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

import com.twitter.bijection.Injection

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

import com.twitter.scalding.DateRange

import com.twitter.scalding.Days

import com.twitter.scalding.Execution

import com.twitter.scalding.Stat

import com.twitter.scalding.TypedTsv

import com.twitter.scalding.UniqueID

import com.twitter.scalding.typed.TypedPipe

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

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

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

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

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

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

import com.twitter.simclusters\_v2.common.clustering.ClusteringMethod

import com.twitter.simclusters\_v2.common.clustering.ClusteringStatistics.\_

import com.twitter.simclusters\_v2.common.clustering.ClusterRepresentativeSelectionMethod

import com.twitter.simclusters\_v2.common.clustering.ClusterRepresentativeSelectionStatistics.\_

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

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

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

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

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

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

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

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

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

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

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

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

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

import com.twitter.simclusters\_v2.thriftscala.SimClustersMultiEmbedding.Ids

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

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

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

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

SimClustersEmbeddingId => SimClustersEmbeddingIdThrift

}

import com.twitter.util.Stopwatch

import java.util.TimeZone

import scala.util.Random.shuffle

/\*\*

\* Base app for computing User InterestedIn multi-embedding representation.

\* TWICE: Capturing users’ long-term interests using multiple SimClusters embeddings.

\* This job will

\* - Randomly select K follow/fav actions for each user,

\* - cluster the follow/fav actions for each user,

\* - for each cluster, construct a representation (e.g. average or medoid).

\*

\* @tparam T type of producer embedding. e.g. SimClustersEmbedding

\*/

trait InterestedInTwiceBaseApp[T] {

import InterestedInTwiceBaseApp.\_

def modelVersion: ModelVersion = ModelVersion.Model20m145k2020

/\*\*

\* function to output similarity (>=0, the larger, more similar), given two producer embeddings.

\*/

def producerProducerSimilarityFnForClustering: (T, T) => Double

def producerProducerSimilarityFnForClusterRepresentative: (T, T) => Double

// Sort clusters by decreasing size, fall back to entity ID to break tie

val clusterOrdering: Ordering[Set[Long]] = math.Ordering.by(c => (-c.size, c.min))

/\*\*

\* Read user-user graph.

\*/

def getUserUserGraph(

implicit dateRange: DateRange,

timeZone: TimeZone

): TypedPipe[UserAndNeighbors] = {

DAL

.readMostRecentSnapshot(

UserUserGraphScalaDataset

)

.withRemoteReadPolicy(AllowCrossDC)

.toTypedPipe

}

/\*\*

\* Randomly select up to maxNeighborsByUser neighbors for each user.

\* Attempts to equally sample both follow and fav edges (e.g. maxNeighborsByUser/2 for each).

\* However, if one type of edge is insufficient, backfill with other type up to maxNeighborsByUser neighbours.

\* @param userUserGraph User-User follow/fav graph.

\* @param maxNeighborsByUser How many neighbors to keep for each user.

\*/

def selectMaxProducersPerUser(

userUserGraph: TypedPipe[UserAndNeighbors],

maxNeighborsByUser: Int = MaxNeighborsByUser

)(

implicit uniqueID: UniqueID

): TypedPipe[UserAndNeighbors] = {

val numOfFollowEdgesStat = Stat(StatNumOfFollowEdges)

val numOfFavEdgesStat = Stat(StatNumOfFavEdges)

val numOfEdgesCumulativeFrequencyBeforeFilter = Util.CumulativeStat(

StatCFNumProducersPerConsumerBeforeFilter,

StatCFNumProducersPerConsumerBeforeFilterBuckets)

userUserGraph.map { userAndNeighbors: UserAndNeighbors =>

numOfEdgesCumulativeFrequencyBeforeFilter.incForValue(userAndNeighbors.neighbors.size)

val (followEdges, favEdges) =

userAndNeighbors.neighbors.partition(\_.isFollowed.contains(true))

val randomFollowEdges = shuffle(followEdges)

val randomFavEdges = shuffle(favEdges)

