package com.twitter.simclusters\_v2.scalding.inferred\_entities

import com.twitter.algebird.Max

import com.twitter.scalding.Args

import com.twitter.scalding.DateRange

import com.twitter.scalding.Days

import com.twitter.scalding.Duration

import com.twitter.scalding.Execution

import com.twitter.scalding.RichDate

import com.twitter.scalding.TypedPipe

import com.twitter.scalding.TypedTsv

import com.twitter.scalding.UniqueID

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

import com.twitter.wtf.scalding.jobs.common.AdhocExecutionApp

import com.twitter.wtf.scalding.jobs.common.ScheduledExecutionApp

import java.util.TimeZone

import com.twitter.onboarding.relevance.source.UttAccountRecommendationsScalaDataset

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

import com.twitter.wtf.entity\_real\_graph.scalding.common.SemanticCoreFilters.getValidSemanticCoreEntities

import com.twitter.wtf.entity\_real\_graph.scalding.common.DataSources

/\*\*

\* Infer interested-in entities for a given user. Depending on how and where the entity source comes

\* from, this can be achieve a number of ways. For example, we can use user->interested-in clusters

\* and cluster-> semanticcore entity embeddings to derive user->entity. Or, we can use a producers'

\* UTT embeddings and user-user engagement graph to aggregate UTT engagement history.

\*/

object InferredEntitiesFromInterestedIn {

def getUserToKnownForUttEntities(

dateRange: DateRange,

maxUttEntitiesPerUser: Int

)(

implicit timeZone: TimeZone

): TypedPipe[(UserId, Seq[(Long, Double)])] = {

val validEntities = getValidSemanticCoreEntities(

DataSources.semanticCoreMetadataSource(dateRange, timeZone)).distinct.map { entityId =>

Set(entityId)

}.sum

DAL

.readMostRecentSnapshot(UttAccountRecommendationsScalaDataset, dateRange)

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

.flatMapWithValue(validEntities) {

// Keep only valid Entities

case (KeyVal(interest, candidates), Some(validUTTEntities))

if validUTTEntities.contains(interest.uttID) =>

candidates.recommendations.map { rec =>

(rec.candidateUserID, (interest.uttID, rec.score.getOrElse(0.0)))

}

case \_ => None

}

.group

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

.toTypedPipe

}

def filterUTTEntities(

interestedInEntities: TypedPipe[(UserId, Seq[(UTTEntityId, Int)])],

minSocialProofThreshold: Int,

maxInterestsPerUser: Int

): TypedPipe[(UserId, Seq[UTTEntityId])] = {

interestedInEntities

.map {

case (userId, entities) =>

val topEntities = entities

.filter(\_.\_2 >= minSocialProofThreshold)

.sortBy(-\_.\_2)

.take(maxInterestsPerUser)

.map(\_.\_1)

(userId, topEntities)

}

.filter(\_.\_2.nonEmpty)

}

def getUserToUTTEntities(

userUserGraph: TypedPipe[UserAndNeighbors],

knownForEntities: TypedPipe[(UserId, Seq[UTTEntityId])]

)(

implicit uniqueId: UniqueID

): TypedPipe[(UserId, Seq[(UTTEntityId, Int)])] = {

val flatEngagementGraph =

userUserGraph

.count("num\_user\_user\_graph\_records")

.flatMap { userAndNeighbors =>

userAndNeighbors.neighbors.flatMap { neighbor =>

val producerId = neighbor.neighborId

val hasFav = neighbor.favScoreHalfLife100Days.exists(\_ > 0)

val hasFollow = neighbor.isFollowed.contains(true)

if (hasFav || hasFollow) {

Some((producerId, userAndNeighbors.userId))

} else {

None

}

}

}

.count("num\_flat\_user\_user\_graph\_edges")

flatEngagementGraph

.join(knownForEntities.count("num\_producer\_to\_entities"))

.withReducers(3000)

.flatMap {

case (producerId, (userId, entities)) =>

entities.map { entityId => ((userId, entityId), 1) }

}

.count("num\_flat\_user\_to\_entity")

.sumByKey

.withReducers(2999)

.toTypedPipe

.count("num\_user\_with\_entities")

