package com.twitter.simclusters\_v2.scalding

import com.twitter.scalding.{DateOps, DateParser, Execution, Stat, TypedPipe, TypedTsv, UniqueID}

import com.twitter.scalding\_internal.job.TwitterExecutionApp

import com.twitter.simclusters\_v2.common.{ClusterId, UserId}

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

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

object CompareClusters {

def norm(a: Iterable[Float]): Float = {

math

.sqrt(a.map { x => x \* x }.sum).toFloat

}

def cosine(a: Map[Long, Float], b: Map[Long, Float]): Float = {

val intersect = a.toList.collect {

case (id, score) if b.contains(id) =>

score \* b(id)

}

val dot = if (intersect.nonEmpty) intersect.sum else 0

val aNorm = norm(a.values)

val bNorm = norm(b.values)

if (aNorm > 0 && bNorm > 0) {

dot / aNorm / bNorm

} else 0

}

/\*\*

\* Compare two known-for data set, and generate change in cluster assignment stats

\*/

def compareClusterAssignments(

newKnownFor: TypedPipe[(UserId, List[(ClusterId, Float)])],

oldKnownFor: TypedPipe[(UserId, List[(ClusterId, Float)])]

)(

implicit uniqueID: UniqueID

): Execution[String] = {

val emptyToSomething = Stat("no\_assignment\_to\_some")

val somethingToEmpty = Stat("some\_assignment\_to\_none")

val emptyToEmpty = Stat("empty\_to\_empty")

val sameCluster = Stat("same\_cluster")

val diffCluster = Stat("diff\_cluster")

val calculateStatExec = newKnownFor

.outerJoin(oldKnownFor)

.map {

case (userId, (newKnownForListOpt, oldKnownForListOpt)) =>

val newKnownFor = newKnownForListOpt.getOrElse(Nil)

val oldKnownFor = oldKnownForListOpt.getOrElse(Nil)

if (newKnownFor.nonEmpty && oldKnownFor.isEmpty) {

emptyToSomething.inc()

}

if (newKnownFor.isEmpty && oldKnownFor.nonEmpty) {

somethingToEmpty.inc()

}

if (newKnownFor.isEmpty && oldKnownFor.isEmpty) {

emptyToEmpty.inc()

}

if (newKnownFor.nonEmpty && oldKnownFor.nonEmpty) {

val newClusterId = newKnownFor.head.\_1

val oldClusterId = oldKnownFor.head.\_1

if (newClusterId == oldClusterId) {

sameCluster.inc()

} else {

diffCluster.inc()

}

}

userId

}

.toIterableExecution

Util.getCustomCountersString(calculateStatExec)

}

/\*\*

\* Compare two cluster assignments in terms of cosine similarity of corresponding clusters.

\* Excludes clusters which are too small

\* @param knownForA

\* @param knownForB

\* @param minSizeOfBiggerCluster Set to 10 or some such.

\* @return

\*/

def compare(

knownForA: TypedPipe[(Int, List[(Long, Float)])],

knownForB: TypedPipe[(Int, List[(Long, Float)])],

minSizeOfBiggerCluster: Int

): TypedPipe[(Int, Float)] = {

knownForA

.outerJoin(knownForB)

.collect {

case (clusterId, (membersInAOpt, membersInBOpt))

if membersInAOpt.exists(\_.size >= minSizeOfBiggerCluster) || membersInBOpt

.exists(\_.size >= minSizeOfBiggerCluster) =>

val membersInA =

membersInAOpt.map(\_.toMap).getOrElse(Map.empty[Long, Float])

val membersInB =

membersInBOpt.map(\_.toMap).getOrElse(Map.empty[Long, Float])

(clusterId, cosine(membersInA, membersInB))

}

}

def summarize(clusterToCosines: TypedPipe[(Int, Float)]): Execution[Option[Distribution]] = {

clusterToCosines.values.map(x => List(x)).sum.toOptionExecution.map { listOpt =>

listOpt.map { list => Util.distributionFromArray(list.map(\_.toDouble).toArray) }

}

}

}

object CompareClustersAdhoc extends TwitterExecutionApp {

implicit val tz: java.util.TimeZone = DateOps.UTC

implicit val dp = DateParser.default

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

val knownForA = KnownForSources.transpose(KnownForSources.readKnownFor(args("knownForA")))

val knownForB = KnownForSources.transpose(KnownForSources.readKnownFor(args("knownForB")))

CompareClusters

.compare(knownForA, knownForB, minSizeOfBiggerCluster = 10)

.map { case (cId, cos) => "%d\t%.2f".format(cId, cos) }

.writeExecution(TypedTsv(args("outputDir")))

}

}

}