package com.twitter.simclusters\_v2.scalding.offline\_job.adhoc

import com.twitter.bijection.{Bufferable, Injection}

import com.twitter.scalding.\_

import com.twitter.scalding.commons.source.VersionedKeyValSource

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

import com.twitter.simclusters\_v2.scalding.common.matrix.SparseRowMatrix

import com.twitter.simclusters\_v2.scalding.offline\_job.SimClustersOfflineJobUtil

import com.twitter.wtf.scalding.jobs.common.AdhocExecutionApp

import java.util.TimeZone

/\*\*

\*

\* A job to sample some tweets for evaluation.

\*

\* we bucket tweets by the log(# of fav + 1) and randomly pick 1000 for each bucket for evaluation.

\*

\* to run the job:

\*

scalding remote run \

--target src/scala/com/twitter/simclusters\_v2/scalding/offline\_job/adhoc:tweet\_embedding\_evaluation\_samples-adhoc \

--user recos-platform \

--reducers 1000 \

--main-class com.twitter.simclusters\_v2.scalding.offline\_job.adhoc.TweetSimilarityEvaluationSamplingAdhocApp -- \

--date 2021-01-27 2021-01-28 \

--output /user/recos-platform/adhoc/tweet\_embedding\_01\_27\_28\_sample\_tweets

\*/

object TweetSimilarityEvaluationSamplingAdhocApp extends AdhocExecutionApp {

override def runOnDateRange(

args: Args

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

val random = new java.util.Random(args.long("seed", 20200322L))

// # of tweets in each bucket

val topK = args.int("bucket\_size", 1000)

val output = args("output")

SimClustersOfflineJobUtil

.readTimelineFavoriteData(dateRange)

.map {

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

tweetId -> 1L

}

.sumByKey

.filter(\_.\_2 >= 10L) // only consider tweets with more than 10 favs

.map {

case (tweetId, tweetFavs) =>

val bucket = math.log10(tweetFavs + 1.0).toInt

bucket -> (tweetId, random.nextDouble())

}

.group

.sortedReverseTake(topK)(Ordering.by(\_.\_2))

.flatMap {

case (bucket, tweets) =>

val bucketSize = tweets.length

tweets.map {

case (tweetId, \_) =>

(tweetId, bucket, bucketSize)

}

}

.writeExecution(

TypedTsv[(Long, Int, Int)](output)

)

}

}

/\*\*

\*

\* A job for evaluating the performance of an approximate nearest neighbor search method with a brute

\* force method.

\*

\* Evaluation method:

\*

\* After getting the embeddings for these tweets, we bucketize tweets based on the number of favs they have

\* (i.e., math.log10(numFavors).toInt), and then randomly select 1000 tweets from each bucket.

\* We do not include tweets with fewer than 10 favs. We compute the nearest neighbors (in terms of cosine similarity)

\* for these tweets using the brute force method and use up to top 100 neighbors with the cosine

\* similarity score >0.8 for each tweet as ground-truth set G.

\*

\* We then compute the nearest neighbors for these tweets based on the approximate nearest neighbor search: for each tweet, we find the top clusters, and then find top tweets in each cluster as potential candidates. We rank these potential candidates by the cosine similarity scores and take top 100 as prediction set P. We evaluate the precision and recall using

\*

\* Precision = |P \intersect G| / |P|

\* Recall = |P \intersect G| / |G|

\*

\* Note that |P| and |G| can be different, when there are not many neighbors returned.

\*

scalding remote run \

--target src/scala/com/twitter/simclusters\_v2/scalding/offline\_job/adhoc:tweet\_embedding\_evaluation-adhoc \

--user recos-platform \

--reducers 1000 \

--main-class com.twitter.simclusters\_v2.scalding.offline\_job.adhoc.TweetSimilarityEvaluationAdhocApp -- \

--date 2021-01-27 \

--tweet\_top\_k /user/recos-platform/adhoc/tweet\_embedding\_01\_27\_28\_unnormalized\_t9/tweet\_top\_k\_clusters \

--cluster\_top\_k /user/recos-platform/adhoc/tweet\_embedding\_01\_27\_28\_unnormalized\_t9/cluster\_top\_k\_tweets \

--tweets /user/recos-platform/adhoc/tweet\_embedding\_01\_27\_28\_sample\_tweets \

