package com.twitter.simclusters\_v2.scalding

import com.twitter.algebird.Aggregator

import com.twitter.algebird.Monoid

import com.twitter.scalding.\_

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.scalding\_internal.job.TwitterExecutionApp

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

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

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

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

import com.twitter.simclusters\_v2.hdfs\_sources.SimclustersV2InterestedInScalaDataset

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

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

import com.twitter.simclusters\_v2.scalding.BipartiteClusterEvaluationClasses.\_

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

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

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

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

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

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

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

import scala.collection.JavaConverters.\_

object BipartiteClusterEvaluation extends TwitterExecutionApp {

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

implicit val dp = DateParser.default

private def getClusterL2Norms(

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

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

knownFor

.flatMap {

case (\_, clusterArray) =>

clusterArray.map {

case (clusterId, score) =>

Map(clusterId -> score \* score)

}

}

.sum

.getExecution

.map(\_.mapValues { x => math.sqrt(x).toFloat })

}

def l2NormalizeKnownFor(

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

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

getClusterL2Norms(knownFor).map { clusterToNorms =>

knownFor.mapValues { clusterScoresArray =>

clusterScoresArray.map {

case (clusterId, score) =>

(clusterId, score / clusterToNorms(clusterId))

}

}

}

}

/\*\*

\* ./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding:bp\_cluster\_evaluation && \

\* oscar hdfs --user frigate --host hadoopnest2.atla.twitter.com --bundle bp\_cluster\_evaluation \

\* --tool com.twitter.simclusters\_v2.scalding.BipartiteClusterEvaluation --screen --screen-detached \

\* --tee logs/newBpQuality\_updateUnnormalizedScores\_interestedInUsing20190329Graph\_evaluatedOn20190329Graph\_run2 \

\* -- --normsAndCountsDir /user/frigate/your\_ldap/producerNormsAndCounts\_20190330 \

\* --graphInputDir /user/frigate/your\_ldap/user\_user\_normalized\_graph\_copiedFromAtlaProc\_20190329 \

\* --knownForDir /user/frigate/your\_ldap/dirFor\_updatedKnownFor20M\_145K\_dec11\_usingSims20190127\_unnormalizedInputScores/knownFor \

\* --interestedInDir /user/frigate/your\_ldap/dirFor\_updatedKnownFor20M\_145K\_dec11\_usingSims20190127\_unnormalizedInputScores/interestedInUsing20190329Graph \

\* --outgoingVolumesResultsDir /user/frigate/your\_ldap/dirFor\_updatedKnownFor20M\_145K\_dec11\_usingSims20190127\_unnormalizedInputScores/bpQualityForInterestedInUsing20190329On20190329Graph\_outgoingVolumes \

\* --incomingVolumesResultsDir /user/frigate/your\_ldap/dirFor\_updatedKnownFor20M\_145K\_dec11\_usingSims20190127\_unnormalizedInputScores/bpQualityForInterestedInUsing20190329On20190329Graph\_incomingVolumes \

\* --outputDir /user/frigate/your\_ldap/dirFor\_updatedKnownFor20M\_145K\_dec11\_usingSims20190127\_unnormalizedInputScores/bpQualityForInterestedInUsing20190329On20190329Graph\_perCluster \

\* --toEmailAddress your\_ldap@twitter.com --modelVersion 20M\_145K\_updated

\*/

override def job: Execution[Unit] = Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

val interestedIn = args.optional("interestedInDir") match {

case Some(dir) =>

TypedPipe

.from(AdhocKeyValSources.interestedInSource(args("interestedInDir")))

case None =>

DAL

.readMostRecentSnapshotNoOlderThan(

SimclustersV2InterestedInScalaDataset,

Days(20)

)

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

.map {

case KeyVal(key, value) => (key, value)

}

}

val inputKnownFor = args

.optional("knownForDir")

