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.Ranker

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

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

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

import com.twitter.frigate.pushservice.ml.HealthFeatureGetter

import com.twitter.frigate.pushservice.ml.PushMLModelScorer

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

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

import com.twitter.frigate.pushservice.params.PushMLModel

import com.twitter.frigate.pushservice.params.PushModelName

import com.twitter.frigate.pushservice.params.PushParams

import com.twitter.frigate.pushservice.util.MediaAnnotationsUtil.updateMediaCategoryStats

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.util.Future

import com.twitter.frigate.pushservice.params.MrQualityUprankingTransformTypeEnum

import com.twitter.storehaus.ReadableStore

import com.twitter.frigate.thriftscala.UserMediaRepresentation

import com.twitter.hss.api.thriftscala.UserHealthSignalResponse

class RFPHRanker(

randomRanker: Ranker[Target, PushCandidate],

weightedOpenOrNtabClickModelScorer: PushMLModelScorer,

subscriptionCreatorRanker: SubscriptionCreatorRanker,

userHealthSignalStore: ReadableStore[Long, UserHealthSignalResponse],

producerMediaRepresentationStore: ReadableStore[Long, UserMediaRepresentation],

stats: StatsReceiver)

extends PushserviceRanker[Target, PushCandidate] {

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

private val boostCRTsRanker = CRTBoostRanker(statsReceiver.scope("boost\_desired\_crts"))

private val crtDownRanker = CRTDownRanker(statsReceiver.scope("down\_rank\_desired\_crts"))

private val crtsToDownRank = statsReceiver.stat("crts\_to\_downrank")

private val crtsToUprank = statsReceiver.stat("crts\_to\_uprank")

private val randomRankingCounter = stats.counter("randomRanking")

private val mlRankingCounter = stats.counter("mlRanking")

private val disableAllRelevanceCounter = stats.counter("disableAllRelevance")

private val disableHeavyRankingCounter = stats.counter("disableHeavyRanking")

private val heavyRankerCandidateCounter = stats.counter("heavy\_ranker\_candidate\_count")

private val heavyRankerScoreStats = statsReceiver.scope("heavy\_ranker\_prediction\_scores")

private val producerUprankingCounter = statsReceiver.counter("producer\_quality\_upranking")

private val producerBoostedCounter = statsReceiver.counter("producer\_boosted\_candidates")

private val producerDownboostedCounter = statsReceiver.counter("producer\_downboosted\_candidates")

override def initialRank(

target: Target,

candidates: Seq[CandidateDetails[PushCandidate]]

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

heavyRankerCandidateCounter.incr(candidates.size)

updateMediaCategoryStats(candidates)(stats)

target.targetUserState

.flatMap { targetUserState =>

val useRandomRanking = target.skipMlRanker || target.params(

PushParams.UseRandomRankingParam

)

if (useRandomRanking) {

randomRankingCounter.incr()

randomRanker.rank(target, candidates)

} else if (target.params(PushParams.DisableAllRelevanceParam)) {

disableAllRelevanceCounter.incr()

Future.value(candidates)

} else if (target.params(PushParams.DisableHeavyRankingParam) || target.params(

PushFeatureSwitchParams.DisableHeavyRankingModelFSParam)) {

disableHeavyRankingCounter.incr()

Future.value(candidates)

} else {

mlRankingCounter.incr()

val scoredCandidatesFut = scoring(target, candidates)

target.rankingModelParam.map { rankingModelParam =>

val modelName = PushModelName(

PushMLModel.WeightedOpenOrNtabClickProbability,

target.params(rankingModelParam)).toString

ModelBasedRanker.populateMrWeightedOpenOrNtabClickScoreStats(

candidates,

heavyRankerScoreStats.scope(modelName)

)

}

if (target.params(

PushFeatureSwitchParams.EnableQualityUprankingCrtScoreStatsForHeavyRankingParam)) {

val modelName = PushModelName(

PushMLModel.FilteringProbability,

target.params(PushFeatureSwitchParams.QualityUprankingModelTypeParam)

).toString

ModelBasedRanker.populateMrQualityUprankingScoreStats(

candidates,

heavyRankerScoreStats.scope(modelName)

)

