package com.twitter.simclusters\_v2.scalding

import com.twitter.algebird.Semigroup

import com.twitter.bijection.Injection

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

import com.twitter.scalding.TypedPipe

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.ClusterId

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

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

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

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

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

/\*\*

\* This file implements the job for computing users' interestedIn vector from KnownFor data set.

\*

\* It reads the UserUserNormalizedGraphScalaDataset to get user-user follow + fav graph, and then

\* based on the known-for clusters of each followed/faved user, we calculate how much a user is

\* interestedIn a cluster.

\*/

/\*\*

\* Production job for computing interestedIn data set for the model version 20M145K2020.

\*

\* To deploy the job:

\*

\* capesospy-v2 update --build\_locally --start\_cron interested\_in\_for\_20M\_145k\_2020 \

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

\*/

object InterestedInFromKnownFor20M145K2020 extends InterestedInFromKnownForBatchBase {

override val firstTime: String = "2020-10-06"

override val outputKVDataset: KeyValDALDataset[KeyVal[Long, ClustersUserIsInterestedIn]] =

SimclustersV2RawInterestedIn20M145K2020ScalaDataset

override val outputPath: String = InternalDataPaths.RawInterestedIn2020Path

override val knownForModelVersion: String = ModelVersions.Model20M145K2020

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

SimclustersV2KnownFor20M145K2020ScalaDataset

}

/\*\*

\* base class for the main logic of computing interestedIn from KnownFor data set.

\*/

trait InterestedInFromKnownForBatchBase extends TwitterScheduledExecutionApp {

implicit val tz = DateOps.UTC

implicit val parser = DateParser.default

def firstTime: String

val batchIncrement: Duration = Days(7)

val lookBackDays: Duration = Days(30)

def outputKVDataset: KeyValDALDataset[KeyVal[Long, ClustersUserIsInterestedIn]]

def outputPath: String

def knownForModelVersion: String

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

private lazy val execArgs = AnalyticsBatchExecutionArgs(

batchDesc = BatchDescription(this.getClass.getName.replace("$", "")),

firstTime = BatchFirstTime(RichDate(firstTime)),

lastTime = None,

batchIncrement = BatchIncrement(batchIncrement)

)

override def scheduledJob: Execution[Unit] = AnalyticsBatchExecution(execArgs) {

implicit dateRange =>

Execution.withId { implicit uniqueId =>

Execution.withArgs { args =>

val normalizedGraph =

DAL.readMostRecentSnapshot(UserUserNormalizedGraphScalaDataset).toTypedPipe

val knownFor = KnownForSources.fromKeyVal(

DAL.readMostRecentSnapshot(knownForDALDataset, dateRange.extend(Days(30))).toTypedPipe,

knownForModelVersion

)

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

val maxClustersPerUser = args.int("maxClustersPerUser", 50)

val result = InterestedInFromKnownFor

.run(

normalizedGraph,

knownFor,

socialProofThreshold,

maxClustersPerUser,

knownForModelVersion

)

val writeKeyValResultExec = result

.map { case (userId, clusters) => KeyVal(userId, clusters) }

.writeDALVersionedKeyValExecution(

outputKVDataset,

D.Suffix(outputPath)

)

// read previous data set for validation purpose

val previousDataset = if (RichDate(firstTime).timestamp != dateRange.start.timestamp) {

DAL

.readMostRecentSnapshot(outputKVDataset, dateRange.prepend(lookBackDays)).toTypedPipe

.map {

case KeyVal(user, interestedIn) =>

(user, interestedIn)

}

} else {

TypedPipe.empty

}

Util.printCounters(

Execution

.zip(

writeKeyValResultExec,

InterestedInFromKnownFor.dataSetStats(result, "NewResult"),

InterestedInFromKnownFor.dataSetStats(previousDataset, "OldResult")

).unit

)

}

}

}

}

/\*\*

\* Adhoc job to compute user interestedIn.