.collect {

case ((userId, uttEntityId), numEngagements) =>

(userId, Seq((uttEntityId, numEngagements)))

}

.sumByKey

}

/\*\*

\* Infer entities using user-interestedIn clusters and entity embeddings for those clusters,

\* based on a threshold

\*/

def getInterestedInFromEntityEmbeddings(

userToInterestedIn: TypedPipe[(UserId, ClustersUserIsInterestedIn)],

clusterToEntities: TypedPipe[(ClusterId, Seq[SemanticCoreEntityWithScore])],

inferredFromCluster: Option[SimClustersSource],

inferredFromEntity: Option[EntitySource]

)(

implicit uniqueId: UniqueID

): TypedPipe[(UserId, Seq[InferredEntity])] = {

val clusterToUsers = userToInterestedIn

.flatMap {

case (userId, clusters) =>

clusters.clusterIdToScores.map {

case (clusterId, score) =>

(clusterId, (userId, score))

}

}

.count("num\_flat\_user\_to\_interested\_in\_cluster")

clusterToUsers

.join(clusterToEntities)

.withReducers(3000)

.map {

case (clusterId, ((userId, interestedInScore), entitiesWithScores)) =>

(userId, entitiesWithScores)

}

.flatMap {

case (userId, entitiesWithScore) =>

// Dedup by entityIds in case user is associated with an entity from different clusters

entitiesWithScore.map { entity => (userId, Map(entity.entityId -> Max(entity.score))) }

}

.sumByKey

.map {

case (userId, entitiesWithMaxScore) =>

val inferredEntities = entitiesWithMaxScore.map { entityWithScore =>

InferredEntity(

entityId = entityWithScore.\_1,

score = entityWithScore.\_2.get,

simclusterSource = inferredFromCluster,

entitySource = inferredFromEntity

)

}.toSeq

(userId, inferredEntities)

}

.count("num\_user\_with\_inferred\_entities")

}

}

/\*\*

capesospy-v2 update --build\_locally --start\_cron \

--start\_cron inferred\_entities\_from\_interested\_in \

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

\*/

object InferredInterestedInSemanticCoreEntitiesBatchApp extends ScheduledExecutionApp {

override def firstTime: RichDate = RichDate("2023-01-01")

override def batchIncrement: Duration = Days(1)

private val outputPath = InferredEntities.MHRootPath + "/interested\_in"

private val outputPathKeyedByCluster =

InferredEntities.MHRootPath + "/interested\_in\_keyed\_by\_cluster"

import InferredEntitiesFromInterestedIn.\_

override def runOnDateRange(

args: Args

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

Execution.unit

val clusterToEntities = InferredEntities

.getLegibleEntityEmbeddings(dateRange, timeZone)

.count("num\_legible\_cluster\_to\_entities")

.forceToDisk

// inferred interests. Only support 2020 model version

val userToClusters2020 =

InterestedInSources.simClustersInterestedIn2020Source(dateRange, timeZone)

val inferredEntities2020 = getInterestedInFromEntityEmbeddings(

userToInterestedIn = userToClusters2020,

clusterToEntities = clusterToEntities,

inferredFromCluster = Some(InferredEntities.InterestedIn2020),

inferredFromEntity = Some(EntitySource.SimClusters20M145K2020EntityEmbeddingsByFavScore)

)(uniqueID)

.count("num\_user\_with\_inferred\_entities\_2020")

val combinedInferredInterests =

InferredEntities.combineResults(inferredEntities2020)

// output cluster -> entity mapping

val clusterToEntityExec = clusterToEntities

.map {

case (clusterId, entities) =>

val inferredEntities = SimClustersInferredEntities(

entities.map(entity => InferredEntity(entity.entityId, entity.score))

)

KeyVal(clusterId, inferredEntities)

}

.writeDALVersionedKeyValExecution(

SimclustersInferredEntitiesFromInterestedInKeyedByClusterScalaDataset,

D.Suffix(outputPathKeyedByCluster)

)

// output user -> entity mapping

val userToEntityExec = combinedInferredInterests

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

.writeDALVersionedKeyValExecution(

SimclustersInferredEntitiesFromInterestedInScalaDataset,

D.Suffix(outputPath)