}

val ooncRankedCandidatesFut =

scoredCandidatesFut.flatMap(ModelBasedRanker.rankByMrWeightedOpenOrNtabClickScore)

val qualityUprankedCandidatesFut =

if (target.params(PushFeatureSwitchParams.EnableQualityUprankingForHeavyRankingParam)) {

ooncRankedCandidatesFut.flatMap { ooncRankedCandidates =>

val transformFunc: Double => Double =

target.params(PushFeatureSwitchParams.QualityUprankingTransformTypeParam) match {

case MrQualityUprankingTransformTypeEnum.Linear =>

ModelBasedRanker.transformLinear(

\_,

bar = target.params(

PushFeatureSwitchParams.QualityUprankingLinearBarForHeavyRankingParam))

case MrQualityUprankingTransformTypeEnum.Sigmoid =>

ModelBasedRanker.transformSigmoid(

\_,

weight = target.params(

PushFeatureSwitchParams.QualityUprankingSigmoidWeightForHeavyRankingParam),

bias = target.params(

PushFeatureSwitchParams.QualityUprankingSigmoidBiasForHeavyRankingParam)

)

case \_ => ModelBasedRanker.transformIdentity

}

ModelBasedRanker.rankByQualityOoncCombinedScore(

ooncRankedCandidates,

transformFunc,

target.params(PushFeatureSwitchParams.QualityUprankingBoostForHeavyRankingParam)

)

}

} else ooncRankedCandidatesFut

if (target.params(

PushFeatureSwitchParams.EnableProducersQualityBoostingForHeavyRankingParam)) {

producerUprankingCounter.incr()

qualityUprankedCandidatesFut.flatMap(cands =>

ModelBasedRanker.rerankByProducerQualityOoncCombinedScore(cands)(statsReceiver))

} else qualityUprankedCandidatesFut

}

}

}

private def scoring(

target: Target,

candidates: Seq[CandidateDetails[PushCandidate]]

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

val ooncScoredCandidatesFut = target.rankingModelParam.map { rankingModelParam =>

weightedOpenOrNtabClickModelScorer.scoreByBatchPredictionForModelVersion(

target,

candidates,

rankingModelParam

)

}

val scoredCandidatesFut = {

if (target.params(PushFeatureSwitchParams.EnableQualityUprankingForHeavyRankingParam)) {

ooncScoredCandidatesFut.map { candidates =>

weightedOpenOrNtabClickModelScorer.scoreByBatchPredictionForModelVersion(

target = target,

candidatesDetails = candidates,

modelVersionParam = PushFeatureSwitchParams.QualityUprankingModelTypeParam,

overridePushMLModelOpt = Some(PushMLModel.FilteringProbability)

)

}

} else ooncScoredCandidatesFut

}

scoredCandidatesFut.foreach { candidates =>

val oonCandidates = candidates.filter {

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

ModelBasedRanker.tweetCandidateSelector(

pushCandidate,

MrQualityUprankingPartialTypeEnum.Oon)

}

setProducerQuality(

target,

oonCandidates,

userHealthSignalStore,

producerMediaRepresentationStore)

}

}

private def setProducerQuality(

target: Target,

candidates: Seq[CandidateDetails[PushCandidate]],

userHealthSignalStore: ReadableStore[Long, UserHealthSignalResponse],

producerMediaRepresentationStore: ReadableStore[Long, UserMediaRepresentation]

): Unit = {

lazy val boostRatio =

target.params(PushFeatureSwitchParams.QualityUprankingBoostForHighQualityProducersParam)

lazy val downboostRatio =

target.params(PushFeatureSwitchParams.QualityUprankingDownboostForLowQualityProducersParam)

candidates.foreach {

case CandidateDetails(pushCandidate, \_) =>

HealthFeatureGetter

.getFeatures(pushCandidate, producerMediaRepresentationStore, userHealthSignalStore).map {

featureMap =>

val agathaNsfwScore = featureMap.numericFeatures.getOrElse("agathaNsfwScore", 0.5)

val textNsfwScore = featureMap.numericFeatures.getOrElse("textNsfwScore", 0.15)

val nudityRate = featureMap.numericFeatures.getOrElse("nudityRate", 0.0)

val activeFollowers = featureMap.numericFeatures.getOrElse("activeFollowers", 0.0)

val favorsRcvd28Days = featureMap.numericFeatures.getOrElse("favorsRcvd28Days", 0.0)

val tweets28Days = featureMap.numericFeatures.getOrElse("tweets28Days", 0.0)

val authorDislikeCount = featureMap.numericFeatures

.getOrElse("authorDislikeCount", 0.0)

val authorDislikeRate = featureMap.numericFeatures.getOrElse("authorDislikeRate", 0.0)

val authorReportRate = featureMap.numericFeatures.getOrElse("authorReportRate", 0.0)

val abuseStrikeTop2Percent =

featureMap.booleanFeatures.getOrElse("abuseStrikeTop2Percent", false)

val abuseStrikeTop1Percent =

featureMap.booleanFeatures.getOrElse("abuseStrikeTop1Percent", false)

val hasNsfwToken = featureMap.booleanFeatures.getOrElse("hasNsfwToken", false)

if ((activeFollowers > 3000000) ||

(activeFollowers > 1000000 && agathaNsfwScore < 0.7 && nudityRate < 0.01 && !hasNsfwToken && !abuseStrikeTop2Percent) ||