// interleave follow and fav edges, and select top k

val interleavedTopKEdges: Seq[NeighborWithWeights] = randomFollowEdges

.map(Some(\_))

.zipAll(

randomFavEdges.map(Some(\_)),

None,

None

) // default None value when one edge Seq is shorter than another

.flatMap {

case (followEdgeOpt, favEdgeOpt) =>

Seq(followEdgeOpt, favEdgeOpt)

}.flatten

.take(maxNeighborsByUser)

// edge stats

interleavedTopKEdges

.foreach { edge =>

if (edge.isFollowed.contains(true)) numOfFollowEdgesStat.inc()

else numOfFavEdgesStat.inc()

}

userAndNeighbors.copy(neighbors = interleavedTopKEdges)

}

}

/\*\*

\* Get multi embedding for each user:

\* - For each user, join their follow / fav - based neighbors to producer embeddings,

\* - Group these neighbors into clusters using the specified clusteringMethod,

\* - For each cluster, select the medoid as the representation.

\*

\* @param userUserGraph User-User follow/fav graph.

\* @param producerEmbedding producer embedding dataset. e.g. simclusters embeddings, simhash, etc.

\* @param clusteringMethod A method to group embeddings together.

\* @param maxClustersPerUser How many clusters to keep per user.

\* @param clusterRepresentativeSelectionMethod A method to select a cluster representative.

\* @param numReducers How many reducers to use for sketch operation.

\*/

def getMultiEmbeddingPerUser(

userUserGraph: TypedPipe[UserAndNeighbors],

producerEmbedding: TypedPipe[(UserId, T)],

clusteringMethod: ClusteringMethod,

maxClustersPerUser: Int = MaxClustersPerUser,

clusterRepresentativeSelectionMethod: ClusterRepresentativeSelectionMethod[T],

numReducers: Int

)(

implicit uniqueID: UniqueID

): TypedPipe[(UserId, Seq[Set[UserId]], SimClustersMultiEmbedding)] = {

val truncatedUserUserGraph: TypedPipe[UserAndNeighbors] = selectMaxProducersPerUser(

userUserGraph)

val validEdges: TypedPipe[(UserId, NeighborWithWeights)] =

truncatedUserUserGraph.flatMap {

case UserAndNeighbors(srcId, neighborsWithWeights) =>

neighborsWithWeights.map { neighborWithWeights =>

(

neighborWithWeights.neighborId, // producerId

neighborWithWeights.copy(neighborId = srcId))

}

}

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

val totalEdgesNonEmptyProducerEmbeddingsStat = Stat(StatTotalEdgesNonEmptyProducerEmbeddings)

val userClusterPairsBeforeTruncation = Stat(StatNumUserClusterPairsBeforeTruncation)

val userClusterPairsAfterTruncation = Stat(StatNumUserClusterPairsAfterTruncation)

val numUsers = Stat(StatNumUsers)

val numOfClustersCumulativeFrequencyBeforeFilter =

Util.CumulativeStat(StatCFNumOfClustersBeforeFilter, StatCFNumOfClustersBeforeFilterBuckets)

// map each clustering statistic to a scalding.Stat

val clusteringStatsMap: Map[String, Stat] = Map(

StatSimilarityGraphTotalBuildTime -> Stat(StatSimilarityGraphTotalBuildTime),

StatClusteringAlgorithmRunTime -> Stat(StatClusteringAlgorithmRunTime),

StatMedoidSelectionTime -> Stat(StatMedoidSelectionTime)

)

val cosineSimilarityCumulativeFrequencyBeforeFilter = Util.CumulativeStat(

StatCFCosineSimilarityBeforeFilter,

StatCFCosineSimilarityBeforeFilterBuckets)

val clusterRepresentativeSelectionTime = Stat(StatClusterRepresentativeSelectionTime)

validEdges

.sketch(numReducers)

.join(producerEmbedding)