\*

\* scalding remote run --target src/scala/com/twitter/simclusters\_v2/scalding:interested\_in\_adhoc \

\* --user recos-platform \

\* --submitter hadoopnest2.atla.twitter.com \

\* --main-class com.twitter.simclusters\_v2.scalding.InterestedInFromKnownForAdhoc -- \

\* --date 2019-08-26 --outputDir /user/recos-platform/adhoc/simclusters\_interested\_in\_log\_fav

\*/

object InterestedInFromKnownForAdhoc extends TwitterExecutionApp {

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

val normalizedGraph = TypedPipe.from(

UserAndNeighborsFixedPathSource(args("graphInputDir"))

)

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

val maxClustersPerUser = args.int("maxClustersPerUser", 20)

val knownForModelVersion = args("knownForModelVersion")

val knownFor = KnownForSources.readKnownFor(args("knownForInputDir"))

val outputSink = AdhocKeyValSources.interestedInSource(args("outputDir"))

Util.printCounters(

InterestedInFromKnownFor

.run(

normalizedGraph,

knownFor,

socialProofThreshold,

maxClustersPerUser,

knownForModelVersion

).writeExecution(outputSink)

)

}

}

}

/\*\*

\* Adhoc job to check the output of an adhoc interestedInSource.

\*/

object DumpInterestedInAdhoc extends TwitterExecutionApp {

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

val users = args.list("users").map(\_.toLong).toSet

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

input.filter { case (userId, rec) => users.contains(userId) }.toIterableExecution.map {

s => println(s.map(Util.prettyJsonMapper.writeValueAsString).mkString("\n"))

}

}

}

}

/\*\*

\* Helper functions

\*/

object InterestedInFromKnownFor {

private def ifNanMake0(x: Double): Double = if (x.isNaN) 0.0 else x

case class SrcClusterIntermediateInfo(

followScore: Double,

followScoreProducerNormalized: Double,

favScore: Double,

favScoreProducerNormalized: Double,

logFavScore: Double,

logFavScoreProducerNormalized: Double,

followSocialProof: List[Long],

favSocialProof: List[Long]) {

// overriding for the sake of unit tests

override def equals(obj: scala.Any): Boolean = {

obj match {

case that: SrcClusterIntermediateInfo =>

math.abs(followScore - that.followScore) < 1e-5 &&

math.abs(followScoreProducerNormalized - that.followScoreProducerNormalized) < 1e-5 &&

math.abs(favScore - that.favScore) < 1e-5 &&

math.abs(favScoreProducerNormalized - that.favScoreProducerNormalized) < 1e-5 &&

math.abs(logFavScore - that.logFavScore) < 1e-5 &&

math.abs(logFavScoreProducerNormalized - that.logFavScoreProducerNormalized) < 1e-5 &&

followSocialProof.toSet == that.followSocialProof.toSet &&

favSocialProof.toSet == that.favSocialProof.toSet

case \_ => false

}

}

}

implicit object SrcClusterIntermediateInfoSemigroup

extends Semigroup[SrcClusterIntermediateInfo] {

override def plus(

left: SrcClusterIntermediateInfo,

right: SrcClusterIntermediateInfo

): SrcClusterIntermediateInfo = {

SrcClusterIntermediateInfo(

followScore = left.followScore + right.followScore,

followScoreProducerNormalized =

left.followScoreProducerNormalized + right.followScoreProducerNormalized,

favScore = left.favScore + right.favScore,

favScoreProducerNormalized =

left.favScoreProducerNormalized + right.favScoreProducerNormalized,

logFavScore = left.logFavScore + right.logFavScore,

logFavScoreProducerNormalized =

left.logFavScoreProducerNormalized + right.logFavScoreProducerNormalized,

followSocialProof =

Semigroup.plus(left.followSocialProof, right.followSocialProof).distinct,

favSocialProof = Semigroup.plus(left.favSocialProof, right.favSocialProof).distinct

)

}

}

/\*\*

\* @param adjacencyLists User-User follow/fav graph

\* @param knownFor KnownFor data set. Each user can be known for several clusters with certain

\* knownFor weights.

\* @param socialProofThreshold A user will only be interested in a cluster if they follow/fav at

\* least certain number of users known for this cluster.