(activeFollowers > 100000 && agathaNsfwScore < 0.7 && nudityRate < 0.01 && !hasNsfwToken && !abuseStrikeTop2Percent &&

tweets28Days > 0 && favorsRcvd28Days / tweets28Days > 3000 && authorReportRate < 0.000001 && authorDislikeRate < 0.0005)) {

producerBoostedCounter.incr()

pushCandidate.setProducerQualityUprankingBoost(boostRatio)

} else if (activeFollowers < 5 || agathaNsfwScore > 0.9 || nudityRate > 0.03 || hasNsfwToken || abuseStrikeTop1Percent ||

textNsfwScore > 0.4 || (authorDislikeRate > 0.005 && authorDislikeCount > 5) ||

(tweets28Days > 56 && favorsRcvd28Days / tweets28Days < 100)) {

producerDownboostedCounter.incr()

pushCandidate.setProducerQualityUprankingBoost(downboostRatio)

} else pushCandidate.setProducerQualityUprankingBoost(1.0)

}

}

}

private def rerankBySubscriptionCreatorRanker(

target: Target,

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

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

if (target.params(PushFeatureSwitchParams.SoftRankCandidatesFromSubscriptionCreators)) {

val factor = target.params(PushFeatureSwitchParams.SoftRankFactorForSubscriptionCreators)

subscriptionCreatorRanker.boostByScoreFactor(rankedCandidates, factor)

} else

subscriptionCreatorRanker.boostSubscriptionCreator(rankedCandidates)

}

override def reRank(

target: Target,

rankedCandidates: Seq[CandidateDetails[PushCandidate]]

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

val numberOfF1Candidates =

rankedCandidates.count(candidateDetails =>

RecTypes.isF1Type(candidateDetails.candidate.commonRecType))

lazy val threshold =

target.params(PushFeatureSwitchParams.NumberOfF1CandidatesThresholdForOONBackfill)

lazy val enableOONBackfillBasedOnF1 =

target.params(PushFeatureSwitchParams.EnableOONBackfillBasedOnF1Candidates)

val f1BoostedCandidates =

if (enableOONBackfillBasedOnF1 && numberOfF1Candidates > threshold) {

boostCRTsRanker.boostCrtsToTopStableOrder(

rankedCandidates,

RecTypes.f1FirstDegreeTypes.toSeq)

} else rankedCandidates

val topTweetsByGeoDownRankedCandidates =

if (target.params(PushFeatureSwitchParams.BackfillRankTopTweetsByGeoCandidates)) {

crtDownRanker.downRank(

f1BoostedCandidates,

Seq(CommonRecommendationType.GeoPopTweet)

)

} else f1BoostedCandidates

val reRankedCandidatesWithBoostedCrts = {

val listOfCrtsToUpRank = target

.params(PushFeatureSwitchParams.ListOfCrtsToUpRank)

.flatMap(CommonRecommendationType.valueOf)

crtsToUprank.add(listOfCrtsToUpRank.size)

boostCRTsRanker.boostCrtsToTop(topTweetsByGeoDownRankedCandidates, listOfCrtsToUpRank)

}

val reRankedCandidatesWithDownRankedCrts = {

val listOfCrtsToDownRank = target

.params(PushFeatureSwitchParams.ListOfCrtsToDownRank)

.flatMap(CommonRecommendationType.valueOf)

crtsToDownRank.add(listOfCrtsToDownRank.size)

crtDownRanker.downRank(reRankedCandidatesWithBoostedCrts, listOfCrtsToDownRank)

}

val rerankBySubscriptionCreatorFut = {

if (target.params(PushFeatureSwitchParams.BoostCandidatesFromSubscriptionCreators)) {

rerankBySubscriptionCreatorRanker(

target,

Future.value(reRankedCandidatesWithDownRankedCrts))

} else Future.value(reRankedCandidatesWithDownRankedCrts)

}

rerankBySubscriptionCreatorFut

}

}