package com.twitter.simclusters\_v2.scalding.tweet\_similarity

import com.twitter.ads.entities.db.thriftscala.PromotedTweet

import com.twitter.dataproducts.estimation.ReservoirSampler

import com.twitter.scalding.typed.TypedPipe

import com.twitter.scalding.{DateRange, Execution, TypedTsv}

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

import com.twitter.scalding\_internal.dalv2.remote\_access.{ExplicitLocation, Proc3Atla, ProcAtla}

import com.twitter.simclusters\_v2.common.{SimClustersEmbedding, Timestamp, TweetId, UserId}

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

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

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

TweetTopKTweetsWithScore,

TweetWithScore,

TweetsWithScore

}

import com.twitter.timelineservice.thriftscala.{ContextualizedFavoriteEvent, FavoriteEventUnion}

import com.twitter.wtf.scalding.client\_event\_processing.thriftscala.{

InteractionDetails,

InteractionType,

TweetImpressionDetails

}

import com.twitter.wtf.scalding.jobs.client\_event\_processing.UserInteractionScalaDataset

import java.util.Random

import scala.collection.mutable.ArrayBuffer

import scala.util.control.Breaks.\_

import twadoop\_config.configuration.log\_categories.group.timeline.TimelineServiceFavoritesScalaDataset

object TweetPairLabelCollectionUtil {

case class FeaturedTweet(

tweet: TweetId,

timestamp: Timestamp, //engagement or impression time

author: Option[UserId],

embedding: Option[SimClustersEmbedding])

extends Ordered[FeaturedTweet] {

import scala.math.Ordered.orderingToOrdered

def compare(that: FeaturedTweet): Int =

(this.tweet, this.timestamp, this.author) compare (that.tweet, that.timestamp, that.author)

}

val MaxFavPerUser: Int = 100

/\*\*

\* Get all fav events within the given dateRange and where all users' out-degree <= maxOutDegree

\* from TimelineServiceFavoritesScalaDataset

\*

\* @param dateRange date of interest

\* @param maxOutgoingDegree max #degrees for the users of interests

\*

\* @return Filtered fav events, TypedPipe of (userid, tweetid, timestamp) tuples

\*/

def getFavEvents(

dateRange: DateRange,

maxOutgoingDegree: Int

): TypedPipe[(UserId, TweetId, Timestamp)] = {

val fullTimelineFavData: TypedPipe[ContextualizedFavoriteEvent] =

DAL

.read(TimelineServiceFavoritesScalaDataset, dateRange)

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

val userTweetTuples = fullTimelineFavData

.flatMap { cfe: ContextualizedFavoriteEvent =>

cfe.event match {

case FavoriteEventUnion.Favorite(fav) =>

Some((fav.userId, (fav.tweetId, fav.eventTimeMs)))

case \_ =>

None

}

}

//Get users with the out-degree <= maxOutDegree first

val usersWithValidOutDegree = userTweetTuples

.groupBy(\_.\_1)

.withReducers(1000)

.size

.filter(\_.\_2 <= maxOutgoingDegree)

// Keep only usersWithValidOutDegree in the graph

userTweetTuples

.join(usersWithValidOutDegree).map {

case (userId, ((tweetId, eventTime), \_)) => (userId, tweetId, eventTime)

}.forceToDisk

}

/\*\*

\* Get impression events where users stay at the tweets for more than one minute

\*

\* @param dateRange time range of interest

\*

\* @return

\*/

def getImpressionEvents(dateRange: DateRange): TypedPipe[(UserId, TweetId, Timestamp)] = {

DAL

.read(UserInteractionScalaDataset, dateRange)

.withRemoteReadPolicy(ExplicitLocation(Proc3Atla))

.toTypedPipe

.flatMap {

case userInteraction

if userInteraction.interactionType == InteractionType.TweetImpressions =>

userInteraction.interactionDetails match {

case InteractionDetails.TweetImpressionDetails(

TweetImpressionDetails(tweetId, \_, dwellTimeInSecOpt))

if dwellTimeInSecOpt.exists(\_ >= 1) =>

Some(userInteraction.userId, tweetId, userInteraction.timeStamp)

case \_ =>

None

}

case \_ => None

}

.forceToDisk

}

/\*\*

\* Given an events dataset, return a filtered events limited to a given set of tweets

\*

\* @param events user fav events, a TypedPipe of (userid, tweetid, timestamp) tuples

\* @param tweets tweets of interest

\*

\* @return Filtered fav events on the given tweets of interest only, TypedPipe of (userid, tweetid, timestamp) tuples

\*/

def getFilteredEvents(

events: TypedPipe[(UserId, TweetId, Timestamp)],

tweets: TypedPipe[TweetId]