.map {

case (producerId: UserId, (srcWithWeights: NeighborWithWeights, embedding)) =>

totalEdgesNonEmptyProducerEmbeddingsStat.inc()

(srcWithWeights.neighborId, (srcWithWeights.copy(neighborId = producerId), embedding))

}

.group

.toList

.map {

case (userId: UserId, embeddings: Seq[(NeighborWithWeights, T)]) =>

numUsers.inc()

val embeddingsMap: Map[Long, T] = embeddings.map {

case (n: NeighborWithWeights, e) => (n.neighborId, e)

}.toMap

val weightsMap: Map[Long, NeighborWithWeights] = embeddings.map {

case (n: NeighborWithWeights, \_) => (n.neighborId, n)

}.toMap

// 1. Cluster embeddings

val clusters: Set[Set[UserId]] =

clusteringMethod

.cluster[T](

embeddingsMap,

producerProducerSimilarityFnForClustering,

// Map.get() returns an Option, so will not throw.

// Use .foreach() to filter out potential Nones.

(name, incr) => {

clusteringStatsMap.get(name).foreach(ctr => ctr.incBy(incr))

if (name == StatComputedSimilarityBeforeFilter)

cosineSimilarityCumulativeFrequencyBeforeFilter.incForValue(incr)

}

)

// 2. Sort clusters

val sortedClusters: Seq[Set[UserId]] = clusters.toSeq.sorted(clusterOrdering)

// 3. Keep only a max number of clusters (avoid OOM)

userClusterPairsBeforeTruncation.incBy(sortedClusters.size)

numOfClustersCumulativeFrequencyBeforeFilter.incForValue(sortedClusters.size)

val truncatedClusters = sortedClusters.take(maxClustersPerUser)

userClusterPairsAfterTruncation.incBy(truncatedClusters.size)

// 4. Get list of cluster representatives

val truncatedIdWithScoreList: Seq[SimClustersEmbeddingIdWithScore] =

truncatedClusters.map { members: Set[UserId] =>

val clusterRepresentationSelectionElapsed = Stopwatch.start()

val medoid: UserId = clusterRepresentativeSelectionMethod.selectClusterRepresentative(

members.map(id => weightsMap(id)),

embeddingsMap)

clusterRepresentativeSelectionTime.incBy(

clusterRepresentationSelectionElapsed().inMilliseconds)

SimClustersEmbeddingIdWithScore(

id = SimClustersEmbeddingIdThrift(

EmbeddingType.TwiceUserInterestedIn,

modelVersion,

InternalId.UserId(medoid)),

score = members.size)

}

(

userId,

sortedClusters,

SimClustersMultiEmbedding.Ids(

SimClustersMultiEmbeddingByIds(ids = truncatedIdWithScoreList)))

}

}

/\*\*

\* Write the output to disk as a TypedTsv.

\*/

def writeOutputToTypedTSV(

output: TypedPipe[(UserId, Seq[Set[UserId]], SimClustersMultiEmbedding)],

userToClusterRepresentativesIndexOutputPath: String,

userToClusterMembersIndexOutputPath: String

): Execution[(Unit, Unit)] = {

// write the user -> cluster representatives index

val writeClusterRepresentatives = output

.collect {

case (userId: Long, \_, Ids(ids)) => (userId, ids.ids)

}

//.shard(partitions = 1)

.writeExecution(TypedTsv[(UserId, Seq[SimClustersEmbeddingIdWithScore])](

userToClusterRepresentativesIndexOutputPath))

// write the user -> cluster members index

val writeClusterMembers = output

.collect {

case (userId: Long, clusters: Seq[Set[UserId]], \_) => (userId, clusters)

}

//.shard(partitions = 1)

.writeExecution(TypedTsv[(UserId, Seq[Set[UserId]])](userToClusterMembersIndexOutputPath))

Execution.zip(writeClusterRepresentatives, writeClusterMembers)

}

/\*\*

\* Write the output to disk as a KeyValDataset.

