package com.twitter.simclustersann.candidate\_source

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

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

/\*\*

\* Compared with ApproximateCosineSimilarity, this implementation:

\* - moves some computation aroudn to reduce allocations

\* - uses a single hashmap to store both scores and normalization coefficients

\* - uses some java collections in place of scala ones

\* Testing is still in progress, but this implementation shows significant (> 2x) improvements in

\* CPU utilization and allocations with 800 tweets per cluster.

\*/

object OptimizedApproximateCosineSimilarity 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

}

}

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, (Double, Double)](InitialCandidateMapSize)

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

clusterTweetsMap.foreach {

case (clusterId, Some(tweetScores)) if sourceEmbedding.contains(clusterId) =>

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.getOrDefault(tweetId, (0.0, 0.0))

val newScores = (

scores.\_1 + score \* sourceClusterScore,

scores.\_2 + score \* score,

)

candidateScoresMap.put(tweetId, newScores)

}

}

case \_ => ()

}

candidateScoresStat(candidateScoresMap.size)

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

case ScoringAlgorithm.LogCosineSimilarity =>

(candidateId: Long, score: (Double, Double)) =>

candidateId -> score.\_1 / sourceEmbedding.logNorm / math.log(1 + score.\_2)

case ScoringAlgorithm.CosineSimilarity =>

(candidateId: Long, score: (Double, Double)) =>

candidateId -> score.\_1 / sourceEmbedding.l2norm / math.sqrt(score.\_2)

case ScoringAlgorithm.CosineSimilarityNoSourceEmbeddingNormalization =>

(candidateId: Long, score: (Double, Double)) =>

candidateId -> score.\_1 / math.sqrt(score.\_2)

case ScoringAlgorithm.DotProduct =>

(candidateId: Long, score: (Double, Double)) => (candidateId, score.\_1)

}

val scoredTweets: java.util.ArrayList[(Long, Double)] =

new java.util.ArrayList(candidateScoresMap.size)

val it = candidateScoresMap.entrySet().iterator()

while (it.hasNext) {

val mapEntry = it.next()

val normedScore = normFn(mapEntry.getKey, mapEntry.getValue)

if (normedScore.\_2 >= config.minScore)

scoredTweets.add(normedScore)

}

import scala.collection.JavaConverters.\_

scoredTweets.asScala

.sortBy(-\_.\_2)

.take(Math.min(config.maxNumResults, MaxNumResultsUpperBound))

}

}