.map { location => KnownForSources.readKnownFor(location) }

.getOrElse(KnownForSources.knownFor\_20M\_Dec11\_145K)

val modelVersion =

args.optional("modelVersion").getOrElse("20M\_145K\_dec11")

val useLogFavWeights = args.boolean("useLogFavWeights")

val shouldL2NormalizeKnownFor = args.boolean("l2NormalizeKnownFor")

val toEmailAddressOpt = args.optional("toEmailAddress")

val knownForExec = if (shouldL2NormalizeKnownFor) {

l2NormalizeKnownFor(inputKnownFor)

} else {

Execution.from(inputKnownFor)

}

val finalExec = knownForExec.flatMap { knownFor =>

val graph = args.optional("graphInputDir") match {

case Some(dir) =>

TypedPipe.from(UserAndNeighborsFixedPathSource(dir))

case None =>

DAL

.readMostRecentSnapshotNoOlderThan(UserUserNormalizedGraphScalaDataset, Days(20))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

}

val producerNormsAndCounts = args.optional("normsAndCountsDir") match {

case Some(dir) =>

TypedPipe.from(NormsAndCountsFixedPathSource(args(dir)))

case None =>

DAL

.readMostRecentSnapshotNoOlderThan(ProducerNormsAndCountsScalaDataset, Days(20))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

}

val clusterIncomingVolumesExec = loadOrMake(

computeClusterIncomingVolumes(knownFor, producerNormsAndCounts, useLogFavWeights),

modelVersion,

args("incomingVolumesResultsDir")

)

val resultsWithOutgoingVolumesExec = loadOrMake(

getResultsWithOutgoingVolumes(graph, interestedIn, useLogFavWeights),

modelVersion,

args("outgoingVolumesResultsDir")

)

val finalPerClusterResultsExec =

finalPerClusterResults(

knownFor,

interestedIn,

resultsWithOutgoingVolumesExec,

clusterIncomingVolumesExec)

.flatMap { pipe => loadOrMake(pipe, modelVersion, args("outputDir")) }

finalPerClusterResultsExec.flatMap { finalPerClusterResults =>

val perClusterResults = finalPerClusterResults.values

val distributionResultsExec = getClusterResultsSummary(perClusterResults).map {

case Some(summary) =>

"Summary of results across clusters: \n" +

Util.prettyJsonMapper.writeValueAsString(summary)

case \_ =>

"No summary of results! The cluster level results pipe must be empty!"

}

val overallResultsExec = perClusterResults.sum.toOptionExecution.map {

case Some(overallQuality) =>

"Overall Quality: \n" +

Util.prettyJsonMapper.writeValueAsString(

printableBipartiteQuality(overallQuality)

)

case \_ =>

"No overall quality! The cluster level results pipe must be empty!"

}

Execution.zip(distributionResultsExec, overallResultsExec).map {

case (distResults, overallResults) =>

toEmailAddressOpt.foreach { address =>

Util.sendEmail(

distResults + "\n" + overallResults,

"Bipartite cluster quality for " + modelVersion,

address

)

}

println(distResults + "\n" + overallResults)

}

}

}

Util.printCounters(finalExec)

}

}

def getResultsWithOutgoingVolumes(

graph: TypedPipe[UserAndNeighbors],

interestedIn: TypedPipe[(Long, ClustersUserIsInterestedIn)],

useLogFavWeights: Boolean

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

graph

.map { un => (un.userId, un.neighbors) }

// should this be a leftJoin? For now, leaving it as an inner join. If in the future,

// we want to compare two approaches with very different coverages on interestedIn, this

// could become a problem.

