package com.twitter.simclusters\_v2.scalding.optout

import com.twitter.algebird.Aggregator.size

import com.twitter.algebird.QTreeAggregatorLowerBound

import com.twitter.octain.identifiers.thriftscala.RawId

import com.twitter.octain.p13n.batch.P13NPreferencesScalaDataset

import com.twitter.octain.p13n.preferences.CompositeInterest

import com.twitter.scalding.DateRange

import com.twitter.scalding.Execution

import com.twitter.scalding.TypedPipe

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

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

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

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

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

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

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

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

import com.twitter.wtf.interest.thriftscala.Interest

/\*\*

\* Opts out InterestedIn clusters based on clusters' entity embeddings. If a user opted out an

\* entity and the user also is interested in a cluster with that entity embedding, unlink the

\* user from that entity.

\*/

object SimClustersOptOutUtil {

/\*\*

\* Reads User's Your Twitter Data opt-out selections

\*/

def getP13nOptOutSources(

dateRange: DateRange,

clusterType: ClusterType

): TypedPipe[(UserId, Set[SemanticCoreEntityId])] = {

DAL

.readMostRecentSnapshot(

P13NPreferencesScalaDataset,

dateRange

)

.withRemoteReadPolicy(AllowCrossClusterSameDC)

.toTypedPipe

.map { record => (record.id, record.preferences) }

.flatMap {

case (RawId.UserId(userId), p13nPreferences) =>

val optedOutEntities = p13nPreferences.interestPreferences

.map { preference =>

preference.disabledInterests

.collect {

case CompositeInterest.RecommendationInterest(recInterest)

if clusterType == ClusterType.InterestedIn =>

recInterest.interest match {

case Interest.SemanticEntityInterest(semanticCoreInterest) =>

Some(semanticCoreInterest.entityId)

case \_ =>

None

}

case CompositeInterest.RecommendationKnownFor(recInterest)

if clusterType == ClusterType.KnownFor =>

recInterest.interest match {

case Interest.SemanticEntityInterest(semanticCoreInterest) =>

Some(semanticCoreInterest.entityId)

case \_ =>

None

}

}.flatten.toSet

}.getOrElse(Set.empty)

if (optedOutEntities.nonEmpty) {

Some((userId, optedOutEntities))

} else {

None

}

case \_ =>

None

}

}

/\*\*

\* Remove user's clusters whose inferred entity embeddings are opted out. Will retain the user

\* entry in the pipe even if all the clusters are filtered out.

\*/

def filterOptedOutClusters(

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

optedOutEntities: TypedPipe[(UserId, Set[SemanticCoreEntityId])],

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

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

val inMemoryValidClusterToEntities =

legibleClusters

.mapValues(\_.map(\_.entityId).toSet)

.map(Map(\_)).sum

userToClusters

.leftJoin(optedOutEntities)

.mapWithValue(inMemoryValidClusterToEntities) {

case ((userId, (userClusters, optedOutEntitiesOpt)), validClusterToEntitiesOpt) =>

val optedOutEntitiesSet = optedOutEntitiesOpt.getOrElse(Set.empty)

val validClusterToEntities = validClusterToEntitiesOpt.getOrElse(Map.empty)

val clustersAfterOptOut = userClusters.filter { clusterId =>

val isClusterOptedOut = validClusterToEntities

.getOrElse(clusterId, Set.empty)

.intersect(optedOutEntitiesSet)

.nonEmpty

!isClusterOptedOut

}.distinct

(userId, clustersAfterOptOut)

}

.filter { \_.\_2.nonEmpty }

}

val AlertEmail = "no-reply@twitter.com"

/\*\*

\* Does sanity check on the results, to make sure the opt out outputs are comparable to the

\* raw version. If the delta in the number of users >= 0.1% or median of number of clusters per

\* user >= 1%, send alert emails

\*/

def sanityCheckAndSendEmail(

oldNumClustersPerUser: TypedPipe[Int],

newNumClustersPerUser: TypedPipe[Int],

modelVersion: String,

alertEmail: String

): Execution[Unit] = {

val oldNumUsersExec = oldNumClustersPerUser.aggregate(size).toOptionExecution

val newNumUsersExec = newNumClustersPerUser.aggregate(size).toOptionExecution

val oldMedianExec = oldNumClustersPerUser

.aggregate(QTreeAggregatorLowerBound(0.5))

.toOptionExecution

val newMedianExec = newNumClustersPerUser

.aggregate(QTreeAggregatorLowerBound(0.5))

.toOptionExecution

Execution

.zip(oldNumUsersExec, newNumUsersExec, oldMedianExec, newMedianExec)

.map {

case (Some(oldNumUsers), Some(newNumUsers), Some(oldMedian), Some(newMedian)) =>

val deltaNum = (newNumUsers - oldNumUsers).toDouble / oldNumUsers.toDouble

val deltaMedian = (oldMedian - newMedian) / oldMedian

val message =

s"num users before optout=$oldNumUsers,\n" +

s"num users after optout=$newNumUsers,\n" +

s"median num clusters per user before optout=$oldMedian,\n" +

s"median num clusters per user after optout=$newMedian\n"

println(message)

if (Math.abs(deltaNum) >= 0.001 || Math.abs(deltaMedian) >= 0.01) {

Util.sendEmail(

message,

s"Anomaly in $modelVersion opt out job. Please check cluster optout jobs in Eagleeye",

alertEmail

)

}

case err =>

Util.sendEmail(

err.toString(),

s"Anomaly in $modelVersion opt out job. Please check cluster optout jobs in Eagleeye",

alertEmail

)

}

}

}