package com.twitter.simclusters\_v2.scalding.embedding.producer

import com.twitter.scalding.\_

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

import com.twitter.scalding\_internal.source.lzo\_scrooge.FixedPathLzoScrooge

import com.twitter.simclusters\_v2.hdfs\_sources.{DataSources, InterestedInSources}

import com.twitter.simclusters\_v2.scalding.common.matrix.{SparseMatrix, SparseRowMatrix}

import com.twitter.simclusters\_v2.scalding.embedding.ProducerEmbeddingsFromInterestedIn

import com.twitter.simclusters\_v2.scalding.embedding.common.EmbeddingUtil.{

ClusterId,

ProducerId,

UserId

}

import com.twitter.simclusters\_v2.scalding.embedding.common.SimClustersEmbeddingBaseJob

import com.twitter.simclusters\_v2.thriftscala.{EmbeddingType, \_}

import java.util.TimeZone

/\*\*

\* This file implements a new Producer SimClusters Embeddings.

\* The differences with existing producer embeddings are:

\*

\* 1) the embedding scores are not normalized, so that one can aggregate multiple producer embeddings by adding them.

\* 2) we use log-fav scores in the user-producer graph and user-simclusters graph.

\* LogFav scores are smoother than fav scores we previously used and they are less sensitive to outliers

\*

\*

\*

\* The main difference with other normalized embeddings is the `convertEmbeddingToAggregatableEmbeddings` function

\* where we multiply the normalized embedding with producer's norms. The resulted embeddings are then

\* unnormalized and aggregatable.

\*

\*/

trait AggregatableProducerEmbeddingsBaseApp extends SimClustersEmbeddingBaseJob[ProducerId] {

val userToProducerScoringFn: NeighborWithWeights => Double

val userToClusterScoringFn: UserToInterestedInClusterScores => Double

val modelVersion: ModelVersion

// Minimum engagement threshold

val minNumFavers: Int = ProducerEmbeddingsFromInterestedIn.minNumFaversForProducer

override def numClustersPerNoun: Int = 60

override def numNounsPerClusters: Int = 500 // this is not used for now

override def thresholdForEmbeddingScores: Double = 0.01

override def prepareNounToUserMatrix(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): SparseMatrix[ProducerId, UserId, Double] = {

SparseMatrix(

ProducerEmbeddingsFromInterestedIn

.getFilteredUserUserNormalizedGraph(

DataSources.userUserNormalizedGraphSource,

DataSources.userNormsAndCounts,

userToProducerScoringFn,

\_.faverCount.exists(

\_ > minNumFavers

)

)

.map {

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

(producerId, userId, score)

})

}

override def prepareUserToClusterMatrix(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): SparseRowMatrix[UserId, ClusterId, Double] = {

SparseRowMatrix(

ProducerEmbeddingsFromInterestedIn

.getUserSimClustersMatrix(

InterestedInSources

.simClustersInterestedInSource(modelVersion, dateRange.embiggen(Days(5)), timeZone),

userToClusterScoringFn,

modelVersion

)

.mapValues(\_.toMap),

isSkinnyMatrix = true

)

}

// in order to make the embeddings aggregatable, we need to revert the normalization

// (multiply the norms) we did when computing embeddings in the base job.

def convertEmbeddingToAggregatableEmbeddings(

embeddings: TypedPipe[(ProducerId, Seq[(ClusterId, Double)])]

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): TypedPipe[(ProducerId, Seq[(ClusterId, Double)])] = {

embeddings.join(prepareNounToUserMatrix.rowL2Norms).map {

case (producerId, (embeddingVec, norm)) =>

producerId -> embeddingVec.map {

case (id, score) => (id, score \* norm)

}

}

}

override final def writeClusterToNounsIndex(

output: TypedPipe[(ClusterId, Seq[(ProducerId, Double)])]

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = { Execution.unit } // we do not need this for now

/\*\*

\* Override this method to write the manhattan dataset.

\*/

def writeToManhattan(

output: TypedPipe[KeyVal[SimClustersEmbeddingId, SimClustersEmbedding]]

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit]

/\*\*

\* Override this method to writethrough the thrift dataset.

\*/

def writeToThrift(

output: TypedPipe[SimClustersEmbeddingWithId]

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit]

val embeddingType: EmbeddingType

override final def writeNounToClustersIndex(

output: TypedPipe[(ProducerId, Seq[(ClusterId, Double)])]

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

val convertedEmbeddings = convertEmbeddingToAggregatableEmbeddings(output)

.map {

case (producerId, topSimClustersWithScore) =>

val id = SimClustersEmbeddingId(

embeddingType = embeddingType,

modelVersion = modelVersion,

internalId = InternalId.UserId(producerId))

val embeddings = SimClustersEmbedding(topSimClustersWithScore.map {

case (clusterId, score) => SimClusterWithScore(clusterId, score)

})

SimClustersEmbeddingWithId(id, embeddings)

}

val keyValuePairs = convertedEmbeddings.map { simClustersEmbeddingWithId =>

KeyVal(simClustersEmbeddingWithId.embeddingId, simClustersEmbeddingWithId.embedding)

}

val manhattanExecution = writeToManhattan(keyValuePairs)

val thriftExecution = writeToThrift(convertedEmbeddings)

Execution.zip(manhattanExecution, thriftExecution).unit

}

}