.join(interestedIn)

.withReducers(4000)

.flatMap {

case (userId, (neighbors, clusters)) =>

getBIResultsFromSingleUser(userId, neighbors, clusters, useLogFavWeights)

}

.sumByKey

.withReducers(600)

.map {

case (clusterId, bir) =>

(

clusterId,

BipartiteClusterQuality(

inClusterFollowEdges = Some(bir.inClusterWeights.isFollowEdge),

inClusterFavEdges = Some(bir.inClusterWeights.isFavEdge),

favWtSumOfInClusterFollowEdges = Some(bir.inClusterWeights.favWtIfFollowEdge),

favWtSumOfInClusterFavEdges = Some(bir.inClusterWeights.favWtIfFavEdge),

outgoingFollowEdges = Some(bir.totalOutgoingVolumes.isFollowEdge),

outgoingFavEdges = Some(bir.totalOutgoingVolumes.isFavEdge),

favWtSumOfOutgoingFollowEdges = Some(bir.totalOutgoingVolumes.favWtIfFollowEdge),

favWtSumOfOutgoingFavEdges = Some(bir.totalOutgoingVolumes.favWtIfFavEdge),

interestedInSize = Some(bir.interestedInSize),

sampledEdges = Some(

bir.edgeSample

.iterator()

.asScala

.toSeq

.map {

case (edge, data) => makeThriftSampledEdge(edge, data)

}

)

)

)

}

}

def getBIResultsFromSingleUser(

userId: Long,

neighbors: Seq[NeighborWithWeights],

clusters: ClustersUserIsInterestedIn,

useLogFavScores: Boolean

): List[(Int, BipartiteIntermediateResults)] = {

val neighborsToWeights = neighbors.map { neighborAndWeights =>

val isFollowEdge = neighborAndWeights.isFollowed match {

case Some(true) => 1.0

case \_ => 0.0

}

val favScore = if (useLogFavScores) {

neighborAndWeights.logFavScore.getOrElse(0.0)

} else neighborAndWeights.favScoreHalfLife100Days.getOrElse(0.0)

val isFavEdge = math.min(1, math.ceil(favScore))

neighborAndWeights.neighborId -> Weights(

isFollowEdge,

isFavEdge,

favScore \* isFollowEdge,

favScore

)

}.toMap

val outgoingVolumes = Monoid.sum(neighborsToWeights.values)(WeightsMonoid)

clusters.clusterIdToScores.toList.map {

case (clusterId, scoresStruct) =>

val inClusterNeighbors =

(scoresStruct.usersBeingFollowed.getOrElse(Nil) ++

scoresStruct.usersThatWereFaved.getOrElse(Nil)).toSet

val edgesForSampling = inClusterNeighbors.flatMap { neighborId =>

if (neighborsToWeights.contains(neighborId)) {

Some(

(userId, neighborId),

SampledEdgeData(

neighborsToWeights(neighborId).favWtIfFollowEdge,

neighborsToWeights(neighborId).favWtIfFavEdge,

scoresStruct.followScore.getOrElse(0.0),

scoresStruct.favScore.getOrElse(0.0)

)

)

} else {

None

}

}

val inClusterWeights =

Monoid.sum(neighborsToWeights.filterKeys(inClusterNeighbors).values)(WeightsMonoid)

(

clusterId,

BipartiteIntermediateResults(

inClusterWeights,

outgoingVolumes,

1,

samplerMonoid.build(edgesForSampling)

))

}

}

def computeClusterIncomingVolumes(

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

producerNormsAndCounts: TypedPipe[NormsAndCounts],

useLogFavWeights: Boolean

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

producerNormsAndCounts

.map { x => (x.userId, x) }

.join(knownFor)

.withReducers(100)

.flatMap {

case (userId, (normsAndCounts, clusters)) =>

clusters.map {

case (clusterId, \_) =>

val followerCount =

normsAndCounts.followerCount.getOrElse(0L).toDouble

val faverCount = normsAndCounts.faverCount.getOrElse(0L).toDouble

val favWtSumOfIncomingFollows = if (useLogFavWeights) {

normsAndCounts.logFavWeightsOnFollowEdgesSum.getOrElse(0.0)