)

Execution.zip(clusterToEntityExec, userToEntityExec).unit

}

}

/\*\*

Adhob debugging job. Uses Entity Embeddings dataset to infer user interests

./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding/inferred\_entities/ &&\

scalding remote run \

--main-class com.twitter.simclusters\_v2.scalding.inferred\_entities.InferredInterestedInSemanticCoreEntitiesAdhocApp \

--target src/scala/com/twitter/simclusters\_v2/scalding/inferred\_entities:inferred\_entities\_from\_interested\_in-adhoc \

--user recos-platform \

-- --date 2019-11-11 --email your\_ldap@twitter.com

\*/

object InferredInterestedInSemanticCoreEntitiesAdhocApp extends AdhocExecutionApp {

import InferredEntitiesFromInterestedIn.\_

override def runOnDateRange(

args: Args

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

val interestedIn = InterestedInSources.simClustersInterestedIn2020Source(dateRange, timeZone)

val clusterToEntities = InferredEntities

.getLegibleEntityEmbeddings(dateRange, timeZone)

.count("num\_legible\_cluster\_to\_entities")

// Debugging InterestedIn -> EntityEmbeddings approach

val interestedInFromEntityEmbeddings = getInterestedInFromEntityEmbeddings(

interestedIn,

clusterToEntities,

None,

None

)(uniqueID)

val distribution = Util

.printSummaryOfNumericColumn(

interestedInFromEntityEmbeddings.map { case (k, v) => v.size },

Some("# of interestedIn entities per user")

).map { results =>

Util.sendEmail(results, "# of interestedIn entities per user", args.getOrElse("email", ""))

}

Execution

.zip(

distribution,

interestedInFromEntityEmbeddings

.writeExecution(

TypedTsv("/user/recos-platform/adhoc/debug/interested\_in\_from\_entity\_embeddings"))

).unit

}

}

/\*\*

Adhob debuggingjob. Runs through the UTT interest inference, analyze the size & distribution of

interests per user.

./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding/inferred\_entities/ &&\

scalding remote run \

--main-class com.twitter.simclusters\_v2.scalding.inferred\_entities.InferredUTTEntitiesFromInterestedInAdhocApp \

--target src/scala/com/twitter/simclusters\_v2/scalding/inferred\_entities:inferred\_entities\_from\_interested\_in-adhoc \

--user recos-platform \

-- --date 2019-11-03 --email your\_ldap@twitter.com

\*/

object InferredUTTEntitiesFromInterestedInAdhocApp extends AdhocExecutionApp {

import InferredEntitiesFromInterestedIn.\_

override def runOnDateRange(

args: Args

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

val employeeGraphPath = "/user/recos-platform/adhoc/employee\_graph\_from\_user\_user/"

val employeeGraph = TypedPipe.from(UserAndNeighborsFixedPathSource(employeeGraphPath))

val maxKnownForUttsPerProducer = 100

val minSocialProofThreshold = 10

val maxInferredInterestsPerUser = 500

// KnownFor UTT entities

val userToUttEntities = getUserToKnownForUttEntities(

dateRange.embiggen(Days(7)),

maxKnownForUttsPerProducer

).map { case (userId, entities) => (userId, entities.map(\_.\_1)) }

val userToInterestsEngagementCounts = getUserToUTTEntities(employeeGraph, userToUttEntities)

val topInterests = filterUTTEntities(

userToInterestsEngagementCounts,

minSocialProofThreshold,

maxInferredInterestsPerUser

).count("num\_users\_with\_inferred\_interests")

// Debugging UTT entities

val analysis = Util

.printSummaryOfNumericColumn(

topInterests.map { case (k, v) => v.size },

Some(

"# of UTT entities per user, maxKnownForUtt=100, minSocialProof=10, maxInferredPerUser=500")

).map { results =>

Util.sendEmail(results, "# of UTT entities per user", args.getOrElse("email", ""))

}

val outputPath = "/user/recos-platform/adhoc/inferred\_utt\_interests"

Execution

.zip(

topInterests.writeExecution(TypedTsv(outputPath)),

analysis

).unit

}

}