): TypedPipe[(UserId, TweetId, Timestamp)] = {

events

.map {

case (userId, tweetId, eventTime) => (tweetId, (userId, eventTime))

}

.join(tweets.asKeys)

.withReducers(1000)

.map {

case (tweetId, ((userId, eventTime), \_)) => (userId, tweetId, eventTime)

}

}

/\*\* Get (tweetId, author userId) of a given dateRange

\*

\* @param dateRange time range of interest

\*

\* @return TypedPipe of (tweetId, userId)

\*/

def getTweetAuthorPairs(dateRange: DateRange): TypedPipe[(TweetId, UserId)] = {

ExternalDataSources

.flatTweetsSource(dateRange)

.collect {

// Exclude retweets and quoted tweets

case record if record.shareSourceTweetId.isEmpty && record.quotedTweetTweetId.isEmpty =>

(record.tweetId, record.userId)

}

}

/\*\* Given a set of tweets, get all non-promoted tweets from the given set

\*

\* @param promotedTweets TypedPipe of promoted tweets

\* @param tweets tweets of interest

\*

\* @return TypedPipe of tweetId

\*/

def getNonPromotedTweets(

promotedTweets: TypedPipe[PromotedTweet],

tweets: TypedPipe[TweetId]

): TypedPipe[TweetId] = {

promotedTweets

.collect {

case promotedTweet if promotedTweet.tweetId.isDefined => promotedTweet.tweetId.get

}

.asKeys

.rightJoin(tweets.asKeys)

.withReducers(1000)

.filterNot(joined => joined.\_2.\_1.isDefined) //filter out those in promotedTweets

.keys

}

/\*\*

\* Given a fav events dataset, return all distinct ordered tweet pairs, labelled by whether they are co-engaged or not

\* Note we distinguish between (t1, t2) and (t2, t1) because o.w we introduce bias to training samples

\*

\* @param events user fav events, a TypedPipe of (userid, featuredTweet) tuples

\* @param timeframe two tweets will be considered co-engaged if they are fav-ed within coengagementTimeframe

\* @param isCoengaged if pairs are co-engaged

\*

\* @return labelled tweet pairs, TypedPipe of (userid, featuredTweet1, featuredTweet2, isCoengaged) tuples

\*/

def getTweetPairs(

events: TypedPipe[(UserId, FeaturedTweet)],

timeframe: Long,

isCoengaged: Boolean

): TypedPipe[(UserId, FeaturedTweet, FeaturedTweet, Boolean)] = {

events

.map {

case (userId, featuredTweet) => (userId, Seq(featuredTweet))

}

.sumByKey

.flatMap {

case (userId, featuredTweets) if featuredTweets.size > 1 =>

val sortedFeaturedTweet = featuredTweets.sortBy(\_.timestamp)

// Get all distinct ordered pairs that happen within coengagementTimeframe

val distinctPairs = ArrayBuffer[(UserId, FeaturedTweet, FeaturedTweet, Boolean)]()

breakable {

for (i <- sortedFeaturedTweet.indices) {

for (j <- i + 1 until sortedFeaturedTweet.size) {

val featuredTweet1 = sortedFeaturedTweet(i)

val featuredTweet2 = sortedFeaturedTweet(j)

if (math.abs(featuredTweet1.timestamp - featuredTweet2.timestamp) <= timeframe)

distinctPairs ++= Seq(

(userId, featuredTweet1, featuredTweet2, isCoengaged),

(userId, featuredTweet2, featuredTweet1, isCoengaged))

else

break

}

}

}

distinctPairs

case \_ => Nil

}

}

/\*\*

\* Get co-engaged tweet pairs

\*

\* @param favEvents user fav events, TypedPipe of (userid, tweetid, timestamp)

\* @param tweets tweets to be considered

\* @param coengagementTimeframe time window for two tweets to be considered as co-engaged

\*

\* @return TypedPipe of co-engaged tweet pairs

\*/

def getCoengagedPairs(

favEvents: TypedPipe[(UserId, TweetId, Timestamp)],

tweets: TypedPipe[TweetId],

coengagementTimeframe: Long

): TypedPipe[(UserId, FeaturedTweet, FeaturedTweet, Boolean)] = {

val userFeaturedTweetPairs =

getFilteredEvents(favEvents, tweets)

.map {

case (user, tweet, timestamp) => (user, FeaturedTweet(tweet, timestamp, None, None))

}

getTweetPairs(userFeaturedTweetPairs, coengagementTimeframe, isCoengaged = true)

}

/\*\*

\* Get co-impressed tweet pairs

\*

\* @param impressionEvents tweet impression events, TypedPipe of (userid, tweetid, timestamp)

