package com.twitter.follow\_recommendations.common.rankers.fatigue\_ranker

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.follow\_recommendations.common.base.Ranker

import com.twitter.follow\_recommendations.common.base.StatsUtil

import com.twitter.follow\_recommendations.common.models.CandidateUser

import com.twitter.follow\_recommendations.common.models.HasDisplayLocation

import com.twitter.follow\_recommendations.common.models.HasWtfImpressions

import com.twitter.follow\_recommendations.common.models.WtfImpression

import com.twitter.follow\_recommendations.common.rankers.common.RankerId.RankerId

import com.twitter.follow\_recommendations.common.rankers.utils.Utils

import com.twitter.product\_mixer.core.model.marshalling.request.HasClientContext

import com.twitter.servo.util.MemoizingStatsReceiver

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

import com.twitter.util.Time

/\*\*

\* Ranks candidates based on the given weights for each algorithm while preserving the ranks inside each algorithm.

\* Reorders the ranked list based on recent impressions from recentImpressionRepo

\*

\* Note that the penalty is added to the rank of each candidate. To make producer-side experiments

\* with multiple rankers possible, we modify the scores for each candidate and ranker as:

\* NewScore(C, R) = -(Rank(C, R) + Impression(C, U) x FatigueFactor),

\* where C is a candidate, R a ranker and U the target user.

\* Note also that fatigue penalty is independent of any of the rankers.

\*/

class ImpressionBasedFatigueRanker[

Target <: HasClientContext with HasDisplayLocation with HasParams with HasWtfImpressions

](

fatigueFactor: Int,

statsReceiver: StatsReceiver)

extends Ranker[Target, CandidateUser] {

val name: String = this.getClass.getSimpleName

val stats = statsReceiver.scope("impression\_based\_fatigue\_ranker")

val droppedStats: MemoizingStatsReceiver = new MemoizingStatsReceiver(stats.scope("hard\_drops"))

val impressionStats: StatsReceiver = stats.scope("wtf\_impressions")

val noImpressionCounter: Counter = impressionStats.counter("no\_impressions")

val oldestImpressionStat: Stat = impressionStats.stat("oldest\_sec")

override def rank(target: Target, candidates: Seq[CandidateUser]): Stitch[Seq[CandidateUser]] = {

StatsUtil.profileStitch(

Stitch.value(rankCandidates(target, candidates)),

stats.scope("rank")

)

}

private def trackTimeSinceOldestImpression(impressions: Seq[WtfImpression]): Unit = {

val timeSinceOldest = Time.now - impressions.map(\_.latestTime).min

oldestImpressionStat.add(timeSinceOldest.inSeconds)

}

private def rankCandidates(

target: Target,

candidates: Seq[CandidateUser]

): Seq[CandidateUser] = {

target.wtfImpressions

.map { wtfImpressions =>

if (wtfImpressions.isEmpty) {

noImpressionCounter.incr()

candidates

} else {

val rankerIds =

candidates.flatMap(\_.scores.map(\_.scores.flatMap(\_.rankerId))).flatten.sorted.distinct

/\*\*

\* In below we create a Map from each CandidateUser's ID to a Map from each Ranker that

\* the user has a score for, and candidate's corresponding rank when candidates are sorted

\* by that Ranker (Only candidates who have this Ranker are considered for ranking).

\*/

val candidateRanks: Map[Long, Map[RankerId, Int]] = rankerIds

.flatMap { rankerId =>

// Candidates with no scores from this Ranker is first removed to calculate ranks.

val relatedCandidates =

candidates.filter(\_.scores.exists(\_.scores.exists(\_.rankerId.contains(rankerId))))

relatedCandidates

.sortBy(-\_.scores

.flatMap(\_.scores.find(\_.rankerId.contains(rankerId)).map(\_.value)).getOrElse(

0.0)).zipWithIndex.map {

case (candidate, rank) => (candidate.id, rankerId, rank)

}

}.groupBy(\_.\_1).map {

case (candidate, ranksForAllRankers) =>

(

candidate,

ranksForAllRankers.map { case (\_, rankerId, rank) => (rankerId, rank) }.toMap)

}

val idFatigueCountMap =

wtfImpressions.groupBy(\_.candidateId).mapValues(\_.map(\_.counts).sum)

trackTimeSinceOldestImpression(wtfImpressions)

val rankedCandidates: Seq[CandidateUser] = candidates

.map { candidate =>

val candidateImpressions = idFatigueCountMap.getOrElse(candidate.id, 0)

val fatiguedScores = candidate.scores.map { ss =>

ss.copy(scores = ss.scores.map { s =>

s.rankerId match {

// We set the new score as -rank after fatigue penalty is applied.

case Some(rankerId) =>

// If the candidate's ID is not in the candidate->ranks map, or there is no

// rank for this specific ranker and this candidate, we use maximum possible

// rank instead. Note that this indicates that there is a problem.

s.copy(value = -(candidateRanks

.getOrElse(candidate.id, Map()).getOrElse(rankerId, candidates.length) +

candidateImpressions \* fatigueFactor))

// In case a score exists without a RankerId, we pass on the score as is.

case None => s

}

})

}

candidate.copy(scores = fatiguedScores)

}.zipWithIndex.map {

// We re-rank candidates with their input ordering (which is done by the request-level

// ranker) and fatigue penalty.

case (candidate, inputRank) =>

val candidateImpressions = idFatigueCountMap.getOrElse(candidate.id, 0)

(candidate, inputRank + candidateImpressions \* fatigueFactor)

}.sortBy(\_.\_2).map(\_.\_1)

// Only populate ranking info when WTF impression info present

val scribeRankingInfo: Boolean =

target.params(ImpressionBasedFatigueRankerParams.ScribeRankingInfoInFatigueRanker)

if (scribeRankingInfo) Utils.addRankingInfo(rankedCandidates, name) else rankedCandidates

}

}.getOrElse(candidates) // no reranking/filtering when wtf impressions not present

}

}

object ImpressionBasedFatigueRanker {

val DefaultFatigueFactor = 5

def build[

Target <: HasClientContext with HasDisplayLocation with HasParams with HasWtfImpressions

](

baseStatsReceiver: StatsReceiver,

fatigueFactor: Int = DefaultFatigueFactor

): ImpressionBasedFatigueRanker[Target] =

new ImpressionBasedFatigueRanker(fatigueFactor, baseStatsReceiver)

}