package com.twitter.simclustersann.candidate\_source

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

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

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

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

import com.twitter.simclustersann.thriftscala.ScoringAlgorithm

import com.twitter.simclustersann.thriftscala.SimClustersANNConfig

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.util.Duration

import com.twitter.util.Time

import com.google.common.collect.Comparators

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

/\*\*

\* A modified version of OptimizedApproximateCosineSimilarity which uses more java streams to avoid

\* materializing intermediate collections. Its performance is still under investigation.

\*/

object ExperimentalApproximateCosineSimilarity extends ApproximateCosineSimilarity {

final val InitialCandidateMapSize = 16384

val MaxNumResultsUpperBound = 1000

final val MaxTweetCandidateAgeUpperBound = 175200

private def parseTweetId(embeddingId: SimClustersEmbeddingId): Option[TweetId] = {

embeddingId.internalId match {

case InternalId.TweetId(tweetId) =>

Some(tweetId)

case \_ =>

None

}

}

private val CompareByScore: java.util.Comparator[(Long, Double)] =

new java.util.Comparator[(Long, Double)] {

override def compare(o1: (Long, Double), o2: (Long, Double)): Int = {

java.lang.Double.compare(o1.\_2, o2.\_2)

}

}

class Scores(var score: Double, var norm: Double)

override def apply(

sourceEmbedding: SimClustersEmbedding,

sourceEmbeddingId: SimClustersEmbeddingId,

config: SimClustersANNConfig,

candidateScoresStat: Int => Unit,

clusterTweetsMap: Map[ClusterId, Option[Seq[(TweetId, Double)]]] = Map.empty,

clusterTweetsMapArray: Map[ClusterId, Option[Array[(TweetId, Double)]]] = Map.empty

): Seq[ScoredTweet] = {

val now = Time.now

val earliestTweetId =

if (config.maxTweetCandidateAgeHours >= MaxTweetCandidateAgeUpperBound)

0L // Disable max tweet age filter

else

SnowflakeId.firstIdFor(now - Duration.fromHours(config.maxTweetCandidateAgeHours))

val latestTweetId =

SnowflakeId.firstIdFor(now - Duration.fromHours(config.minTweetCandidateAgeHours))

val candidateScoresMap = new java.util.HashMap[Long, Scores](InitialCandidateMapSize)

val sourceTweetId = parseTweetId(sourceEmbeddingId).getOrElse(0L)

clusterTweetsMap.foreach {

case (clusterId, Some(tweetScores)) =>

val sourceClusterScore = sourceEmbedding.getOrElse(clusterId)

for (i <- 0 until Math.min(tweetScores.size, config.maxTopTweetsPerCluster)) {

val (tweetId, score) = tweetScores(i)

if (tweetId >= earliestTweetId &&

tweetId <= latestTweetId &&

tweetId != sourceTweetId) {

val scores = candidateScoresMap.get(tweetId)

if (scores == null) {

val scorePair = new Scores(

score = score \* sourceClusterScore,

norm = score \* score

)

candidateScoresMap.put(tweetId, scorePair)

} else {

scores.score = scores.score + (score \* sourceClusterScore)

scores.norm = scores.norm + (score \* score)

}

}

}

case \_ => ()

}

candidateScoresStat(candidateScoresMap.size)

val normFn: (Long, Scores) => (Long, Double) = config.annAlgorithm match {

case ScoringAlgorithm.LogCosineSimilarity =>

(candidateId: Long, score: Scores) =>

(

candidateId,

score.score / sourceEmbedding.logNorm / math.log(1 + score.norm)

)

case ScoringAlgorithm.CosineSimilarity =>

(candidateId: Long, score: Scores) =>

(

candidateId,

score.score / sourceEmbedding.l2norm / math.sqrt(score.norm)

)

case ScoringAlgorithm.CosineSimilarityNoSourceEmbeddingNormalization =>

(candidateId: Long, score: Scores) =>

(

candidateId,

score.score / math.sqrt(score.norm)

)

case ScoringAlgorithm.DotProduct =>

(candidateId: Long, score: Scores) =>

(

candidateId,

score.score

)

}

import scala.collection.JavaConverters.\_

val topKCollector = Comparators.greatest(

Math.min(config.maxNumResults, MaxNumResultsUpperBound),

CompareByScore

)

candidateScoresMap

.entrySet().stream()

.map[(Long, Double)]((e: java.util.Map.Entry[Long, Scores]) => normFn(e.getKey, e.getValue))

.filter((s: (Long, Double)) => s.\_2 >= config.minScore)

.collect(topKCollector)

.asScala

}

}