package com.twitter.cr\_mixer.similarity\_engine

import com.twitter.cr\_mixer.model.TripTweetWithScore

import com.twitter.cr\_mixer.param.ConsumerEmbeddingBasedTripParams

import com.twitter.cr\_mixer.util.InterleaveUtil

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.util.StatsUtil

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

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

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

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

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi

import com.twitter.timelines.configapi.Params

import com.twitter.trends.trip\_v1.trip\_tweets.thriftscala.Cluster

import com.twitter.trends.trip\_v1.trip\_tweets.thriftscala.ClusterDomain

import com.twitter.trends.trip\_v1.trip\_tweets.thriftscala.TripTweet

import com.twitter.trends.trip\_v1.trip\_tweets.thriftscala.TripDomain

import com.twitter.util.Future

case class TripEngineQuery(

modelId: String,

sourceId: InternalId,

tripSourceId: String,

maxResult: Int,

params: Params)

case class ConsumerEmbeddingBasedTripSimilarityEngine(

embeddingStoreLookUpMap: Map[String, ReadableStore[UserId, SimClustersEmbedding]],

tripCandidateSource: ReadableStore[TripDomain, Seq[TripTweet]],

statsReceiver: StatsReceiver,

) extends ReadableStore[TripEngineQuery, Seq[TripTweetWithScore]] {

import ConsumerEmbeddingBasedTripSimilarityEngine.\_

private val scopedStats = statsReceiver.scope(name)

private def fetchTopClusters(query: TripEngineQuery): Future[Option[Seq[ClusterId]]] = {

query.sourceId match {

case InternalId.UserId(userId) =>

val embeddingStore = embeddingStoreLookUpMap.getOrElse(

query.modelId,

throw new IllegalArgumentException(

s"${this.getClass.getSimpleName}: " +

s"ModelId ${query.modelId} does not exist for embeddingStore"

)

)

embeddingStore.get(userId).map(\_.map(\_.topClusterIds(MaxClusters)))

case \_ =>

Future.None

}

}

private def fetchCandidates(

topClusters: Seq[ClusterId],

tripSourceId: String

): Future[Seq[Seq[TripTweetWithScore]]] = {

Future

.collect {

topClusters.map { clusterId =>

tripCandidateSource

.get(

TripDomain(

sourceId = tripSourceId,

clusterDomain = Some(

ClusterDomain(simCluster = Some(Cluster(clusterIntId = Some(clusterId))))))).map {

\_.map {

\_.collect {

case TripTweet(tweetId, score) =>

TripTweetWithScore(tweetId, score)

}

}.getOrElse(Seq.empty).take(MaxNumResultsPerCluster)

}

}

}

}

override def get(engineQuery: TripEngineQuery): Future[Option[Seq[TripTweetWithScore]]] = {

val fetchTopClustersStat = scopedStats.scope(engineQuery.modelId).scope("fetchTopClusters")

val fetchCandidatesStat = scopedStats.scope(engineQuery.modelId).scope("fetchCandidates")

for {

topClustersOpt <- StatsUtil.trackOptionStats(fetchTopClustersStat) {

fetchTopClusters(engineQuery)

}

candidates <- StatsUtil.trackItemsStats(fetchCandidatesStat) {

topClustersOpt match {

case Some(topClusters) => fetchCandidates(topClusters, engineQuery.tripSourceId)

case None => Future.Nil

}

}

} yield {

val interleavedTweets = InterleaveUtil.interleave(candidates)

val dedupCandidates = interleavedTweets

.groupBy(\_.tweetId).flatMap {

case (\_, tweetWithScoreSeq) => tweetWithScoreSeq.sortBy(-\_.score).take(1)

}.toSeq.take(engineQuery.maxResult)

Some(dedupCandidates)

}

}

}

object ConsumerEmbeddingBasedTripSimilarityEngine {

private val MaxClusters: Int = 8

private val MaxNumResultsPerCluster: Int = 25

private val name: String = this.getClass.getSimpleName

def fromParams(

modelId: String,

sourceId: InternalId,

params: configapi.Params

): TripEngineQuery = {

TripEngineQuery(

modelId = modelId,

sourceId = sourceId,

tripSourceId = params(ConsumerEmbeddingBasedTripParams.SourceIdParam),

maxResult = params(ConsumerEmbeddingBasedTripParams.MaxNumCandidatesParam),

params = params

)

}

}