\*/

def writeOutputToKeyValDataset(

output: TypedPipe[(UserId, Seq[Set[UserId]], SimClustersMultiEmbedding)],

embeddingType: MultiEmbeddingType,

userToClusterRepresentativesIndexDataset: KeyValDALDataset[

KeyVal[SimClustersMultiEmbeddingId, SimClustersMultiEmbedding]

],

userToClusterMembersIndexDataset: KeyValDALDataset[KeyVal[UserId, OrderedClustersAndMembers]],

userToClusterRepresentativesIndexOutputPath: String,

userToClusterMembersIndexOutputPath: String

)(

implicit dateRange: DateRange

): Execution[(Unit, Unit)] = {

// write the user -> cluster representatives index

val writeClusterRepresentatives = output

.map {

case (userId: UserId, \_, embeddings: SimClustersMultiEmbedding) =>

KeyVal(

key = SimClustersMultiEmbeddingId(

embeddingType = embeddingType,

modelVersion = modelVersion,

internalId = InternalId.UserId(userId)

),

value = embeddings

)

}

.writeDALVersionedKeyValExecution(

userToClusterRepresentativesIndexDataset,

D.Suffix(userToClusterRepresentativesIndexOutputPath),

ExplicitEndTime(dateRange.end)

)

// write the user -> cluster members index

val writeClusterMembers = output

.map {

case (userId: UserId, clusters: Seq[Set[UserId]], \_) =>

KeyVal(

key = userId,

value = OrderedClustersAndMembers(clusters, Some(clusters.map(ClusterMembers(\_)))))

}

.writeDALVersionedKeyValExecution(

userToClusterMembersIndexDataset,

D.Suffix(userToClusterMembersIndexOutputPath),

ExplicitEndTime(dateRange.end)

)

Execution.zip(writeClusterRepresentatives, writeClusterMembers)

}

/\*\*

\* Main method for scheduled jobs.

\*/

def runScheduledApp(

clusteringMethod: ClusteringMethod,

clusterRepresentativeSelectionMethod: ClusterRepresentativeSelectionMethod[T],

producerEmbedding: TypedPipe[(UserId, T)],

userToClusterRepresentativesIndexPathSuffix: String,

userToClusterMembersIndexPathSuffix: String,

userToClusterRepresentativesIndexDataset: KeyValDALDataset[

KeyVal[SimClustersMultiEmbeddingId, SimClustersMultiEmbedding]

],

userToClusterMembersIndexDataset: KeyValDALDataset[KeyVal[UserId, OrderedClustersAndMembers]],

numReducers: Int

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueId: UniqueID

): Execution[Unit] = {

val userToClusterRepresentativesIndexOutputPath: String = EmbeddingUtil.getHdfsPath(

isAdhoc = false,

isManhattanKeyVal = true,

modelVersion = modelVersion,

pathSuffix = userToClusterRepresentativesIndexPathSuffix

)

val userToClusterMembersIndexOutputPath: String = EmbeddingUtil.getHdfsPath(

isAdhoc = false,

isManhattanKeyVal = true,

modelVersion = modelVersion,

pathSuffix = userToClusterMembersIndexPathSuffix

)

val execution = Execution.withId { implicit uniqueId =>

val output: TypedPipe[(UserId, Seq[Set[UserId]], SimClustersMultiEmbedding)] =

getMultiEmbeddingPerUser(

userUserGraph = getUserUserGraph(dateRange.prepend(Days(30)), implicitly),

producerEmbedding = producerEmbedding,

clusteringMethod = clusteringMethod,

clusterRepresentativeSelectionMethod = clusterRepresentativeSelectionMethod,

numReducers = numReducers

)

writeOutputToKeyValDataset(

output = output,

embeddingType = MultiEmbeddingType.TwiceUserInterestedIn,

userToClusterRepresentativesIndexDataset = userToClusterRepresentativesIndexDataset,

userToClusterMembersIndexDataset = userToClusterMembersIndexDataset,

userToClusterRepresentativesIndexOutputPath = userToClusterRepresentativesIndexOutputPath,

userToClusterMembersIndexOutputPath = userToClusterMembersIndexOutputPath

)

}

execution.unit

}

/\*\*

\* Main method for adhoc jobs.