--output /user/recos-platform/adhoc/tweet\_embedding\_evaluation\_01\_27\_28\_t05\_k50\_1

\*/

object TweetSimilarityEvaluationAdhocApp extends AdhocExecutionApp {

implicit val inj1: Injection[List[(Int, Double)], Array[Byte]] =

Bufferable.injectionOf[List[(Int, Double)]]

implicit val inj2: Injection[List[(Long, Double)], Array[Byte]] =

Bufferable.injectionOf[List[(Long, Double)]]

// Take top 20 candidates, the score \* 100

private def formatList(candidates: Seq[(TweetId, Double)]): Seq[(TweetId, Int)] = {

candidates.take(10).map {

case (clusterId, score) =>

(clusterId, (score \* 100).toInt)

}

}

override def runOnDateRange(

args: Args

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

// path to read the tweet -> top cluster data set. should be the same from the SimClustersTweetEmbeddingAdhocApp job

val tweetTopKClustersPath = args("tweet\_top\_k")

// path to read the cluster -> top tweets data set. should be the same from the SimClustersTweetEmbeddingAdhocApp job

val clusterTopKTweetsPath = args("cluster\_top\_k")

// path to read the sampled tweets, should be the same from TweetSimilarityEvaluationSamplingAdhocApp

val tweetsPath = args("tweets")

// see the comment of this class. this is to determine which tweet should be ground truth

val threshold = args.double("threshold", 0.8)

// see the comment of this class. this is to determine which tweet should be ground truth

val topK = args.int("topK", 100)

// output path for evaluation results

val output = args("output")

// read tweet -> top clusters data set

val tweetTopKClusters: SparseRowMatrix[TweetId, ClusterId, Double] =

SparseRowMatrix(

TypedPipe

.from(

VersionedKeyValSource[TweetId, List[(ClusterId, Double)]](tweetTopKClustersPath)

)

.mapValues(\_.filter(\_.\_2 > 0.001).toMap),

isSkinnyMatrix = true

).rowL2Normalize

// read cluster -> top tweets data set

val clusterTopTweets: SparseRowMatrix[ClusterId, TweetId, Double] =

SparseRowMatrix(

TypedPipe

.from(

VersionedKeyValSource[ClusterId, List[(TweetId, Double)]](clusterTopKTweetsPath)

)

.mapValues(\_.filter(\_.\_2 > 0.02).toMap),

isSkinnyMatrix = false

)

// read the sampled tweets from TweetSimilarityEvaluationSamplingAdhocApp

val tweetSubset = TypedPipe.from(TypedTsv[(Long, Int, Int)](tweetsPath))

// the tweet -> top clusters for the sampled tweets

val tweetEmbeddingSubset =

tweetTopKClusters.filterRows(tweetSubset.map(\_.\_1))

// compute ground-truth top similar tweets for each sampled tweets.

// for each sampled tweets, we compute their similarity with every tweets in the tweet -> top clusters data set.

// we filter out those with similarity score smaller than the threshold and keep top k as the ground truth similar tweets

val groundTruthData = tweetTopKClusters.toSparseMatrix

.multiplySkinnySparseRowMatrix(

tweetEmbeddingSubset.toSparseMatrix.transpose.toSparseRowMatrix(true),

numReducersOpt = Some(5000)

)

.toSparseMatrix

.transpose

.filter((\_, \_, v) => v > threshold)

.sortWithTakePerRow(topK)(Ordering.by(-\_.\_2))

// compute approximate similar tweets for each sampled tweets.

// this is achieved by multiplying "sampled\_tweets -> top clusters" matrix with "cluster -> top tweets" matrix.