\* @param uniqueId required for these Stat

\* @return

\*/

def userClusterPairsWithoutNormalization(

adjacencyLists: TypedPipe[UserAndNeighbors],

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

socialProofThreshold: Int

)(

implicit uniqueId: UniqueID

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

val edgesToUsersWithKnownFor = Stat("num\_edges\_to\_users\_with\_known\_for")

val srcDestClusterTriples = Stat("num\_src\_dest\_cluster\_triples")

val srcClusterPairsBeforeSocialProofThresholding =

Stat("num\_src\_cluster\_pairs\_before\_social\_proof\_thresholding")

val srcClusterPairsAfterSocialProofThresholding =

Stat("num\_src\_cluster\_pairs\_after\_social\_proof\_thresholding")

val edges = adjacencyLists.flatMap {

case UserAndNeighbors(srcId, neighborsWithWeights) =>

neighborsWithWeights.map { neighborWithWeights =>

(

neighborWithWeights.neighborId,

neighborWithWeights.copy(neighborId = srcId)

)

}

}

implicit val l2b: Long => Array[Byte] = Injection.long2BigEndian

edges

.sketch(4000)

.join(knownFor)

.flatMap {

case (destId, (srcWithWeights, clusterArray)) =>

edgesToUsersWithKnownFor.inc()

clusterArray.toList.map {

case (clusterId, knownForScoreF) =>

val knownForScore = math.max(0.0, knownForScoreF.toDouble)

srcDestClusterTriples.inc()

val followScore =

if (srcWithWeights.isFollowed.contains(true)) knownForScore else 0.0

val followScoreProducerNormalizedOnly =

srcWithWeights.followScoreNormalizedByNeighborFollowersL2.getOrElse(

0.0) \* knownForScore

val favScore =

srcWithWeights.favScoreHalfLife100Days.getOrElse(0.0) \* knownForScore

val favScoreProducerNormalizedOnly =

srcWithWeights.favScoreHalfLife100DaysNormalizedByNeighborFaversL2.getOrElse(

0.0) \* knownForScore

val logFavScore = srcWithWeights.logFavScore.getOrElse(0.0) \* knownForScore

val logFavScoreProducerNormalizedOnly = srcWithWeights.logFavScoreL2Normalized

.getOrElse(0.0) \* knownForScore

val followSocialProof = if (srcWithWeights.isFollowed.contains(true)) {

List(destId)

} else Nil

val favSocialProof = if (srcWithWeights.favScoreHalfLife100Days.exists(\_ > 0)) {

List(destId)

} else Nil

(

(srcWithWeights.neighborId, clusterId),

SrcClusterIntermediateInfo(

followScore,

followScoreProducerNormalizedOnly,

favScore,

favScoreProducerNormalizedOnly,

logFavScore,

logFavScoreProducerNormalizedOnly,

followSocialProof,

favSocialProof

)

)

}

}

.sumByKey

.withReducers(10000)

.filter {

case ((\_, \_), SrcClusterIntermediateInfo(\_, \_, \_, \_, \_, \_, followProof, favProof)) =>

srcClusterPairsBeforeSocialProofThresholding.inc()

val distinctSocialProof = (followProof ++ favProof).toSet

val result = distinctSocialProof.size >= socialProofThreshold

if (result) {

srcClusterPairsAfterSocialProofThresholding.inc()

}

result

}

}

/\*\*

\* Add the cluster-level l2 norm scores, and use them to normalize follow/fav scores.

\*/

def attachNormalizedScores(

intermediate: TypedPipe[((Long, Int), SrcClusterIntermediateInfo)]

)(

implicit uniqueId: UniqueID

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

def square(x: Double): Double = x \* x

val clusterCountsAndNorms =

intermediate

.map {

case (

(\_, clusterId),

SrcClusterIntermediateInfo(

followScore,

followScoreProducerNormalizedOnly,

favScore,

favScoreProducerNormalizedOnly,

logFavScore,

logFavScoreProducerNormalizedOnly,

\_,

\_

)

) =>

(

clusterId,

(

1,

square(followScore),

square(followScoreProducerNormalizedOnly),

square(favScore),

square(favScoreProducerNormalizedOnly),

square(logFavScore),

square(logFavScoreProducerNormalizedOnly)

)

)

}

.sumByKey

// .withReducers(100)

