package com.twitter.follow\_recommendations.common.rankers.weighted\_candidate\_source\_ranker

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

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

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

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

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

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

/\*\*

\* Candidate Ranker that mixes and ranks multiple candidate lists from different candidate sources with the

\* following steps:

\* 1) generate a ranked candidate list of each candidate source by sorting and shuffling the candidate list

\* of the algorithm.

\* 2) merge the ranked lists generated in 1) into a single list using weighted randomly sampling.

\* 3) If dedup is required, dedup the output from 2) by candidate id.

\*

\* @param basedRanker base ranker

\* @param shuffleFn the shuffle function that will be used to shuffle each algorithm's sorted candidate list.

\* @param dedup whether to remove duplicated candidates from the final output.

\*/

class WeightedCandidateSourceRanker[Target <: HasParams](

basedRanker: WeightedCandidateSourceBaseRanker[

CandidateSourceIdentifier,

CandidateUser

],

shuffleFn: Seq[CandidateUser] => Seq[CandidateUser],

dedup: Boolean)

extends Ranker[Target, CandidateUser] {

val name: String = this.getClass.getSimpleName

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

val scribeRankingInfo: Boolean =

target.params(WeightedCandidateSourceRankerParams.ScribeRankingInfoInWeightedRanker)

val rankedCands = rankCandidates(group(candidates))

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

}

private def group(

candidates: Seq[CandidateUser]

): Map[CandidateSourceIdentifier, Seq[CandidateUser]] = {

val flattened = for {

candidate <- candidates

identifier <- candidate.getPrimaryCandidateSource

} yield (identifier, candidate)

flattened.groupBy(\_.\_1).mapValues(\_.map(\_.\_2))

}

private def rankCandidates(

input: Map[CandidateSourceIdentifier, Seq[CandidateUser]]

): Seq[CandidateUser] = {

// Sort and shuffle candidates per candidate source.

// Note 1: Using map instead mapValue here since mapValue somehow caused infinite loop when used as part of Stream.

val sortAndShuffledCandidates = input.map {

case (source, candidates) =>

// Note 2: toList is required here since candidates is a view, and it will result in infinit loop when used as part of Stream.

// Note 3: there is no real sorting logic here, it assumes the input is already sorted by candidate sources

val sortedCandidates = candidates.toList

source -> shuffleFn(sortedCandidates).iterator

}

val rankedCandidates = basedRanker(sortAndShuffledCandidates)

if (dedup) DedupCandidates(rankedCandidates) else rankedCandidates

}

}

object WeightedCandidateSourceRanker {

def build[Target <: HasParams](

candidateSourceWeight: Map[CandidateSourceIdentifier, Double],

shuffleFn: Seq[CandidateUser] => Seq[CandidateUser] = identity,

dedup: Boolean = false,

randomSeed: Option[Long] = None

): WeightedCandidateSourceRanker[Target] = {

new WeightedCandidateSourceRanker(

new WeightedCandidateSourceBaseRanker(

candidateSourceWeight,

WeightMethod.WeightedRandomSampling,

randomSeed = randomSeed),

shuffleFn,

dedup

)

}

}

object WeightedCandidateSourceRankerWithoutRandomSampling {

def build[Target <: HasParams](

candidateSourceWeight: Map[CandidateSourceIdentifier, Double]

): WeightedCandidateSourceRanker[Target] = {

new WeightedCandidateSourceRanker(

new WeightedCandidateSourceBaseRanker(

candidateSourceWeight,

WeightMethod.WeightedRoundRobin,

randomSeed = None),

identity,

false,

)

}

}