package com.twitter.follow\_recommendations.common.rankers.first\_n\_ranker

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.models.CandidateUser

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

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

import com.twitter.product\_mixer.core.model.common.identifier.CandidateSourceIdentifier

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

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

/\*\*

\* This class is meant to filter candidates between stages of our ranker by taking the first N

\* candidates, merging any candidate source information for candidates with multiple entries.

\* To allow us to chain this truncation operation any number of times sequentially within the main

\* ranking builder, we abstract the truncation as a separate Ranker

\*/

@Singleton

class FirstNRanker[Target <: HasClientContext with HasParams with HasQualityFactor] @Inject() (

stats: StatsReceiver)

extends Ranker[Target, CandidateUser] {

val name: String = this.getClass.getSimpleName

private val baseStats = stats.scope("first\_n\_ranker")

val scaledDownByQualityFactorCounter =

baseStats.counter("scaled\_down\_by\_quality\_factor")

private val mergeStat = baseStats.scope("merged\_candidates")

private val mergeStat2 = mergeStat.counter("2")

private val mergeStat3 = mergeStat.counter("3")

private val mergeStat4 = mergeStat.counter("4+")

private val candidateSizeStats = baseStats.scope("candidate\_size")

private case class CandidateSourceScore(

candidateId: Long,

sourceId: CandidateSourceIdentifier,

score: Option[Double])

/\*\*

\* Adds the rank of each candidate based on the primary candidate source's score.

\* In the event where the provided ordering of candidates do not align with the score,

\* we will respect the score, since the ordering might have been mixed up due to other previous

\* steps like the shuffleFn in the `WeightedCandidateSourceRanker`.

\* @param candidates ordered list of candidates

\* @return same ordered list of candidates, but with the rank information appended

\*/

def addRank(candidates: Seq[CandidateUser]): Seq[CandidateUser] = {

val candidateSourceRanks = for {

(sourceIdOpt, sourceCandidates) <- candidates.groupBy(\_.getPrimaryCandidateSource)

(candidate, rank) <- sourceCandidates.sortBy(-\_.score.getOrElse(0.0)).zipWithIndex

} yield {

(candidate, sourceIdOpt) -> rank

}

candidates.map { c =>

c.getPrimaryCandidateSource

.map { sourceId =>

val sourceRank = candidateSourceRanks((c, c.getPrimaryCandidateSource))

c.addCandidateSourceRanksMap(Map(sourceId -> sourceRank))

}.getOrElse(c)

}

}

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

val scaleDownFactor = Math.max(

target.qualityFactor.getOrElse(1.0d),

target.params(FirstNRankerParams.MinNumCandidatesScoredScaleDownFactor)

)

if (scaleDownFactor < 1.0d)

scaledDownByQualityFactorCounter.incr()

val n = (target.params(FirstNRankerParams.CandidatesToRank) \* scaleDownFactor).toInt

val scribeRankingInfo: Boolean =

target.params(FirstNRankerParams.ScribeRankingInfoInFirstNRanker)

candidateSizeStats.counter(s"n$n").incr()

val candidatesWithRank = addRank(candidates)

if (target.params(FirstNRankerParams.GroupDuplicateCandidates)) {

val groupedCandidates: Map[Long, Seq[CandidateUser]] = candidatesWithRank.groupBy(\_.id)

val topN = candidates

.map { c =>

merge(groupedCandidates(c.id))

}.distinct.take(n)

Stitch.value(if (scribeRankingInfo) Utils.addRankingInfo(topN, name) else topN)

} else {

Stitch.value(

if (scribeRankingInfo) Utils.addRankingInfo(candidatesWithRank, name).take(n)

else candidatesWithRank.take(n))

} // for efficiency, if don't need to deduplicate

}

/\*\*

\* we use the primary candidate source of the first entry, and aggregate all of the other entries'

\* candidate source scores into the first entry's candidateSourceScores

\* @param candidates list of candidates with the same id

\* @return a single merged candidate

\*/

private[first\_n\_ranker] def merge(candidates: Seq[CandidateUser]): CandidateUser = {

if (candidates.size == 1) {

candidates.head

} else {

candidates.size match {

case 2 => mergeStat2.incr()

case 3 => mergeStat3.incr()

case i if i >= 4 => mergeStat4.incr()

case \_ =>

}

val allSources = candidates.flatMap(\_.getCandidateSources).toMap

val allRanks = candidates.flatMap(\_.getCandidateRanks).toMap

candidates.head.addCandidateSourceScoresMap(allSources).addCandidateSourceRanksMap(allRanks)

}

}

}