package com.twitter.frigate.pushservice.rank

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

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

import com.twitter.util.Future

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.pushservice.params.MrQualityUprankingPartialTypeEnum

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

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

import com.twitter.frigate.pushservice.params.PushConstants.OoncQualityCombinedScore

object ModelBasedRanker {

def rankBySpecifiedScore(

candidatesDetails: Seq[CandidateDetails[PushCandidate]],

scoreExtractor: PushCandidate => Future[Option[Double]]

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

val scoredCandidatesFutures = candidatesDetails.map { cand =>

scoreExtractor(cand.candidate).map { scoreOp => (cand, scoreOp.getOrElse(0.0)) }

}

Future.collect(scoredCandidatesFutures).map { scores =>

val sorted = scores.sortBy { candidateDetails => -1 \* candidateDetails.\_2 }

sorted.map(\_.\_1)

}

}

def populatePredictionScoreStats(

candidatesDetails: Seq[CandidateDetails[PushCandidate]],

scoreExtractor: PushCandidate => Future[Option[Double]],

predictionScoreStats: StatsReceiver

): Unit = {

val scoreScaleFactorForStat = 10000

val statName = "prediction\_scores"

candidatesDetails.map {

case CandidateDetails(candidate, source) =>

val crt = candidate.commonRecType

scoreExtractor(candidate).map { scoreOp =>

val scaledScore = (scoreOp.getOrElse(0.0) \* scoreScaleFactorForStat).toFloat

predictionScoreStats.scope("all\_candidates").stat(statName).add(scaledScore)

predictionScoreStats.scope(crt.toString()).stat(statName).add(scaledScore)

}

}

}

def populateMrWeightedOpenOrNtabClickScoreStats(

candidatesDetails: Seq[CandidateDetails[PushCandidate]],

predictionScoreStats: StatsReceiver

): Unit = {

populatePredictionScoreStats(

candidatesDetails,

candidate => candidate.mrWeightedOpenOrNtabClickRankingProbability,

predictionScoreStats

)

}

def populateMrQualityUprankingScoreStats(

candidatesDetails: Seq[CandidateDetails[PushCandidate]],

predictionScoreStats: StatsReceiver

): Unit = {

populatePredictionScoreStats(

candidatesDetails,

candidate => candidate.mrQualityUprankingProbability,

predictionScoreStats

)

}

def rankByMrWeightedOpenOrNtabClickScore(

candidatesDetails: Seq[CandidateDetails[PushCandidate]]

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

rankBySpecifiedScore(

candidatesDetails,

candidate => candidate.mrWeightedOpenOrNtabClickRankingProbability

)

}

def transformSigmoid(

score: Double,

weight: Double = 1.0,

bias: Double = 0.0

): Double = {

val base = -1.0 \* (weight \* score + bias)

val cappedBase = math.max(math.min(base, 100.0), -100.0)

1.0 / (1.0 + math.exp(cappedBase))

}

def transformLinear(

score: Double,

bar: Double = 1.0

): Double = {

val positiveBar = math.abs(bar)

val cappedScore = math.max(math.min(score, positiveBar), -1.0 \* positiveBar)

cappedScore / positiveBar

}

def transformIdentity(

score: Double

): Double = score

def rankByQualityOoncCombinedScore(

candidatesDetails: Seq[CandidateDetails[PushCandidate]],

qualityScoreTransform: Double => Double,

qualityScoreBoost: Double = 1.0

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

rankBySpecifiedScore(

candidatesDetails,

candidate => {

val ooncScoreFutOpt: Future[Option[Double]] =

candidate.mrWeightedOpenOrNtabClickRankingProbability

val qualityScoreFutOpt: Future[Option[Double]] =

candidate.mrQualityUprankingProbability

Future

.join(

ooncScoreFutOpt,

qualityScoreFutOpt

).map {

case (Some(ooncScore), Some(qualityScore)) =>

val transformedQualityScore = qualityScoreTransform(qualityScore)

val combinedScore = ooncScore \* (1.0 + qualityScoreBoost \* transformedQualityScore)

candidate

.cacheExternalScore(OoncQualityCombinedScore, Future.value(Some(combinedScore)))

Some(combinedScore)

case \_ => None

}

}

)

}

def rerankByProducerQualityOoncCombinedScore(

candidateDetails: Seq[CandidateDetails[PushCandidate]]

)(

implicit stat: StatsReceiver

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

val scopedStat = stat.scope("producer\_quality\_reranking")

val oonCandidates = candidateDetails.filter {

case CandidateDetails(pushCandidate: PushCandidate, \_) =>

tweetCandidateSelector(pushCandidate, MrQualityUprankingPartialTypeEnum.Oon)

}

val rankedOonCandidatesFut = rankBySpecifiedScore(

oonCandidates,

candidate => {

val baseScoreFutureOpt: Future[Option[Double]] = {

val qualityCombinedScoreFutureOpt =

candidate.getExternalCachedScoreByName(OoncQualityCombinedScore)

val ooncScoreFutureOpt = candidate.mrWeightedOpenOrNtabClickRankingProbability

Future.join(qualityCombinedScoreFutureOpt, ooncScoreFutureOpt).map {

case (Some(qualityCombinedScore), \_) =>

scopedStat.counter("quality\_combined\_score").incr()

Some(qualityCombinedScore)

case (\_, ooncScoreOpt) =>

scopedStat.counter("oonc\_score").incr()

ooncScoreOpt

}

}

baseScoreFutureOpt.map {

case Some(baseScore) =>

val boostRatio = candidate.mrProducerQualityUprankingBoost.getOrElse(1.0)

if (boostRatio > 1.0) scopedStat.counter("author\_uprank").incr()

else if (boostRatio < 1.0) scopedStat.counter("author\_downrank").incr()

else scopedStat.counter("author\_noboost").incr()

Some(baseScore \* boostRatio)

case \_ =>

scopedStat.counter("empty\_score").incr()

None

}

}

)

rankedOonCandidatesFut.map { rankedOonCandidates =>

val sortedOonCandidateIterator = rankedOonCandidates.toIterator

candidateDetails.map { ooncRankedCandidate =>

val isOon = tweetCandidateSelector(

ooncRankedCandidate.candidate,

MrQualityUprankingPartialTypeEnum.Oon)

if (sortedOonCandidateIterator.hasNext && isOon)

sortedOonCandidateIterator.next()

else ooncRankedCandidate

}

}

}

def tweetCandidateSelector(

pushCandidate: PushCandidate,

selectedCandidateType: MrQualityUprankingPartialTypeEnum.Value

): Boolean = {

pushCandidate match {

case candidate: PushCandidate with TweetCandidate =>

selectedCandidateType match {

case MrQualityUprankingPartialTypeEnum.Oon =>

val crt = candidate.commonRecType

RecTypes.isOutOfNetworkTweetRecType(crt) || RecTypes.outOfNetworkTopicTweetTypes

.contains(crt)

case \_ => true

}

case \_ => false

}

}

}