package com.twitter.simclusters\_v2.scalding

import com.twitter.dal.client.dataset.KeyValDALDataset

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

import com.twitter.pluck.source.cassowary.FollowingsCosineSimilaritiesManhattanSource

import com.twitter.pluck.source.cassowary.SimsCandidatesSource

import com.twitter.scalding.\_

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

import com.twitter.scalding\_internal.dalv2.DALWrite.\_

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

import com.twitter.scalding\_internal.job.analytics\_batch.AnalyticsBatchExecution

import com.twitter.scalding\_internal.job.analytics\_batch.AnalyticsBatchExecutionArgs

import com.twitter.scalding\_internal.job.analytics\_batch.BatchDescription

import com.twitter.scalding\_internal.job.analytics\_batch.BatchFirstTime

import com.twitter.scalding\_internal.job.analytics\_batch.BatchIncrement

import com.twitter.scalding\_internal.job.analytics\_batch.TwitterScheduledExecutionApp

import com.twitter.scalding\_internal.multiformat.format.keyval.KeyVal

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

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

import com.twitter.simclusters\_v2.scalding.UpdateKnownFor.ClusterScoresForNode

import com.twitter.simclusters\_v2.scalding.UpdateKnownFor.NeighborhoodInformation

import com.twitter.simclusters\_v2.scalding.common.TypedRichPipe.\_

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

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

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

import scala.util.Success

object UpdateKnownForApps {

/\*\*

\* Average edge weight of an input graph

\* @param graph a TypedPipe with nodeId as key and adjacency list as value. We don't care about

\* the keys in this method.

\* @return avg edge weight wrapped in an option in an execution

\*/

def getGlobalAvgWeight(graph: TypedPipe[(Long, Map[Long, Float])]): Execution[Option[Double]] = {

graph.values

.flatMap(\_.values)

.map { x => (x.toDouble, 1L) }

.sum

.toOptionExecution

.map {

case Some((sum, cnt)) =>

val res = sum / cnt

println("globalAvgWeight is " + res)

Some(res)

case \_ =>

println("Input graph to globalAvgWeight seems to be empty")

None

}

}

/\*\*

\* Average membership score for a particular knownFor assignment

\* @param knownFor TypedPipe from nodeId to the clusters it's been assigned to along with

\* membership scores. We don't care about the keys in this method.

\* @return average membership score

\*/

def getAvgMembershipScore(knownFor: TypedPipe[(Long, Array[(Int, Float)])]): Execution[Double] = {

knownFor.values

.flatMap(\_.map(\_.\_2))

.map { x => (x, 1L) }

.sum

.map { case (num, den) => num / den.toDouble }

.getExecution

.onComplete {

case Success(x) => println("Avg. membership score is " + x)

case \_ => println("Failed to calculate avg. membership score")

}

}

/\*\*

\* For each cluster, get two statistics about it: the number of nodes assigned to it, and the

\* sum of the membership scores

\*

\* @param knownFor TypedPipe from nodeId to the clusters it's been assigned to along with

\* membership scores.

\* @return Map giving the NeighborhoodInformation for each cluster. The nodeCount and

\* sumOfMembershipWeights fields in NeighborhoodInformation are populated, others are 0.

\*/

def getClusterStats(

knownFor: TypedPipe[(Long, Array[(Int, Float)])]

): Execution[Map[Int, NeighborhoodInformation]] = {

knownFor

.flatMap {

case (\_, clusterArray) =>

clusterArray.map {

case (clusterId, score) =>

Map(clusterId -> (1, score))

}

}

.sum

.getExecution

.map { map =>

map.mapValues {

case (count, sum) =>

NeighborhoodInformation(count, 0, 0, sum)

}

}

}

/\*\*

\* Adds self-loops and also potentially raises all edge weights to an exponent

\* (typically exponent > 1, and has the effect of increasing inequality in edge weights to

\* "clarify" structure in the graph - currently we just set exponent to 1).

\* @param symmetrizedSims input symmetrized similarity graph

\* @param exponentForEdgeWeight exponent to raise all edge weights to.

