currClustersForUserZeroIndexed.sorted.toArray)

}

}

println("Done initializing from previous knownFor assignment")

z

}

/\*\*

\* Optimize the sparseBinaryMatrix. This function runs the clustering epochs and computes the

\* cluster assignments for the next week, based on the underlying user-user graph

\*/

def optimizeSparseBinaryMatrix(

algoConfig: AlgorithmConfig,

graph: Graph,

z: SparseBinaryMatrix

): SparseBinaryMatrix = {

val prec0 = MHAlgorithm.clusterPrecision(graph, z, 0, 1000, algoConfig.rng)

println("Precision of cluster 0:" + prec0.precision)

val prec1 = MHAlgorithm.clusterPrecision(graph, z, 1, 1000, algoConfig.rng)

println("Precision of cluster 1:" + prec1.precision)

val algo = new MHAlgorithm(algoConfig, graph, z, new PrintWriter(System.err))

val optimizedZ = algo.optimize

optimizedZ

}

/\*\*

\* Helper function that takes the heuristically scored association of user to a cluster

\* and returns the knownFor result

\* @param srm SparseRealMatrix with (row, col) score denoting the membership score of user in the cluster

\* @return assignments of users (mapped integer indices) to clusters with knownFor scores.

\*/

def getKnownForHeuristicScores(srm: SparseRealMatrix): List[(Int, List[(ClusterId, Float)])] = {

val knownForAssignmentsFromClusterScores = (0 until srm.getNumRows).map { rowId =>

val rowWithIndices = srm.getColIdsForRow(rowId)

val rowWithScores = srm.getValuesForRow(rowId)

val allClustersWithScores: Array[(ClusterId, Float)] =

rowWithIndices.zip(rowWithScores).map {

case (colId, score) => (colId + 1, score.toFloat)

}

if (rowId % 100000 == 0) {

println("Inside outputIter:" + rowId + " " + srm.getNumRows)

}

val clusterAssignmentWithMaxScore: List[(ClusterId, Float)] =

if (allClustersWithScores.length > 1) {

// if sparseBinaryMatrix z has rows with more than one non-zero column (i.e a user

// initialized with more than one cluster), and the clustering algorithm doesnot find

// a better proposal for cluster assignment, the user's multi-cluster membership

// from the initialization step can continue.

// We found that this happens in ~0.1% of the knownFor users. Hence choose the

// cluster with the highest score to deal with such edge cases.

val result: (ClusterId, Float) = allClustersWithScores.maxBy(\_.\_2)

println(

"Found a user with mappedId: %s with more than 1 cluster assignment:%s; Assigned to the best cluster: %s"

.format(

rowId.toString,

allClustersWithScores.mkString("Array(", ", ", ")"),

result

.toString()))

List(result)

} else {

allClustersWithScores.toList

}

(rowId, clusterAssignmentWithMaxScore)

}

knownForAssignmentsFromClusterScores.toList

}

/\*\*

\* Function that computes the clustering assignments to users

\*

\* @param mappedSimsGraph user-user graph as input to clustering

\* @param previousKnownForAssignments previous week clustering assignments

\* @param maxEpochsForClustering number of neighbors for each user

\* @param squareWeights boolean flag for the edge weights in the sims graph

\* @param wtCoeff wtCoeff

\*

\* @return users with clusters assigned

\*/

def getClusteringAssignments(

mappedSimsGraph: TypedPipe[(Int, List[(Int, Float)])],

previousKnownForAssignments: TypedPipe[(Int, List[(ClusterId, Float)])],

maxEpochsForClustering: Int,

squareWeights: Boolean,

wtCoeff: Double

)(

implicit uniqueId: UniqueID

): Execution[List[(Int, List[(ClusterId, Float)])]] = {

attachClusterAssignments(

mappedSimsGraph,

previousKnownForAssignments,

squareWeights).toIterableExecution.flatMap { mappedSimsGraphWithClustersIter =>

val tic = System.currentTimeMillis

var maxVertexId = 0

var maxClusterIdInPreviousAssignment = 0

mappedSimsGraphWithClustersIter.foreach {

case (id, \_, \_, knownFor) =>

maxVertexId = Math.max(id, maxVertexId)

knownFor.foreach {

case (clusterId, \_) =>

maxClusterIdInPreviousAssignment =

Math.max(clusterId, maxClusterIdInPreviousAssignment)

}

}

val numUsersToCluster =

maxVertexId + 1 //since users were mapped with index starting from 0, using zipWithIndex

println("Total number of topK users to be clustered this time:" + numUsersToCluster)

println(

"Total number of clusters in the previous knownFor assignment:" + maxClusterIdInPreviousAssignment)

println("Will set number of communities to " + maxClusterIdInPreviousAssignment)

// Initialize the graph with users, neighbors and the corresponding edge weights

val graph = getGraphFromSimsInput(mappedSimsGraphWithClustersIter, numUsersToCluster)

val toc = System.currentTimeMillis()

println("Time to load the graph " + (toc - tic) / 1000.0 / 60.0 + " minutes")

// define the algoConfig parameters

val algoConfig = new AlgorithmConfig()

.withCpu(16).withK(maxClusterIdInPreviousAssignment)

.withWtCoeff(wtCoeff.toDouble)

.withMaxEpoch(maxEpochsForClustering)

algoConfig.divideResultIntoConnectedComponents = false

algoConfig.minClusterSize = 1

algoConfig.updateImmediately = true

algoConfig.rng = new RandomAdaptor(new JDKRandomGenerator(1))

// Initialize a sparseBinaryMatrix with users assigned to their previous week knownFor

// assignments. For those users who do not a prior assignment, we assign

// the (user + the neighbors from the graph) to the empty clusters.

// Please note that this neighborhood-based initialization to empty clusters can

// have a few cases where the same user was assigned to more than one cluster

val z = initializeSparseBinaryMatrix(

graph,

mappedSimsGraphWithClustersIter,

numUsersToCluster,

maxClusterIdInPreviousAssignment,

algoConfig

)

// Run the epochs of the clustering algorithm to find the new cluster assignments

val tic2 = System.currentTimeMillis

val optimizedZ = optimizeSparseBinaryMatrix(algoConfig, graph, z)

val toc2 = System.currentTimeMillis

println("Time to optimize: %.2f seconds\n".format((toc2 - tic2) / 1000.0))

println("Time to initialize & optimize: %.2f seconds\n".format((toc2 - toc) / 1000.0))

// Attach scores to the cluster assignments

val srm = MHAlgorithm.heuristicallyScoreClusterAssignments(graph, optimizedZ)

// Get the knownfor assignments of users from the heuristic scores

// assigned based on neigbhorhood of the user and their cluster assignments

// The returned result has userIDs in the mapped integer indices

val knownForAssignmentsFromClusterScores: List[(Int, List[(ClusterId, Float)])] =

getKnownForHeuristicScores(srm)

Execution.from(knownForAssignmentsFromClusterScores)

}

}

}