// note that in the implementation, we first compute the transponse of this matrix in order to ultlize the optimization done on skinny matrices

val predictionData = clusterTopTweets.toSparseMatrix.transpose

.multiplySkinnySparseRowMatrix(

tweetEmbeddingSubset.toSparseMatrix.transpose.toSparseRowMatrix(true),

numReducersOpt = Some(5000)

)

.toSparseMatrix

.transpose

.toTypedPipe

.map {

case (queryTweet, candidateTweet, \_) =>

(queryTweet, candidateTweet)

}

.join(tweetEmbeddingSubset.toTypedPipe)

.map {

case (queryId, (candidateId, queryEmbedding)) =>

candidateId -> (queryId, queryEmbedding)

}

.join(tweetTopKClusters.toTypedPipe)

.map {

case (candidateId, ((queryId, queryEmbedding), candidateEmbedding)) =>

queryId -> (candidateId, CosineSimilarityUtil

.dotProduct(

queryEmbedding,

candidateEmbedding

))

}

.filter(\_.\_2.\_2 > threshold)

.group

.sortedReverseTake(topK)(Ordering.by(\_.\_2))

// Exist in Ground Truth but not exist in Predication

val potentialData =

groundTruthData

.leftJoin(predictionData)

.map {

case (tweetId, (groundTruthCandidates, predictedCandidates)) =>

val predictedCandidateSet = predictedCandidates.toSeq.flatten.map(\_.\_1).toSet

val potentialTweets = groundTruthCandidates.filterNot {

case (candidateId, \_) =>

predictedCandidateSet.contains(candidateId)

}

(tweetId, potentialTweets)

}

val debuggingData =

groundTruthData

.leftJoin(predictionData)

.map {

case (tweetId, (groundTruthTweets, maybepredictedTweets)) =>

val predictedTweets = maybepredictedTweets.toSeq.flatten

val predictedTweetSet = predictedTweets.map(\_.\_1).toSet

val potentialTweets = groundTruthTweets.filterNot {

case (candidateId, \_) =>

predictedTweetSet.contains(candidateId)

}

(

tweetId,

Seq(

formatList(potentialTweets),

formatList(groundTruthTweets),

formatList(predictedTweets)))

}

// for each tweet, compare the approximate topk and ground-truth topk.

// compute precision and recall, then averaging them per bucket.

val eval = tweetSubset

.map {

case (tweetId, bucket, bucketSize) =>

tweetId -> (bucket, bucketSize)

}

.leftJoin(groundTruthData)

.leftJoin(predictionData)

.map {

case (\_, (((bucket, bucketSize), groundTruthOpt), predictionOpt)) =>

val groundTruth = groundTruthOpt.getOrElse(Nil).map(\_.\_1)

val prediction = predictionOpt.getOrElse(Nil).map(\_.\_1)

assert(groundTruth.distinct.size == groundTruth.size)

assert(prediction.distinct.size == prediction.size)

val intersection = groundTruth.toSet.intersect(prediction.toSet)

val precision =

if (prediction.nonEmpty)

intersection.size.toDouble / prediction.size.toDouble

else 0.0

val recall =

if (groundTruth.nonEmpty)

intersection.size.toDouble / groundTruth.size.toDouble

else 0.0

(

bucket,

bucketSize) -> (groundTruth.size, prediction.size, intersection.size, precision, recall, 1.0)

}

.sumByKey

.map {

case (

(bucket, bucketSize),

(groundTruthSum, predictionSum, interSectionSum, precisionSum, recallSum, count)) =>

(

bucket,

bucketSize,

groundTruthSum / count,

predictionSum / count,

interSectionSum / count,

precisionSum / count,

recallSum / count,

count)

}

// output the eval results and some sample results for eyeballing

Execution

.zip(

eval

.writeExecution(TypedTsv(output)),

groundTruthData

.map {

case (tweetId, neighbors) =>

tweetId -> neighbors

.map {

case (id, score) => s"$id:$score"

}

.mkString(",")

}

.writeExecution(

TypedTsv(args("output") + "\_ground\_truth")

),

predictionData

.map {

case (tweetId, neighbors) =>

tweetId -> neighbors

.map {

case (id, score) => s"$id:$score"

}

.mkString(",")

}

.writeExecution(

TypedTsv(args("output") + "\_prediction")

),

potentialData

.map {

case (tweetId, neighbors) =>

tweetId -> neighbors

.map {

case (id, score) => s"$id:$score"

}

.mkString(",")

}.writeExecution(

TypedTsv(args("output") + "\_potential")

),

debuggingData

.map {

case (tweetId, candidateList) =>

val value = candidateList

.map { candidates =>

candidates

.map {

case (id, score) =>

s"${id}D$score"

}.mkString("C")

}.mkString("B")

s"${tweetId}A$value"

}.writeExecution(

TypedTsv(args("output") + "\_debugging")

)

)

.unit

}

}