\* Set to 1.0 to make this a no-op

\* @param maxWtToSelfLoopWtMultFactor What to multiply the max wt among non-self-loop edges to

\* derive the weight on the self-loop edge.

\* @return New graph

\*/

def simsGraphForUpdateFromSymmetrizedSims(

symmetrizedSims: TypedPipe[(Long, Map[Long, Float])],

exponentForEdgeWeight: Float,

maxWtToSelfLoopWtMultFactor: Float

): TypedPipe[(Long, Map[Long, Float])] = {

val expWeighted = symmetrizedSims.mapValues { y =>

y.mapValues { x => math.pow(x, exponentForEdgeWeight).toFloat }

}

TopUsersSimilarityGraph.addSelfLoop(

input = expWeighted,

maxToSelfLoopWeight = { x: Float => x \* maxWtToSelfLoopWtMultFactor }

)

}

/\*\*

\* Runs the job

\* @param args args which specify many parameters

\* @param inputKnownFor

\* @param inputSimsGraph

\* @param defaultEmailAddress by default, the email address to send an to email to, which has

\* a bunch of evaluation metrics

\* @param writeKnownForFunction function that takes a knownFor and writes to some

\* persistent location

\* @param readKnownForFunction function that reads the knownFor which was written to using the

\* writeKnownForFunction

\* @param dateRange dateRange, used for reading UserSource

\* @param uniqueID need for creating stats

\* @return Execution[Unit] encapsulating the whole job

\*/

def runUpdateKnownForGeneric(

args: Args,

inputKnownFor: TypedPipe[(Long, Array[(Int, Float)])],

inputSimsGraph: TypedPipe[Candidates],

defaultEmailAddress: String,

writeKnownForFunction: TypedPipe[(Long, Array[(Int, Float)])] => Execution[Unit],

readKnownForFunction: => TypedPipe[(Long, Array[(Int, Float)])],

includeEvaluationResultsInEmail: Boolean

)(

implicit dateRange: DateRange,

uniqueID: UniqueID

): Execution[Unit] = {

val minActiveFollowers = args.int("minActiveFollowers", 400)

val topK = args.int("topK")

val maxSimsNeighborsForUpdate =

args.int("maxSimsNeighborsForUpdate", 40)

val minNeighborsInCluster = args.int("minNeighborsInCluster", 2)

val maxWtToSelfLoopWtMultFactor =

args.float("maxWtToSelfLoopWtMultFactor", 2)

val exponentForEdgeWeight = args.float("exponentForEdgeWeights", 1.0f)

val updateMethod: ClusterScoresForNode => Double = args("updateMethod") match {

case "sumScoreIgnoringMembershipScores" => { x: ClusterScoresForNode =>

x.sumScoreIgnoringMembershipScores

}

case "ratioScoreIgnoringMembershipScores" => { x: ClusterScoresForNode =>

x.ratioScoreIgnoringMembershipScores

}

case "ratioScoreUsingMembershipScores" => { x: ClusterScoresForNode =>

x.ratioScoreUsingMembershipScores

}

case x @ \_ =>

throw new Exception(s"value for --updateMethod $x is unknown. It must be one of " +

s"[sumScoreIgnoringMembershipScores, ratioScoreIgnoringMembershipScores, ratioScoreUsingMembershipScores]")

}

val truePositiveWtFactor = args.float("truePositiveWtFactor", 10)

val modelVersion = args("outputModelVersion")

val emailAddress =

args.optional("emailAddress").getOrElse(defaultEmailAddress)

val topUsers = TopUsersSimilarityGraph

.topUserIds(

DAL

.readMostRecentSnapshot(UsersourceFlatScalaDataset, dateRange)

.toTypedPipe,

minActiveFollowers,

topK).count("num\_top\_users")

TopUsersSimilarityGraph

.getSubgraphFromUserGroupedInput(

fullGraph = inputSimsGraph,

usersToInclude = topUsers,

maxNeighborsPerNode = maxSimsNeighborsForUpdate,

degreeThresholdForStat = minNeighborsInCluster

)