} else {

normsAndCounts.favWeightsOnFollowEdgesSum.getOrElse(0.0)

}

val favWtSumOfIncomingFavs = if (useLogFavWeights) {

normsAndCounts.logFavWeightsOnFavEdgesSum.getOrElse(0.0)

} else {

normsAndCounts.favWeightsOnFavEdgesSum.getOrElse(0.0)

}

(

clusterId,

BipartiteClusterQuality(

incomingFollowEdges = Some(followerCount),

incomingFavEdges = Some(faverCount),

favWtSumOfIncomingFollowEdges = Some(favWtSumOfIncomingFollows),

favWtSumOfIncomingFavEdges = Some(favWtSumOfIncomingFavs)

))

}

}

.sumByKey

.toTypedPipe

}

def loadOrMake(

pipe: TypedPipe[(Int, BipartiteClusterQuality)],

modelVersion: String,

path: String

): Execution[TypedPipe[(Int, BipartiteClusterQuality)]] = {

val mapped = pipe.map {

case (clusterId, struct) => ((modelVersion, clusterId), struct)

}

makeForKeyValSource(mapped, AdhocKeyValSources.bipartiteQualitySource(path), path).map { pipe =>

// discard model version

pipe.map { case ((\_, clusterId), struct) => (clusterId, struct) }

}

}

def makeForKeyValSource[K, V](

pipe: TypedPipe[(K, V)],

dest: VersionedKeyValSource[K, V],

path: String

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

Execution.getMode.flatMap { mode =>

if (dest.resourceExists(mode)) {

println(s"validated path $path")

Execution.from(TypedPipe.from(dest))

} else {

println(s"Could not load from $path")

pipe.writeThrough(dest)

}

}

def precisionOfWholeGraph(

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

interestedIn: TypedPipe[(Long, ClustersUserIsInterestedIn)],

clusterIncomingVolumesExec: Execution[TypedPipe[(Int, BipartiteClusterQuality)]]

): Execution[Option[Double]] = {

val knownForSizeExec = knownFor.aggregate(Aggregator.size).toOptionExecution

val interestedInSizeExec =

interestedIn.aggregate(Aggregator.size).toOptionExecution

val numExec = clusterIncomingVolumesExec.flatMap { volumes =>

volumes.values.flatMap(\_.favWtSumOfIncomingFavEdges).sum.toOptionExecution

}

Execution.zip(numExec, interestedInSizeExec, knownForSizeExec).map {

case (Some(num), Some(interestedInSize), Some(knownForSize)) =>

Some(num / interestedInSize / knownForSize)

case x @ \_ =>

println("Precision of whole graph zip: " + x)

None

}

}

def finalPerClusterResults(

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

interestedIn: TypedPipe[(Long, ClustersUserIsInterestedIn)],

resultsWithOutgoingVolumesExec: Execution[TypedPipe[(Int, BipartiteClusterQuality)]],

incomingVolumesExec: Execution[TypedPipe[(Int, BipartiteClusterQuality)]]

): Execution[TypedPipe[(Int, BipartiteClusterQuality)]] = {

val knownForTranspose = KnownForSources.transpose(knownFor)

val precisionOfWholeGraphExec =

precisionOfWholeGraph(knownFor, interestedIn, incomingVolumesExec)

Execution

.zip(resultsWithOutgoingVolumesExec, incomingVolumesExec, precisionOfWholeGraphExec)

.map {

case (resultsWithOutgoingVolumes, clusterIncomingVolumes, precisionOfWholeGraph) =>

println("Precision of whole graph " + precisionOfWholeGraph)

resultsWithOutgoingVolumes

.join(knownForTranspose)

.leftJoin(clusterIncomingVolumes)

.withReducers(500)