\* @param tweets set of tweets considered to be part of co-impressed tweet pairs

\* @param timeframe time window for two tweets to be considered as co-impressed

\*

\* @return TypedPipe of co-impressed tweet pairs

\*/

def getCoimpressedPairs(

impressionEvents: TypedPipe[(UserId, TweetId, Timestamp)],

tweets: TypedPipe[TweetId],

timeframe: Long

): TypedPipe[(UserId, FeaturedTweet, FeaturedTweet, Boolean)] = {

val userFeaturedTweetPairs = getFilteredEvents(impressionEvents, tweets)

.map {

case (user, tweet, timestamp) => (user, FeaturedTweet(tweet, timestamp, None, None))

}

getTweetPairs(userFeaturedTweetPairs, timeframe, isCoengaged = false)

}

/\*\*

\* Consolidate co-engaged pairs and co-impressed pairs, and compute all the labelled tweet pairs

\* Given a pair:

\* label = 1 if co-engaged (whether or not it's co-impressed)

\* label = 0 if co-impressed and not co-engaged

\*

\* @param coengagedPairs co-engaged tweet pairs, TypedPipe of (user, queryFeaturedTweet, candidateFeaturedTweet, label)

\* @param coimpressedPairs co-impressed tweet pairs, TypedPipe of (user, queryFeaturedTweet, candidateFeaturedTweet, label)

\*

\* @return labelled tweet pairs, TypedPipe of (queryFeaturedTweet, candidateFeaturedTweet, label) tuples

\*/

def computeLabelledTweetPairs(

coengagedPairs: TypedPipe[(UserId, FeaturedTweet, FeaturedTweet, Boolean)],

coimpressedPairs: TypedPipe[(UserId, FeaturedTweet, FeaturedTweet, Boolean)]

): TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)] = {

(coengagedPairs ++ coimpressedPairs)

.groupBy {

case (userId, queryFeaturedTweet, candidateFeaturedTweet, \_) =>

(userId, queryFeaturedTweet.tweet, candidateFeaturedTweet.tweet)

}

// consolidate all the labelled pairs into one with the max label

// (label order: co-engagement = true > co-impression = false)

.maxBy {

case (\_, \_, \_, label) => label

}

.values

.map { case (\_, queryTweet, candidateTweet, label) => (queryTweet, candidateTweet, label) }

}

/\*\*

\* Get a balanced-class sampling of tweet pairs.

\* For each query tweet, we make sure the numbers of positives and negatives are equal.

\*

\* @param labelledPairs labelled tweet pairs, TypedPipe of (queryFeaturedTweet, candidateFeaturedTweet, label) tuples

\* @param maxSamplesPerClass max number of samples per class

\*

\* @return sampled labelled pairs after balanced-class sampling

\*/

def getQueryTweetBalancedClassPairs(

labelledPairs: TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)],

maxSamplesPerClass: Int

): TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)] = {

val queryTweetToSampleCount = labelledPairs

.map {

case (queryTweet, \_, label) =>

if (label) (queryTweet.tweet, (1, 0)) else (queryTweet.tweet, (0, 1))

}

.sumByKey

.map {

case (queryTweet, (posCount, negCount)) =>

(queryTweet, Math.min(Math.min(posCount, negCount), maxSamplesPerClass))

}

labelledPairs

.groupBy { case (queryTweet, \_, \_) => queryTweet.tweet }

.join(queryTweetToSampleCount)

.values

.map {

case ((queryTweet, candidateTweet, label), samplePerClass) =>

((queryTweet.tweet, label, samplePerClass), (queryTweet, candidateTweet, label))

}

.group

.mapGroup {

case ((\_, \_, samplePerClass), iter) =>

val random = new Random(123L)

val sampler =

new ReservoirSampler[(FeaturedTweet, FeaturedTweet, Boolean)](samplePerClass, random)

iter.foreach { pair => sampler.sampleItem(pair) }

sampler.sample.toIterator

}

.values

}

/\*\*

\* Given a user fav dataset, computes the similarity scores (based on engagers) between every tweet pairs

\*

\* @param events user fav events, a TypedPipe of (userid, tweetid, timestamp) tuples

\* @param minInDegree min number of engagement count for the tweets

\* @param coengagementTimeframe two tweets will be considered co-engaged if they are fav-ed within coengagementTimeframe

\*

\* @return tweet similarity based on engagers, a TypedPipe of (tweet1, tweet2, similarity\_score) tuples

\*\*/

def getScoredCoengagedTweetPairs(

events: TypedPipe[(UserId, TweetId, Timestamp)],

minInDegree: Int,

coengagementTimeframe: Long

)(

): TypedPipe[(TweetId, TweetWithScore)] = {

// compute tweet norms (based on engagers)