.forceToDiskExecution

.flatMap { symmetrizedSims =>

val modifiedSims =

UpdateKnownForApps.simsGraphForUpdateFromSymmetrizedSims(

symmetrizedSims = symmetrizedSims,

exponentForEdgeWeight = exponentForEdgeWeight,

maxWtToSelfLoopWtMultFactor = maxWtToSelfLoopWtMultFactor

)

val previouslyFamousUsersExec = inputKnownFor

.leftJoin(topUsers.asKeys)

.collect { case (userId, (clusters, None)) => userId }

.getSummaryString(

"Users previously in known for but not in topUsers anymore",

numRecords = 20)

val clusterStatsExec = UpdateKnownForApps.getClusterStats(inputKnownFor)

val globalAvgWeightExec =

UpdateKnownForApps.getGlobalAvgWeight(modifiedSims)

val globalAvgMembershipScoreExec = UpdateKnownForApps.getAvgMembershipScore(inputKnownFor)

Execution.zip(globalAvgWeightExec, clusterStatsExec, globalAvgMembershipScoreExec).flatMap {

case (Some(globalAvgWeight), clusterStats, globalAvgMembershipScore) =>

println("Size of clusterStats: " + clusterStats.size)

println("First few entries from clusterStats: " + clusterStats.take(5))

println("globalAvgWeight: " + globalAvgWeight)

println("globalAvgMembershipScore: " + globalAvgMembershipScore)

val knownForWithUnnormalizedScores = UpdateKnownFor

.newKnownForScores(

inputKnownFor,

modifiedSims,

globalAvgWeight,

clusterStats,

globalAvgMembershipScore

)

val writeNewKnownForExec = writeKnownForFunction(

UpdateKnownFor.updateGeneric(

modifiedSims,

knownForWithUnnormalizedScores,

clusterStats,

minNeighborsInCluster,

globalAvgWeight,

globalAvgMembershipScore,

truePositiveWtFactor,

updateMethod

)

)

writeNewKnownForExec.flatMap { \_ =>

Util.getCustomCountersString(writeNewKnownForExec).flatMap { customCountersString =>

if (includeEvaluationResultsInEmail) {

// It's unfortunate that we're not using the newKnownFor directly, but are instead

// first writing it out and then reading it back in. The reason for doing it in this

// convoluted way is that when we directly use the newKnownFor, the clusterEvaluation

// metrics are being incorrectly computed.

val newKnownFor = readKnownForFunction

val newResultsExec =

ClusterEvaluation

.overallEvaluation(symmetrizedSims, newKnownFor, "newKnownForEval")

val oldResultsExec =

ClusterEvaluation

.overallEvaluation(symmetrizedSims, inputKnownFor, "oldKnownForEval")

val minSizeOfBiggerClusterForComparison = 10

val compareExec = CompareClusters.summarize(

CompareClusters.compare(

KnownForSources.transpose(inputKnownFor),

KnownForSources.transpose(newKnownFor),

minSizeOfBiggerCluster = minSizeOfBiggerClusterForComparison

))

Execution

.zip(oldResultsExec, newResultsExec, compareExec, previouslyFamousUsersExec)

.map {

case (oldResults, newResults, compareResults, previouslyFamousUsersString) =>

val emailText = "Evaluation Results for existing knownFor:\n" +

Util.prettyJsonMapper.writeValueAsString(oldResults) +

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

"Evaluation Results for new knownFor:\n" +

Util.prettyJsonMapper.writeValueAsString(newResults) +

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

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

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

Util.prettyJsonMapper

.writeValueAsString(compareResults) +

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

"Custom counters:\n" + customCountersString +

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

previouslyFamousUsersString

Util

.sendEmail(

emailText,

s"Evaluation results of new knownFor $modelVersion",

emailAddress)

}

} else {

Util

.sendEmail(

customCountersString,

s"Change in cluster assignments for update of knownFor $modelVersion",

emailAddress

)

Execution.unit

}

}

}

}

}

}

}

trait UpdateKnownForBatch extends TwitterScheduledExecutionApp {

implicit val tz: java.util.TimeZone = DateOps.UTC

implicit val dp = DateParser.default

def firstTime: String

val batchIncrement: Duration = Days(30)

def batchDescription: String

private lazy val execArgs = AnalyticsBatchExecutionArgs(

batchDesc = BatchDescription(batchDescription),

firstTime = BatchFirstTime(RichDate(firstTime)),

lastTime = None,

batchIncrement = BatchIncrement(batchIncrement)