.map {

case (

clusterId,

(

cnt,

squareFollowScore,

squareFollowScoreProducerNormalizedOnly,

squareFavScore,

squareFavScoreProducerNormalizedOnly,

squareLogFavScore,

squareLogFavScoreProducerNormalizedOnly

)) =>

(

clusterId,

(

cnt,

math.sqrt(squareFollowScore),

math.sqrt(squareFollowScoreProducerNormalizedOnly),

math.sqrt(squareFavScore),

math.sqrt(squareFavScoreProducerNormalizedOnly),

math.sqrt(squareLogFavScore),

math.sqrt(squareLogFavScoreProducerNormalizedOnly)

))

}

implicit val i2b: Int => Array[Byte] = Injection.int2BigEndian

intermediate

.map {

case ((srcId, clusterId), clusterScoresTuple) =>

(clusterId, (srcId, clusterScoresTuple))

}

.sketch(reducers = 900)

.join(clusterCountsAndNorms)

.map {

case (

clusterId,

(

(

srcId,

SrcClusterIntermediateInfo(

followScore,

followScoreProducerNormalizedOnly,

favScore,

favScoreProducerNormalizedOnly,

logFavScore,

logFavScoreProducerNormalizedOnly, // not used for now

followProof,

favProof

)

),

(

cnt,

followNorm,

followProducerNormalizedNorm,

favNorm,

favProducerNormalizedNorm,

logFavNorm,

logFavProducerNormalizedNorm // not used for now

)

)

) =>

(

srcId,

List(

(

clusterId,

UserToInterestedInClusterScores(

followScore = Some(ifNanMake0(followScore)),

followScoreClusterNormalizedOnly = Some(ifNanMake0(followScore / followNorm)),

followScoreProducerNormalizedOnly =

Some(ifNanMake0(followScoreProducerNormalizedOnly)),

followScoreClusterAndProducerNormalized = Some(

ifNanMake0(followScoreProducerNormalizedOnly / followProducerNormalizedNorm)),

favScore = Some(ifNanMake0(favScore)),

favScoreClusterNormalizedOnly = Some(ifNanMake0(favScore / favNorm)),

favScoreProducerNormalizedOnly = Some(ifNanMake0(favScoreProducerNormalizedOnly)),

favScoreClusterAndProducerNormalized =

Some(ifNanMake0(favScoreProducerNormalizedOnly / favProducerNormalizedNorm)),

usersBeingFollowed = Some(followProof),

usersThatWereFaved = Some(favProof),

numUsersInterestedInThisClusterUpperBound = Some(cnt),

logFavScore = Some(ifNanMake0(logFavScore)),

logFavScoreClusterNormalizedOnly = Some(ifNanMake0(logFavScore / logFavNorm))

))

)

)

}

.sumByKey

// .withReducers(1000)

.toTypedPipe

}

/\*\*

\* aggregate cluster scores for each user, to be used instead of attachNormalizedScores

\* when we donot want to compute cluster-level l2 norm scores

\*/

def groupClusterScores(

intermediate: TypedPipe[((Long, Int), SrcClusterIntermediateInfo)]

)(

implicit uniqueId: UniqueID

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

intermediate

.map {

case (

(srcId, clusterId),

SrcClusterIntermediateInfo(

followScore,

followScoreProducerNormalizedOnly,

favScore,

favScoreProducerNormalizedOnly,

logFavScore,

logFavScoreProducerNormalizedOnly,

followProof,

favProof

)

) =>

(

srcId,

List(

(

clusterId,

UserToInterestedInClusterScores(

followScore = Some(ifNanMake0(followScore)),

followScoreProducerNormalizedOnly =

Some(ifNanMake0(followScoreProducerNormalizedOnly)),

favScore = Some(ifNanMake0(favScore)),

favScoreProducerNormalizedOnly = Some(ifNanMake0(favScoreProducerNormalizedOnly)),

usersBeingFollowed = Some(followProof),

usersThatWereFaved = Some(favProof),

logFavScore = Some(ifNanMake0(logFavScore)),

))

)

)

}

.sumByKey

.withReducers(1000)

.toTypedPipe

}

/\*\*

\* For each user, only keep up to a certain number of clusters.

\* @param allInterests user with a list of interestedIn clusters.

