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

import scala.collection.mutable

/\*\*

\* This store looks for tweets whose similarity is close to a Source SimClustersEmbeddingId.

\*

\* Approximate cosine similarity is the core algorithm to drive this store.

\*

\* Step 1 - 4 are in "fetchCandidates" method.

\* 1. Retrieve the SimClusters Embedding by the SimClustersEmbeddingId

\* 2. Fetch top N clusters' top tweets from the clusterTweetCandidatesStore (TopTweetsPerCluster index).

\* 3. Calculate all the tweet candidates' dot-product or approximate cosine similarity to source tweets.

\* 4. Take top M tweet candidates by the step 3's score

\*/

trait ApproximateCosineSimilarity {

type ScoredTweet = (Long, Double)

def apply(

sourceEmbedding: SimClustersEmbedding,

sourceEmbeddingId: SimClustersEmbeddingId,

config: SimClustersANNConfig,

candidateScoresStat: Int => Unit,

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

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

): Seq[ScoredTweet]

}

object ApproximateCosineSimilarity extends ApproximateCosineSimilarity {

final val InitialCandidateMapSize = 16384

val MaxNumResultsUpperBound = 1000

final val MaxTweetCandidateAgeUpperBound = 175200

private class HashMap[A, B](initSize: Int) extends mutable.HashMap[A, B] {

override def initialSize: Int = initSize // 16 - by default

}

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))

// Use Mutable map to optimize performance. The method is thread-safe.

// Set initial map size to around p75 of map size distribution to avoid too many copying

// from extending the size of the mutable hashmap

val candidateScoresMap =

new HashMap[TweetId, Double](InitialCandidateMapSize)

val candidateNormalizationMap =

new HashMap[TweetId, Double](InitialCandidateMapSize)

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 (!parseTweetId(sourceEmbeddingId).contains(tweetId) &&

tweetId >= earliestTweetId && tweetId <= latestTweetId) {

candidateScoresMap.put(

tweetId,

candidateScoresMap.getOrElse(tweetId, 0.0) + score \* sourceClusterScore)

candidateNormalizationMap

.put(tweetId, candidateNormalizationMap.getOrElse(tweetId, 0.0) + score \* score)

}

}

case \_ => ()

}

candidateScoresStat(candidateScoresMap.size)

// Re-Rank the candidate by configuration

val processedCandidateScores: Seq[(TweetId, Double)] = candidateScoresMap.map {

case (candidateId, score) =>

// Enable Partial Normalization

val processedScore = {

// We applied the "log" version of partial normalization when we rank candidates

// by log cosine similarity

config.annAlgorithm match {

case ScoringAlgorithm.LogCosineSimilarity =>

score / sourceEmbedding.logNorm / math.log(1 + candidateNormalizationMap(candidateId))

case ScoringAlgorithm.CosineSimilarity =>

score / sourceEmbedding.l2norm / math.sqrt(candidateNormalizationMap(candidateId))

case ScoringAlgorithm.CosineSimilarityNoSourceEmbeddingNormalization =>

score / math.sqrt(candidateNormalizationMap(candidateId))

case ScoringAlgorithm.DotProduct => score

}

}

candidateId -> processedScore

}.toSeq

processedCandidateScores

.filter(\_.\_2 >= config.minScore)

.sortBy(-\_.\_2)

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

}

}