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

import com.twitter.escherbird.metadata.thriftscala.FullMetadata

import com.twitter.scalding.\_

import com.twitter.scalding.typed.TypedPipe

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

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.\_

import com.twitter.wtf.entity\_real\_graph.scalding.common.{DataSources => ERGDataSources}

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

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

import java.util.TimeZone

/\*\*

\* Infer Known-For entities based on users' different variations of SimClusters Known-Fors.

\* The basic idea is to look at the Known-For datasets (User, Cluster) and the entity embeddings

\* (Cluster, Entities) to derive the (User, Entities).

\*/

object InferredSemanticCoreEntitiesFromKnownFor {

/\*\*

\* Given a (user, cluster) and (cluster, entity) mappings, generate (user, entity) mappings

\*/

def getUserToEntities(

userToClusters: TypedPipe[(UserId, Seq[SimClusterWithScore])],

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

inferredFromCluster: Option[SimClustersSource],

inferredFromEntity: Option[EntitySource],

minEntityScore: Double

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

val validClusterToEntities = clusterToEntities.flatMap {

case (clusterId, entities) =>

entities.collect {

case entity if entity.score >= minEntityScore =>

(clusterId, (entity.entityId, entity.score))

}

}

userToClusters

.flatMap {

case (userId, clusters) =>

clusters.map { cluster => (cluster.clusterId, userId) }

}

.join(validClusterToEntities)

.map {

case (clusterId, (userId, (entityId, score))) =>

((userId, entityId), score)

}

// If a user is known for the same entity through multiple cluster-entity mappings, sum the scores

.sumByKey

.map {

case ((userId, entityId), score) =>

(userId, Seq(InferredEntity(entityId, score, inferredFromCluster, inferredFromEntity)))

}

.sumByKey

}

}

/\*\*

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

inferred\_entities\_from\_known\_for \

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

\*/

object InferredKnownForSemanticCoreEntitiesBatchApp extends ScheduledExecutionApp {

import InferredSemanticCoreEntitiesFromKnownFor.\_

override def firstTime: RichDate = RichDate("2023-01-23")

override def batchIncrement: Duration = Days(1)

private val outputPath = InferredEntities.MHRootPath + "/known\_for"

override def runOnDateRange(

args: Args

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

val clusterToEntities = EntityEmbeddingsSources

.getReverseIndexedSemanticCoreEntityEmbeddingsSource(

EmbeddingType.FavBasedSematicCoreEntity,

ModelVersions.Model20M145K2020,

dateRange.embiggen(Days(7)) // read 7 days before & after to give buffer

)

.forceToDisk

val userToEntities2020 = getUserToEntities(

ProdSources.getUpdatedKnownFor,

clusterToEntities,

Some(InferredEntities.KnownFor2020),

Some(EntitySource.SimClusters20M145K2020EntityEmbeddingsByFavScore),

InferredEntities.MinLegibleEntityScore

)

val userToEntities = InferredEntities.combineResults(userToEntities2020)

userToEntities

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

.writeDALVersionedKeyValExecution(

SimclustersInferredEntitiesFromKnownForScalaDataset,

D.Suffix(outputPath)

)

}

}

/\*\*

./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding/inferred\_entities:inferred\_entities\_from\_known\_for-adhoc && \

oscar hdfs --user recos-platform --screen --tee your\_ldap-logs/ \

--bundle inferred\_entities\_from\_known\_for-adhoc \

--tool com.twitter.simclusters\_v2.scalding.inferred\_entities.InferredSemanticCoreEntitiesFromKnownForAdhocApp \

-- --date 2019-11-02 --email your\_ldap@twitter.com

\*/

object InferredSemanticCoreEntitiesFromKnownForAdhocApp extends AdhocExecutionApp {

private def readEntityEmbeddingsFromPath(

path: String

): TypedPipe[(ClusterId, Seq[SemanticCoreEntityWithScore])] = {

TypedPipe

.from(AdhocKeyValSources.clusterToEntitiesSource(path))

.map {

case (embeddingId, embedding) =>

embeddingId.internalId match {

case InternalId.ClusterId(clusterId) =>

val semanticCoreEntities = embedding.embedding.map {

case InternalIdWithScore(InternalId.EntityId(entityId), score) =>

SemanticCoreEntityWithScore(entityId, score)

case \_ =>

throw new IllegalArgumentException(

"The value to the entity embeddings dataset isn't entityId"

)

}

(clusterId, semanticCoreEntities)

case \_ =>

throw new IllegalArgumentException(

"The key to the entity embeddings dataset isn't clusterId"

)

}

}

}

override def runOnDateRange(

args: Args

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

import InferredSemanticCoreEntitiesFromKnownFor.\_

val entityIdToString: TypedPipe[(Long, String)] =

ERGDataSources.semanticCoreMetadataSource

.collect {

case FullMetadata(domainId, entityId, Some(basicMetadata), \_, \_, \_)

if domainId == 131L && !basicMetadata.indexableFields.exists(

\_.tags.exists(\_.contains("utt:sensitive\_interest"))) =>

entityId -> basicMetadata.name

}.distinctBy(\_.\_1)

val clusterToEntitiesUpdated = EntityEmbeddingsSources

.getReverseIndexedSemanticCoreEntityEmbeddingsSource(

EmbeddingType.FavBasedSematicCoreEntity,

ModelVersions.Model20M145KUpdated,

dateRange.embiggen(Days(4)) // read 4 days before & after to give buffer

)

.forceToDisk

// Inferred entities based on Updated version's entity embeddings

val dec11UserToUpdatedEntities = getUserToEntities(

ProdSources.getDec11KnownFor,

clusterToEntitiesUpdated,

Some(InferredEntities.Dec11KnownFor),

Some(EntitySource.SimClusters20M145KUpdatedEntityEmbeddingsByFavScore),

InferredEntities.MinLegibleEntityScore

)

val updatedUserToUpdatedEntities = getUserToEntities(

ProdSources.getUpdatedKnownFor,

clusterToEntitiesUpdated,

Some(InferredEntities.UpdatedKnownFor),

Some(EntitySource.SimClusters20M145KUpdatedEntityEmbeddingsByFavScore),

InferredEntities.MinLegibleEntityScore

)

// Updated entities data

val entitiesPipe = (

dec11UserToUpdatedEntities ++ updatedUserToUpdatedEntities

).sumByKey

val userToEntitiesWithString = entitiesPipe

.flatMap {

case (userId, entities) =>

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

}

.hashJoin(entityIdToString)

.map {

case (entityId, ((userId, inferredEntity), entityStr)) =>

(userId, Seq((entityStr, inferredEntity)))

}

.sumByKey

val outputPath = "/user/recos-platform/adhoc/known\_for\_inferred\_entities\_updated"

val scoreDistribution = Util

.printSummaryOfNumericColumn(

entitiesPipe.flatMap { case (k, v) => v.map(\_.score) },

Some("Distributions of scores, Updated version")

).map { results =>

Util.sendEmail(

results,

"Distributions of scores, Updated version",

args.getOrElse("email", "")

)

}

val coverageDistribution = Util

.printSummaryOfNumericColumn(

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

Some("# of knownFor entities per user, Updated version")

).map { results =>

Util.sendEmail(

results,

"# of knownFor entities per user, Updated version",

args.getOrElse("email", "")

)

}

Execution

.zip(

userToEntitiesWithString.writeExecution(TypedTsv(outputPath)),

scoreDistribution,

coverageDistribution

).unit

}

}