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

import com.google.common.annotations.VisibleForTesting

import com.google.inject.Inject

import com.google.inject.Singleton

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.rankers.common.RankerId

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

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

@Singleton

class InterleaveRanker[Target <: HasParams] @Inject() (

statsReceiver: StatsReceiver)

extends Ranker[Target, CandidateUser] {

val name: String = this.getClass.getSimpleName

private val stats = statsReceiver.scope("interleave\_ranker")

private val inputStats = stats.scope("input")

private val interleavingStats = stats.scope("interleave")

override def rank(

target: Target,

candidates: Seq[CandidateUser]

): Stitch[Seq[CandidateUser]] = {

StatsUtil.profileStitch(

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

stats.scope("rank")

)

}

private def rankCandidates(

target: Target,

candidates: Seq[CandidateUser]

): Seq[CandidateUser] = {

/\*\*

\* By this stage, all valid candidates should have:

\* 1. Their Scores field populated.

\* 2. Their selectedRankerId set.

\* 3. Have a score associated to their selectedRankerId.

\* If there is any candidate that doesn't meet the conditions above, there is a problem in one

\* of the previous rankers. Since no new scoring is done in this ranker, we simply remove them.

\*/

val validCandidates =

candidates.filter { c =>

c.scores.isDefined &&

c.scores.exists(\_.selectedRankerId.isDefined) &&

getCandidateScoreByRankerId(c, c.scores.flatMap(\_.selectedRankerId)).isDefined

}

// To monitor the percentage of valid candidates, as defined above, we track the following:

inputStats.counter("candidates\_with\_no\_scores").incr(candidates.count(\_.scores.isEmpty))

inputStats

.counter("candidates\_with\_no\_selected\_ranker").incr(candidates.count { c =>

c.scores.isEmpty || c.scores.exists(\_.selectedRankerId.isEmpty)

})

inputStats

.counter("candidates\_with\_no\_score\_for\_selected\_ranker").incr(candidates.count { c =>

c.scores.isEmpty ||

c.scores.exists(\_.selectedRankerId.isEmpty) ||

getCandidateScoreByRankerId(c, c.scores.flatMap(\_.selectedRankerId)).isEmpty

})

inputStats.counter("total\_num\_candidates").incr(candidates.length)

inputStats.counter("total\_valid\_candidates").incr(validCandidates.length)

// We only count rankerIds from those candidates who are valid to exclude those candidates with

// a valid selectedRankerId that don't have an associated score for it.

val rankerIds = validCandidates.flatMap(\_.scores.flatMap(\_.selectedRankerId)).sorted.distinct

rankerIds.foreach { rankerId =>

inputStats

.counter(s"valid\_scores\_for\_${rankerId.toString}").incr(

candidates.count(getCandidateScoreByRankerId(\_, Some(rankerId)).isDefined))

inputStats.counter(s"total\_candidates\_for\_${rankerId.toString}").incr(candidates.length)

}

inputStats.counter(s"num\_ranker\_ids=${rankerIds.length}").incr()

val scribeRankingInfo: Boolean =

target.params(InterleaveRankerParams.ScribeRankingInfoInInterleaveRanker)

if (rankerIds.length <= 1)

// In the case of "Number of RankerIds = 0", we pass on the candidates even though there is

// a problem in a previous ranker that provided the scores.

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

else

if (scribeRankingInfo)

Utils.addRankingInfo(interleaveCandidates(validCandidates, rankerIds), name)

else interleaveCandidates(validCandidates, rankerIds)

}

@VisibleForTesting

private[interleave\_ranker] def interleaveCandidates(

candidates: Seq[CandidateUser],

rankerIds: Seq[RankerId.RankerId]

): Seq[CandidateUser] = {

val candidatesWithRank = rankerIds

.flatMap { ranker =>

candidates

// We first sort all candidates using this ranker.

.sortBy(-getCandidateScoreByRankerId(\_, Some(ranker)).getOrElse(Double.MinValue))

.zipWithIndex.filter(

// but only hold those candidates whose selected ranker is this ranker.

// These ranks will be forced in the final ordering.