// only keep tweets whose indegree >= minInDegree

val tweetNorms = events

.map { case (\_, tweetId, \_) => (tweetId, 1.0) }

.sumByKey //the number of engagers per tweetId

.filter(\_.\_2 >= minInDegree)

.mapValues(math.sqrt)

val edgesWithWeight = events

.map {

case (userId, tweetId, eventTime) => (tweetId, (userId, eventTime))

}

.join(tweetNorms)

.map {

case (tweetId, ((userId, eventTime), norm)) =>

(userId, Seq((tweetId, eventTime, 1 / norm)))

}

// get cosine similarity

val tweetPairsWithWeight = edgesWithWeight.sumByKey

.flatMap {

case (\_, tweets) if tweets.size > 1 =>

allUniquePairs(tweets).flatMap {

case ((tweetId1, eventTime1, weight1), (tweetId2, eventTime2, weight2)) =>

// consider only co-engagement happened within the given timeframe

if ((eventTime1 - eventTime2).abs <= coengagementTimeframe) {

if (tweetId1 > tweetId2) // each worker generate allUniquePairs in different orders, hence should standardize the pairs

Some(((tweetId2, tweetId1), weight1 \* weight2))

else

Some(((tweetId1, tweetId2), weight1 \* weight2))

} else {

None

}

case \_ =>

None

}

case \_ => Nil

}

tweetPairsWithWeight.sumByKey

.flatMap {

case ((tweetId1, tweetId2), weight) =>

Seq(

(tweetId1, TweetWithScore(tweetId2, weight)),

(tweetId2, TweetWithScore(tweetId1, weight))

)

case \_ => Nil

}

}

/\*\*

\* Get the write exec for per-query stats

\*

\* @param tweetPairs input dataset

\* @param outputPath output path for the per-query stats

\* @param identifier identifier for the tweetPairs dataset

\*

\* @return execution of the the writing exec

\*/

def getPerQueryStatsExec(

tweetPairs: TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)],

outputPath: String,

identifier: String

): Execution[Unit] = {

val queryTweetsToCounts = tweetPairs

.map {

case (queryTweet, \_, label) =>

if (label) (queryTweet.tweet, (1, 0)) else (queryTweet.tweet, (0, 1))

}

.sumByKey

.map { case (queryTweet, (posCount, negCount)) => (queryTweet, posCount, negCount) }

Execution

.zip(

queryTweetsToCounts.writeExecution(

TypedTsv[(TweetId, Int, Int)](s"${outputPath}\_$identifier")),

Util.printSummaryOfNumericColumn(

queryTweetsToCounts

.map { case (\_, posCount, \_) => posCount },

Some(s"Per-query Positive Count ($identifier)")),

Util.printSummaryOfNumericColumn(

queryTweetsToCounts

.map { case (\_, \_, negCount) => negCount },

Some(s"Per-query Negative Count ($identifier)"))

).unit

}

/\*\*

\* Get the top K similar tweets key-val dataset

\*

\* @param allTweetPairs all tweet pairs with their similarity scores

\* @param k the maximum number of top results for each user

\*

\* @return key-val top K results for each tweet

\*/

def getKeyValTopKSimilarTweets(

allTweetPairs: TypedPipe[(TweetId, TweetWithScore)],

k: Int

)(

): TypedPipe[(TweetId, TweetsWithScore)] = {

allTweetPairs.group

.sortedReverseTake(k)(Ordering.by(\_.score))

.map { case (tweetId, tweetWithScoreSeq) => (tweetId, TweetsWithScore(tweetWithScoreSeq)) }

}

/\*\*

\* Get the top K similar tweets dataset.

\*

\* @param allTweetPairs all tweet pairs with their similarity scores

\* @param k the maximum number of top results for each user

\*

\* @return top K results for each tweet

\*/

def getTopKSimilarTweets(

allTweetPairs: TypedPipe[(TweetId, TweetWithScore)],

k: Int

)(

): TypedPipe[TweetTopKTweetsWithScore] = {

allTweetPairs.group

.sortedReverseTake(k)(Ordering.by(\_.score))

.map {

case (tweetId, tweetWithScoreSeq) =>

TweetTopKTweetsWithScore(tweetId, TweetsWithScore(tweetWithScoreSeq))

}

}

/\*\*

\* Given a input sequence, output all unique pairs in this sequence.

\*/

def allUniquePairs[T](input: Seq[T]): Stream[(T, T)] = {

input match {

case Nil => Stream.empty

case seq =>

seq.tail.toStream.map(a => (seq.head, a)) #::: allUniquePairs(seq.tail)

}

}

}