\* @param maxClustersPerUser number of clusters to keep for each user

\* @param knownForModelVersion known for model version

\* @param uniqueId required for these Stat

\* @return

\*/

def keepOnlyTopClusters(

allInterests: TypedPipe[(Long, List[(Int, UserToInterestedInClusterScores)])],

maxClustersPerUser: Int,

knownForModelVersion: String

)(

implicit uniqueId: UniqueID

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

val userClusterPairsBeforeUserTruncation =

Stat("num\_user\_cluster\_pairs\_before\_user\_truncation")

val userClusterPairsAfterUserTruncation =

Stat("num\_user\_cluster\_pairs\_after\_user\_truncation")

val usersWithALotOfClusters =

Stat(s"num\_users\_with\_more\_than\_${maxClustersPerUser}\_clusters")

allInterests

.map {

case (srcId, fullClusterList) =>

userClusterPairsBeforeUserTruncation.incBy(fullClusterList.size)

val truncatedClusters = if (fullClusterList.size > maxClustersPerUser) {

usersWithALotOfClusters.inc()

fullClusterList

.sortBy {

case (\_, clusterScores) =>

(

-clusterScores.favScore.getOrElse(0.0),

-clusterScores.logFavScore.getOrElse(0.0),

-clusterScores.followScore.getOrElse(0.0),

-clusterScores.logFavScoreClusterNormalizedOnly.getOrElse(0.0),

-clusterScores.followScoreProducerNormalizedOnly.getOrElse(0.0)

)

}

.take(maxClustersPerUser)

} else {

fullClusterList

}

userClusterPairsAfterUserTruncation.incBy(truncatedClusters.size)

(srcId, ClustersUserIsInterestedIn(knownForModelVersion, truncatedClusters.toMap))

}

}

def run(

adjacencyLists: TypedPipe[UserAndNeighbors],

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

socialProofThreshold: Int,

maxClustersPerUser: Int,

knownForModelVersion: String

)(

implicit uniqueId: UniqueID

): TypedPipe[(UserId, ClustersUserIsInterestedIn)] = {

keepOnlyTopClusters(

attachNormalizedScores(

userClusterPairsWithoutNormalization(

adjacencyLists,

knownFor,

socialProofThreshold

)

),

maxClustersPerUser,

knownForModelVersion

)

}

/\*\*

\* run the interestedIn job, cluster normalized scores are not attached to user's clusters.

\*/

def runWithoutClusterNormalizedScores(

adjacencyLists: TypedPipe[UserAndNeighbors],

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

socialProofThreshold: Int,

maxClustersPerUser: Int,

knownForModelVersion: String

)(

implicit uniqueId: UniqueID

): TypedPipe[(UserId, ClustersUserIsInterestedIn)] = {

keepOnlyTopClusters(

groupClusterScores(

userClusterPairsWithoutNormalization(

adjacencyLists,

knownFor,

socialProofThreshold

)

),

maxClustersPerUser,

knownForModelVersion

)

}

/\*\*

\* print out some basic stats of the data set to make sure things are not broken

\*/

def dataSetStats(

interestedInData: TypedPipe[(UserId, ClustersUserIsInterestedIn)],

dataSetName: String = ""

): Execution[Unit] = {

Execution

.zip(

Util.printSummaryOfNumericColumn(

interestedInData.map {

case (user, interestedIn) =>

interestedIn.clusterIdToScores.size

},

Some(s"$dataSetName UserInterestedIn Size")

),

Util.printSummaryOfNumericColumn(

interestedInData.flatMap {

case (user, interestedIn) =>

interestedIn.clusterIdToScores.map {

case (\_, scores) =>

scores.favScore.getOrElse(0.0)

}

},

Some(s"$dataSetName UserInterestedIn favScore")

),

Util.printSummaryOfNumericColumn(

interestedInData.flatMap {

case (user, interestedIn) =>

interestedIn.clusterIdToScores.map {

case (\_, scores) =>

scores.favScoreClusterNormalizedOnly.getOrElse(0.0)

}

},

Some(s"$dataSetName UserInterestedIn favScoreClusterNormalizedOnly")

),

Util.printSummaryOfNumericColumn(

interestedInData.flatMap {

case (user, interestedIn) =>

interestedIn.clusterIdToScores.map {

case (\_, scores) =>

scores.logFavScoreClusterNormalizedOnly.getOrElse(0.0)

}

},

Some(s"$dataSetName UserInterestedIn logFavScoreClusterNormalizedOnly")

)

).unit

}

}