package com.twitter.frigate.pushservice.rank

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.CandidateDetails

import com.twitter.frigate.common.base.TweetAuthor

import com.twitter.frigate.common.base.TweetCandidate

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

import com.twitter.storehaus.FutureOps

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

class SubscriptionCreatorRanker(

superFollowEligibilityUserStore: ReadableStore[Long, Boolean],

statsReceiver: StatsReceiver) {

private val scopedStats = statsReceiver.scope("SubscriptionCreatorRanker")

private val boostStats = scopedStats.scope("boostSubscriptionCreator")

private val softUprankStats = scopedStats.scope("boostByScoreFactor")

private val boostTotalCandidates = boostStats.stat("total\_input\_candidates")

private val softRankTotalCandidates = softUprankStats.stat("total\_input\_candidates")

private val softRankNumCandidatesCreators = softUprankStats.counter("candidates\_from\_creators")

private val softRankNumCandidatesNonCreators =

softUprankStats.counter("candidates\_not\_from\_creators")

private val boostNumCandidatesCreators = boostStats.counter("candidates\_from\_creators")

private val boostNumCandidatesNonCreators =

boostStats.counter("candidates\_not\_from\_creators")

def boostSubscriptionCreator(

inputCandidatesFut: Future[Seq[CandidateDetails[PushCandidate]]]

): Future[Seq[CandidateDetails[PushCandidate]]] = {

inputCandidatesFut.flatMap { inputCandidates =>

boostTotalCandidates.add(inputCandidates.size)

val tweetAuthorIds = inputCandidates.flatMap {

case CandidateDetails(candidate: TweetCandidate with TweetAuthor, s) =>

candidate.authorId

case \_ => None

}.toSet

FutureOps

.mapCollect(superFollowEligibilityUserStore.multiGet(tweetAuthorIds))

.map { creatorAuthorMap =>

val (upRankedCandidates, otherCandidates) = inputCandidates.partition {

case CandidateDetails(candidate: TweetCandidate with TweetAuthor, s) =>

candidate.authorId match {

case Some(authorId) =>

creatorAuthorMap(authorId).getOrElse(false)

case \_ => false

}

case \_ => false

}

boostNumCandidatesCreators.incr(upRankedCandidates.size)

boostNumCandidatesNonCreators.incr(otherCandidates.size)

upRankedCandidates ++ otherCandidates

}

}

}

def boostByScoreFactor(

inputCandidatesFut: Future[Seq[CandidateDetails[PushCandidate]]],

factor: Double = 1.0,

): Future[Seq[CandidateDetails[PushCandidate]]] = {

inputCandidatesFut.flatMap { inputCandidates =>

softRankTotalCandidates.add(inputCandidates.size)

val tweetAuthorIds = inputCandidates.flatMap {

case CandidateDetails(candidate: TweetCandidate with TweetAuthor, s) =>

candidate.authorId

case \_ => None

}.toSet

FutureOps

.mapCollect(superFollowEligibilityUserStore.multiGet(tweetAuthorIds))

.flatMap { creatorAuthorMap =>

val (upRankedCandidates, otherCandidates) = inputCandidates.partition {

case CandidateDetails(candidate: TweetCandidate with TweetAuthor, s) =>

candidate.authorId match {

case Some(authorId) =>

creatorAuthorMap(authorId).getOrElse(false)

case \_ => false

}

case \_ => false

}

softRankNumCandidatesCreators.incr(upRankedCandidates.size)

softRankNumCandidatesNonCreators.incr(otherCandidates.size)

ModelBasedRanker.rankBySpecifiedScore(

inputCandidates,

candidate => {

val isFromCreator = candidate match {

case candidate: TweetCandidate with TweetAuthor =>

candidate.authorId match {

case Some(authorId) =>

creatorAuthorMap(authorId).getOrElse(false)

case \_ => false

}

case \_ => false

}

candidate.mrWeightedOpenOrNtabClickRankingProbability.map {

case Some(score) =>

if (isFromCreator) Some(score \* factor)

else Some(score)

case \_ => None

}

}

)

}

}

}

}