package com.twitter.frigate.pushservice.util

import com.twitter.contentrecommender.thriftscala.MetricTag

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.OutOfNetworkTweetCandidate

import com.twitter.frigate.common.base.SocialContextAction

import com.twitter.frigate.common.base.SocialContextActions

import com.twitter.frigate.common.base.TargetInfo

import com.twitter.frigate.common.base.TargetUser

import com.twitter.frigate.common.base.TopicProofTweetCandidate

import com.twitter.frigate.common.base.TweetAuthorDetails

import com.twitter.frigate.common.candidate.TargetABDecider

import com.twitter.frigate.common.rec\_types.RecTypes

import com.twitter.frigate.pushservice.model.PushTypes.PushCandidate

import com.twitter.frigate.pushservice.model.PushTypes.RawCandidate

import com.twitter.frigate.pushservice.params.CrtGroupEnum

import com.twitter.frigate.pushservice.params.PushFeatureSwitchParams

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.frigate.thriftscala.CommonRecommendationType.TripGeoTweet

import com.twitter.frigate.thriftscala.CommonRecommendationType.TripHqTweet

import com.twitter.frigate.thriftscala.{SocialContextAction => TSocialContextAction}

import com.twitter.util.Future

object CandidateUtil {

private val mrTwistlyMetricTags =

Seq(MetricTag.PushOpenOrNtabClick, MetricTag.RequestHealthFilterPushOpenBasedTweetEmbedding)

def getSocialContextActionsFromCandidate(candidate: RawCandidate): Seq[TSocialContextAction] = {

candidate match {

case candidateWithSocialContex: RawCandidate with SocialContextActions =>

candidateWithSocialContex.socialContextActions.map { scAction =>

TSocialContextAction(

scAction.userId,

scAction.timestampInMillis,

scAction.tweetId

)

}

case \_ => Seq.empty

}

}

/\*\*

\* Ranking Social Context based on the Real Graph weight

\* @param socialContextActions Sequence of Social Context Actions

\* @param seedsWithWeight Real Graph map consisting of User ID as key and RG weight as the value

\* @param defaultToRecency Boolean to represent if we should use the timestamp of the SC to rank

\* @return Returns the ranked sequence of SC Actions

\*/

def getRankedSocialContext(

socialContextActions: Seq[SocialContextAction],

seedsWithWeight: Future[Option[Map[Long, Double]]],

defaultToRecency: Boolean

): Future[Seq[SocialContextAction]] = {

seedsWithWeight.map {

case Some(followingsMap) =>

socialContextActions.sortBy { action => -followingsMap.getOrElse(action.userId, 0.0) }

case \_ =>

if (defaultToRecency) socialContextActions.sortBy(-\_.timestampInMillis)

else socialContextActions

}

}

def shouldApplyHealthQualityFiltersForPrerankingPredicates(

candidate: TweetAuthorDetails with TargetInfo[TargetUser with TargetABDecider]

)(

implicit stats: StatsReceiver

): Future[Boolean] = {

candidate.tweetAuthor.map {

case Some(user) =>

val numFollowers: Double = user.counts.map(\_.followers.toDouble).getOrElse(0.0)

numFollowers < candidate.target

.params(PushFeatureSwitchParams.NumFollowerThresholdForHealthAndQualityFiltersPreranking)

case \_ => true

}

}

def shouldApplyHealthQualityFilters(

candidate: PushCandidate

)(

implicit stats: StatsReceiver

): Boolean = {

val numFollowers =

candidate.numericFeatures.getOrElse("RecTweetAuthor.User.ActiveFollowers", 0.0)

numFollowers < candidate.target

.params(PushFeatureSwitchParams.NumFollowerThresholdForHealthAndQualityFilters)

}

def useAggressiveHealthThresholds(cand: PushCandidate): Boolean =

isMrTwistlyCandidate(cand) ||

(cand.commonRecType == CommonRecommendationType.GeoPopTweet && cand.target.params(

PushFeatureSwitchParams.PopGeoTweetEnableAggressiveThresholds))

def isMrTwistlyCandidate(cand: PushCandidate): Boolean =

cand match {

case oonCandidate: PushCandidate with OutOfNetworkTweetCandidate =>

oonCandidate.tagsCR

.getOrElse(Seq.empty).intersect(mrTwistlyMetricTags).nonEmpty && oonCandidate.tagsCR

.map(\_.toSet.size).getOrElse(0) == 1

case oonCandidate: PushCandidate with TopicProofTweetCandidate

if cand.target.params(PushFeatureSwitchParams.EnableHealthFiltersForTopicProofTweet) =>

oonCandidate.tagsCR

.getOrElse(Seq.empty).intersect(mrTwistlyMetricTags).nonEmpty && oonCandidate.tagsCR

.map(\_.toSet.size).getOrElse(0) == 1

case \_ => false

}

def getTagsCRCount(cand: PushCandidate): Double =

cand match {

case oonCandidate: PushCandidate with OutOfNetworkTweetCandidate =>

oonCandidate.tagsCR.map(\_.toSet.size).getOrElse(0).toDouble

case oonCandidate: PushCandidate with TopicProofTweetCandidate

if cand.target.params(PushFeatureSwitchParams.EnableHealthFiltersForTopicProofTweet) =>

oonCandidate.tagsCR.map(\_.toSet.size).getOrElse(0).toDouble

case \_ => 0.0

}

def isRelatedToMrTwistlyCandidate(cand: PushCandidate): Boolean =

cand match {

case oonCandidate: PushCandidate with OutOfNetworkTweetCandidate =>

oonCandidate.tagsCR.getOrElse(Seq.empty).intersect(mrTwistlyMetricTags).nonEmpty

case oonCandidate: PushCandidate with TopicProofTweetCandidate

if cand.target.params(PushFeatureSwitchParams.EnableHealthFiltersForTopicProofTweet) =>

oonCandidate.tagsCR.getOrElse(Seq.empty).intersect(mrTwistlyMetricTags).nonEmpty

case \_ => false

}

def getCrtGroup(commonRecType: CommonRecommendationType): CrtGroupEnum.Value = {

commonRecType match {

case crt if RecTypes.twistlyTweets(crt) => CrtGroupEnum.Twistly

case crt if RecTypes.frsTypes(crt) => CrtGroupEnum.Frs

case crt if RecTypes.f1RecTypes(crt) => CrtGroupEnum.F1

case crt if crt == TripGeoTweet || crt == TripHqTweet => CrtGroupEnum.Trip

case crt if RecTypes.TopicTweetTypes(crt) => CrtGroupEnum.Topic

case crt if RecTypes.isGeoPopTweetType(crt) => CrtGroupEnum.GeoPop

case \_ => CrtGroupEnum.Other

}

}

}