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

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

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

import com.twitter.cr\_mixer.thriftscala.SimilarityEngineType

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.Stats

import com.twitter.product\_mixer.core.thriftscala.ClientContext

import com.twitter.qig\_ranker.thriftscala.Product

import com.twitter.qig\_ranker.thriftscala.ProductContext

import com.twitter.qig\_ranker.thriftscala.QigRanker

import com.twitter.qig\_ranker.thriftscala.QigRankerProductResponse

import com.twitter.qig\_ranker.thriftscala.QigRankerRequest

import com.twitter.qig\_ranker.thriftscala.QigRankerResponse

import com.twitter.qig\_ranker.thriftscala.TwistlySimilarTweetsProductContext

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

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi

import com.twitter.util.Future

import javax.inject.Singleton

/\*\*

\* This store looks for similar tweets from QueryInteractionGraph (QIG) for a source tweet id.

\* For a given query tweet, QIG returns us the similar tweets that have an overlap of engagements

\* (with the query tweet) on different search queries

\*/

@Singleton

case class TweetBasedQigSimilarityEngine(

qigRanker: QigRanker.MethodPerEndpoint,

statsReceiver: StatsReceiver)

extends ReadableStore[

TweetBasedQigSimilarityEngine.Query,

Seq[TweetWithScore]

] {

private val stats = statsReceiver.scope(this.getClass.getSimpleName)

private val fetchCandidatesStat = stats.scope("fetchCandidates")

override def get(

query: TweetBasedQigSimilarityEngine.Query

): Future[Option[Seq[TweetWithScore]]] = {

query.sourceId match {

case InternalId.TweetId(tweetId) =>

val qigSimilarTweetsRequest = getQigSimilarTweetsRequest(tweetId)

Stats.trackOption(fetchCandidatesStat) {

qigRanker

.getSimilarCandidates(qigSimilarTweetsRequest)

.map { qigSimilarTweetsResponse =>

getCandidatesFromQigResponse(qigSimilarTweetsResponse)

}

}

case \_ =>

Future.value(None)

}

}

private def getQigSimilarTweetsRequest(

tweetId: Long

): QigRankerRequest = {

// Note: QigRanker needs a non-empty userId to be passed to return results.

// We are passing in a dummy userId until we fix this on QigRanker side

val clientContext = ClientContext(userId = Some(0L))

val productContext = ProductContext.TwistlySimilarTweetsProductContext(

TwistlySimilarTweetsProductContext(tweetId = tweetId))

QigRankerRequest(

clientContext = clientContext,

product = Product.TwistlySimilarTweets,

productContext = Some(productContext),

)

}

private def getCandidatesFromQigResponse(

qigSimilarTweetsResponse: QigRankerResponse

): Option[Seq[TweetWithScore]] = {

qigSimilarTweetsResponse.productResponse match {

case QigRankerProductResponse

.TwistlySimilarTweetCandidatesResponse(response) =>

val tweetsWithScore = response.similarTweets

.map { similarTweetResult =>

TweetWithScore(

similarTweetResult.tweetResult.tweetId,

similarTweetResult.tweetResult.score.getOrElse(0L))

}

Some(tweetsWithScore)

case \_ => None

}

}

}

object TweetBasedQigSimilarityEngine {

def toSimilarityEngineInfo(score: Double): SimilarityEngineInfo = {

SimilarityEngineInfo(

similarityEngineType = SimilarityEngineType.Qig,

modelId = None,

score = Some(score))

}

case class Query(sourceId: InternalId)

def fromParams(

sourceId: InternalId,

params: configapi.Params,

): EngineQuery[Query] = {

EngineQuery(

Query(sourceId = sourceId),

params

)

}

}