\*/

def runAdhocApp(

clusteringMethod: ClusteringMethod,

clusterRepresentativeSelectionMethod: ClusterRepresentativeSelectionMethod[T],

producerEmbedding: TypedPipe[(UserId, T)],

userToClusterRepresentativesIndexPathSuffix: String,

userToClusterMembersIndexPathSuffix: String,

numReducers: Int

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueId: UniqueID

): Execution[Unit] = {

val userToClusterRepresentativesIndexOutputPath: String = EmbeddingUtil.getHdfsPath(

isAdhoc = true,

isManhattanKeyVal = false,

modelVersion = modelVersion,

pathSuffix = userToClusterRepresentativesIndexPathSuffix

)

val userToClusterMembersIndexOutputPath: String = EmbeddingUtil.getHdfsPath(

isAdhoc = true,

isManhattanKeyVal = false,

modelVersion = modelVersion,

pathSuffix = userToClusterMembersIndexPathSuffix

)

val execution = Execution.withId { implicit uniqueId =>

val output: TypedPipe[(UserId, Seq[Set[UserId]], SimClustersMultiEmbedding)] =

getMultiEmbeddingPerUser(

userUserGraph = getUserUserGraph(dateRange.prepend(Days(30)), implicitly),

producerEmbedding = producerEmbedding,

clusteringMethod = clusteringMethod,

clusterRepresentativeSelectionMethod = clusterRepresentativeSelectionMethod,

numReducers = numReducers

)

writeOutputToTypedTSV(

output,

userToClusterRepresentativesIndexOutputPath,

userToClusterMembersIndexOutputPath)

}

execution.unit

}

}

object InterestedInTwiceBaseApp {

// Statistics

val StatNumOfFollowEdges = "num\_of\_follow\_edges"

val StatNumOfFavEdges = "num\_of\_fav\_edges"

val StatTotalEdgesNonEmptyProducerEmbeddings = "total\_edges\_with\_non\_empty\_producer\_embeddings"

val StatNumUserClusterPairsBeforeTruncation = "num\_user\_cluster\_pairs\_before\_truncation"

val StatNumUserClusterPairsAfterTruncation = "num\_user\_cluster\_pairs\_after\_truncation"

val StatNumUsers = "num\_users"

// Cumulative Frequency

val StatCFNumProducersPerConsumerBeforeFilter = "num\_producers\_per\_consumer\_cf\_before\_filter"

val StatCFNumProducersPerConsumerBeforeFilterBuckets: Seq[Double] =

Seq(0, 10, 20, 50, 100, 500, 1000)

val StatCFCosineSimilarityBeforeFilter = "cosine\_similarity\_cf\_before\_filter"

val StatCFCosineSimilarityBeforeFilterBuckets: Seq[Double] =

Seq(0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100)

val StatCFNumOfClustersBeforeFilter = "num\_of\_clusters\_cf\_before\_filter"

val StatCFNumOfClustersBeforeFilterBuckets: Seq[Double] =

Seq(1, 3, 5, 10, 15, 20, 50, 100, 200, 300, 500)

val MaxClustersPerUser: Int = 10

val MaxNeighborsByUser: Int = 500

object ProducerEmbeddingSource {

/\*\*

\* Read log-fav based Aggregatable Producer embeddings dataset.

\*/

def getAggregatableProducerEmbeddings(

implicit dateRange: DateRange,

timeZone: TimeZone

): TypedPipe[(UserId, SimClustersEmbedding)] =

ProducerEmbeddingSources

.producerEmbeddingSource(

EmbeddingType.AggregatableLogFavBasedProducer,

ModelVersion.Model20m145k2020)(dateRange.prepend(Days(30)))

.mapValues(s => SimClustersEmbedding(s))

}

}