)

val emailAddress: String = "no-reply@twitter.com"

def inputDALDataset: KeyValDALDataset[KeyVal[Long, ClustersUserIsKnownFor]]

def inputModelVersion: String

def outputModelVersion: String

def outputPath: String

def outputDALDataset: KeyValDALDataset[KeyVal[Long, ClustersUserIsKnownFor]]

override def scheduledJob: Execution[Unit] =

AnalyticsBatchExecution(execArgs) { implicit dateRange =>

Execution.withId { implicit uniqueId =>

Execution.withArgs { args =>

val inputKnownFor =

KnownForSources.readDALDataset(inputDALDataset, Days(30), inputModelVersion)

val inputSimsGraph = TypedPipe

.from(FollowingsCosineSimilaritiesManhattanSource())

.map(\_.\_2)

def writeKnownFor(knownFor: TypedPipe[(Long, Array[(Int, Float)])]): Execution[Unit] = {

KnownForSources

.toKeyVal(knownFor, outputModelVersion)

.writeDALVersionedKeyValExecution(

outputDALDataset,

D.Suffix(outputPath)

)

}

def readKnownFor =

KnownForSources.readDALDataset(outputDALDataset, Days(1), outputModelVersion)

UpdateKnownForApps.runUpdateKnownForGeneric(

args,

inputKnownFor,

inputSimsGraph,

emailAddress,

writeKnownFor,

readKnownFor,

includeEvaluationResultsInEmail = false

)

}

}

}

}

/\*\*

capesospy-v2 update --build\_locally --start\_cron update\_known\_for\_20M\_145k \

src/scala/com/twitter/simclusters\_v2/capesos\_config/atla\_proc.yaml

\*/

object UpdateKnownFor20M145K extends UpdateKnownForBatch {

override val firstTime: String = "2019-06-06"

override val batchIncrement: Duration = Days(7)

override val batchDescription: String =

"com.twitter.simclusters\_v2.scalding.UpdateKnownFor20M145K"

override val inputModelVersion: String = ModelVersions.Model20M145KUpdated

override val inputDALDataset: KeyValDALDataset[KeyVal[Long, ClustersUserIsKnownFor]] =

SimclustersV2RawKnownFor20M145KUpdatedScalaDataset

override val outputModelVersion: String = ModelVersions.Model20M145KUpdated

override val outputDALDataset: KeyValDALDataset[KeyVal[Long, ClustersUserIsKnownFor]] =

SimclustersV2RawKnownFor20M145KUpdatedScalaDataset

override val outputPath: String = InternalDataPaths.RawKnownForUpdatedPath

}

/\*\* This one's end-to-end, doesn't save any intermediate data etc. \*\*/

object UpdateKnownForAdhoc extends TwitterExecutionApp {

implicit val tz: java.util.TimeZone = DateOps.UTC

implicit val dp = DateParser.default

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

implicit val date: DateRange = DateRange.parse(args("date"))

val defaultEmailAddress = "your\_ldap@twitter.com"

val inputKnownFor = args.optional("inputKnownForDir") match {

case Some(inputKnownForDir) => KnownForSources.readKnownFor(inputKnownForDir)

case None => KnownForSources.knownFor\_20M\_Dec11\_145K

}

val inputSimsGraph = TopUsersSimilarityGraph.readSimsInput(

args.boolean("simsInputIsKeyValSource"),

args("simsInputDir")

)

def readKnownFor() = KnownForSources.readKnownFor(args("outputDir"))

UpdateKnownForApps.runUpdateKnownForGeneric(

args,

inputKnownFor,

inputSimsGraph,

defaultEmailAddress,

{ input: TypedPipe[(Long, Array[(Int, Float)])] =>

KnownForSources.writeKnownForTypedTsv(input, args("outputDir"))

},

readKnownFor,

includeEvaluationResultsInEmail = true

)

}

}

}