.map {

case (clusterId, ((outgoingVolumeQuality, knownForList), incomingVolumesOpt)) =>

val incomingVolumes =

incomingVolumesOpt.getOrElse(BipartiteClusterQuality())

val knownForMap = knownForList.toMap

(

clusterId,

getFullQuality(

outgoingVolumeQuality,

incomingVolumes,

knownForMap,

precisionOfWholeGraph))

}

}

}

def getFullQuality(

qualityWithOutgoingVolumes: BipartiteClusterQuality,

incomingVolumes: BipartiteClusterQuality,

knownFor: Map[Long, Float],

precisionOfWholeGraph: Option[Double]

): BipartiteClusterQuality = {

val newSampledEdges = qualityWithOutgoingVolumes.sampledEdges.map { sampledEdges =>

sampledEdges.map { sampledEdge =>

val knownForScore = knownFor.getOrElse(sampledEdge.followeeId, 0.0f)

sampledEdge.copy(

predictedFollowScore = sampledEdge.followScoreToCluster.map { x => x \* knownForScore },

predictedFavScore = sampledEdge.favScoreToCluster.map { x => x \* knownForScore }

)

}

}

val correlationOfFavWtIfFollow = newSampledEdges.map { samples =>

val pairs = samples.map { s =>

(s.predictedFollowScore.getOrElse(0.0), s.favWtIfFollowEdge.getOrElse(0.0))

}

Util.computeCorrelation(pairs.iterator)

}

val correlationOfFavWtIfFav = newSampledEdges.map { samples =>

val pairs = samples.map { s =>

(s.predictedFavScore.getOrElse(0.0), s.favWtIfFavEdge.getOrElse(0.0))

}

Util.computeCorrelation(pairs.iterator)

}

val relativePrecisionNum = {

if (qualityWithOutgoingVolumes.interestedInSize.exists(\_ > 0) && knownFor.nonEmpty) {

qualityWithOutgoingVolumes.favWtSumOfInClusterFavEdges

.getOrElse(0.0) / qualityWithOutgoingVolumes.interestedInSize.get / knownFor.size

} else 0.0

}

val relativePrecision = if (precisionOfWholeGraph.exists(\_ > 0.0)) {

Some(relativePrecisionNum / precisionOfWholeGraph.get)

} else None

qualityWithOutgoingVolumes.copy(

incomingFollowEdges = incomingVolumes.incomingFollowEdges,

incomingFavEdges = incomingVolumes.incomingFavEdges,

favWtSumOfIncomingFollowEdges = incomingVolumes.favWtSumOfIncomingFollowEdges,

favWtSumOfIncomingFavEdges = incomingVolumes.favWtSumOfIncomingFavEdges,

knownForSize = Some(knownFor.size),

correlationOfFavWtIfFollowWithPredictedFollow = correlationOfFavWtIfFollow,

correlationOfFavWtIfFavWithPredictedFav = correlationOfFavWtIfFav,

sampledEdges = newSampledEdges,

relativePrecisionUsingFavWtIfFav = relativePrecision,

averagePrecisionOfWholeGraphUsingFavWtIfFav = precisionOfWholeGraph

)

}

}

object DumpBpQuality extends TwitterExecutionApp {

def job: Execution[Unit] = Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

val inputDir = args("inputDir")

val clusters = args.list("clusters").map(\_.toInt).toSet

val input =

TypedPipe

.from(AdhocKeyValSources.bipartiteQualitySource(inputDir))

.map {

case ((modelVersion, clusterId), quality) =>

(

(modelVersion, clusterId),

BipartiteClusterEvaluationClasses

.printableBipartiteQuality(quality))

}

if (clusters.isEmpty) {

input.printSummary("Bipartite quality")

} else {

input

.collect {

case rec @ ((\_, clusterId), quality) if clusters(clusterId) =>

Util.prettyJsonMapper

.writeValueAsString(rec)

.replaceAll("\n", " ")

}

.toIterableExecution

.map { strings => println(strings.mkString("\n")) }

}

}

}

}