\_.\_1.scores.flatMap(\_.selectedRankerId).contains(ranker))

}

// Only candidates who have isInProducerScoringExperiment set to true will have their position enforced. We

// separate candidates into two groups: (1) Production and (2) Experiment.

val (expCandidates, prodCandidates) =

candidatesWithRank.partition(\_.\_1.scores.exists(\_.isInProducerScoringExperiment))

// We resolve (potential) conflicts between the enforced ranks of experimental models.

val expCandidatesFinalPos = resolveConflicts(expCandidates)

// Retrieve non-occupied positions and assign them to candidates who use production ranker.

val occupiedPos = expCandidatesFinalPos.map(\_.\_2).toSet

val prodCandidatesFinalPos =

prodCandidates

.map(\_.\_1).zip(

candidates.indices.filterNot(occupiedPos.contains).sorted.take(prodCandidates.length))

// Merge the two groups and sort them by their corresponding positions.

val finalCandidates = (prodCandidatesFinalPos ++ expCandidatesFinalPos).sortBy(\_.\_2).map(\_.\_1)

// We count the presence of each ranker in the top-3 final positions.

finalCandidates.zip(0 until 3).foreach {

case (c, r) =>

// We only do so for candidates that are in a producer-side experiment.

if (c.scores.exists(\_.isInProducerScoringExperiment))

c.scores.flatMap(\_.selectedRankerId).map(\_.toString).foreach { rankerName =>

interleavingStats

.counter(s"num\_final\_position\_${r}\_$rankerName")

.incr()

}

}

finalCandidates

}

@VisibleForTesting

private[interleave\_ranker] def resolveConflicts(

candidatesWithRank: Seq[(CandidateUser, Int)]

): Seq[(CandidateUser, Int)] = {

// The following two metrics will allow us to calculate the rate of conflicts occurring.

// Example: If overall there are 10 producers in different bucketing experiments, and 3 of them

// are assigned to the same position. The rate would be 3/10, 30%.

val numCandidatesWithConflicts = interleavingStats.counter("candidates\_with\_conflict")

val numCandidatesNoConflicts = interleavingStats.counter("candidates\_without\_conflict")

val candidatesGroupedByRank = candidatesWithRank.groupBy(\_.\_2).toSeq.sortBy(\_.\_1).map {

case (rank, candidatesWithRank) => (rank, candidatesWithRank.map(\_.\_1))

}

candidatesGroupedByRank.foldLeft(Seq[(CandidateUser, Int)]()) { (upToHere, nextGroup) =>

val (rank, candidates) = nextGroup

if (candidates.length > 1)

numCandidatesWithConflicts.incr(candidates.length)

else

numCandidatesNoConflicts.incr()

// We use the position after the last-assigned candidate as a starting point, or 0 otherwise.

// If candidates' position is after this "starting point", we enforce that position instead.

val minAvailableIndex = scala.math.max(upToHere.lastOption.map(\_.\_2).getOrElse(-1) + 1, rank)

val enforcedPos =

(minAvailableIndex until minAvailableIndex + candidates.length).toList

val shuffledEnforcedPos =

if (candidates.length > 1) scala.util.Random.shuffle(enforcedPos) else enforcedPos

if (shuffledEnforcedPos.length > 1) {

candidates.zip(shuffledEnforcedPos).sortBy(\_.\_2).map(\_.\_1).zipWithIndex.foreach {

case (c, r) =>

c.scores.flatMap(\_.selectedRankerId).map(\_.toString).foreach { rankerName =>

// For each ranker, we count the total number of times it has been in a conflict.

interleavingStats

.counter(s"num\_${shuffledEnforcedPos.length}-way\_conflicts\_$rankerName")

.incr()

// We also count the positions each of the rankers have fallen randomly into. In any

// experiment this should converge to uniform distribution given enough occurrences.

// Note that the position here is relative to the other candidates in the conflict and

// not the overall position of each candidate.

interleavingStats

.counter(

s"num\_position\_${r}\_after\_${shuffledEnforcedPos.length}-way\_conflict\_$rankerName")

.incr()

}

}

}

upToHere ++ candidates.zip(shuffledEnforcedPos).sortBy(\_.\_2)

}

}

@VisibleForTesting

private[interleave\_ranker] def getCandidateScoreByRankerId(

candidate: CandidateUser,

rankerIdOpt: Option[RankerId.RankerId]

): Option[Double] = {

rankerIdOpt match {

case None => None

case Some(rankerId) =>

candidate.scores.flatMap {

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

}

}

}

}