package com.twitter.cr\_mixer.util

import com.twitter.cr\_mixer.model.Candidate

import com.twitter.cr\_mixer.model.CandidateGenerationInfo

import com.twitter.cr\_mixer.model.RankedCandidate

import com.twitter.cr\_mixer.model.SourceInfo

import com.twitter.cr\_mixer.thriftscala.SimilarityEngineType

import com.twitter.simclusters\_v2.common.TweetId

import scala.collection.mutable

import scala.collection.mutable.ArrayBuffer

object InterleaveUtil {

/\*\*

\* Interleaves candidates by iteratively taking one candidate from the 1st Seq and adding it to the result.

\* Once we take a candidate from a Seq, we move this Seq to the end of the queue to process,

\* and remove the candidate from that Seq.

\*

\* We keep a mutable.Set[TweetId] buffer to ensure there are no duplicates.

\*

\* @param candidates candidates assumed to be sorted by eventTime (latest event comes first)

\* @return interleaved candidates

\*/

def interleave[CandidateType <: Candidate](

candidates: Seq[Seq[CandidateType]]

): Seq[CandidateType] = {

// copy candidates into a mutable map so this method is thread-safe

val candidatesPerSequence = candidates.map { tweetCandidates =>

mutable.Queue() ++= tweetCandidates

}

val seen = mutable.Set[TweetId]()

val candidateSeqQueue = mutable.Queue() ++= candidatesPerSequence

val result = ArrayBuffer[CandidateType]()

while (candidateSeqQueue.nonEmpty) {

val candidatesQueue = candidateSeqQueue.head

if (candidatesQueue.nonEmpty) {

val candidate = candidatesQueue.dequeue()

val candidateTweetId = candidate.tweetId

val seenCandidate = seen.contains(candidateTweetId)

if (!seenCandidate) {

result += candidate

seen.add(candidate.tweetId)

candidateSeqQueue.enqueue(

candidateSeqQueue.dequeue()

) // move this Seq to end

}

} else {

candidateSeqQueue.dequeue() //finished processing this Seq

}

}

//convert result to immutable seq

result.toList

}

/\*\*

\* Interleaves candidates by iteratively

\* 1. Checking weight to see if enough accumulation has occurred to sample from

\* 2. If yes, taking one candidate from the the Seq and adding it to the result.

\* 3. Move this Seq to the end of the queue to process (and remove the candidate from that Seq if

\* we sampled it from step 2).

\*

\* We keep count of the iterations to prevent infinite loops.

\* We keep a mutable.Set[TweetId] buffer to ensure there are no duplicates.

\*

\* @param candidatesAndWeight candidates assumed to be sorted by eventTime (latest event comes first),

\* along with sampling weights to help prioritize important groups.

\* @param maxWeightAdjustments Maximum number of iterations to account for weighting before

\* defaulting to uniform interleaving.

\* @return interleaved candidates

\*/

def weightedInterleave[CandidateType <: Candidate](

candidatesAndWeight: Seq[(Seq[CandidateType], Double)],

maxWeightAdjustments: Int = 0

): Seq[CandidateType] = {

// Set to avoid numerical issues around 1.0

val min\_weight = 1 - 1e-30

// copy candidates into a mutable map so this method is thread-safe

// adds a counter to use towards sampling

val candidatesAndWeightsPerSequence: Seq[

(mutable.Queue[CandidateType], InterleaveWeights)

] =

candidatesAndWeight.map { candidatesAndWeight =>

(mutable.Queue() ++= candidatesAndWeight.\_1, InterleaveWeights(candidatesAndWeight.\_2, 0.0))

}

val seen: mutable.Set[TweetId] = mutable.Set[TweetId]()

val candidateSeqQueue: mutable.Queue[(mutable.Queue[CandidateType], InterleaveWeights)] =

mutable.Queue() ++= candidatesAndWeightsPerSequence

val result: ArrayBuffer[CandidateType] = ArrayBuffer[CandidateType]()

var number\_iterations: Int = 0

while (candidateSeqQueue.nonEmpty) {

val (candidatesQueue, currentWeights) = candidateSeqQueue.head

if (candidatesQueue.nonEmpty) {

// Confirm weighting scheme

currentWeights.summed\_weight += currentWeights.weight

number\_iterations += 1

if (currentWeights.summed\_weight >= min\_weight || number\_iterations >= maxWeightAdjustments) {

// If we sample, then adjust the counter

currentWeights.summed\_weight -= 1.0

val candidate = candidatesQueue.dequeue()

val candidateTweetId = candidate.tweetId

val seenCandidate = seen.contains(candidateTweetId)

if (!seenCandidate) {

result += candidate

seen.add(candidate.tweetId)

candidateSeqQueue.enqueue(candidateSeqQueue.dequeue()) // move this Seq to end

}

} else {

candidateSeqQueue.enqueue(candidateSeqQueue.dequeue()) // move this Seq to end

}

} else {

candidateSeqQueue.dequeue() //finished processing this Seq

}

}

//convert result to immutable seq

result.toList

}

def buildCandidatesKeyByCGInfo(

candidates: Seq[RankedCandidate],

): Seq[Seq[RankedCandidate]] = {

// To accommodate the re-grouping in InterleaveRanker

// In InterleaveBlender, we have already abandoned the grouping keys, and use Seq[Seq[]] to do interleave

// Since that we build the candidateSeq with groupingKey, we can guarantee there is no empty candidateSeq

val candidateSeqKeyByCG =

candidates.groupBy(candidate => GroupingKey.toGroupingKey(candidate.reasonChosen))

candidateSeqKeyByCG.map {

case (groupingKey, candidateSeq) =>

candidateSeq.sortBy(-\_.predictionScore)

}.toSeq

}

}

case class GroupingKey(

sourceInfoOpt: Option[SourceInfo],

similarityEngineType: SimilarityEngineType,

modelId: Option[String]) {}

object GroupingKey {

def toGroupingKey(candidateGenerationInfo: CandidateGenerationInfo): GroupingKey = {

GroupingKey(

candidateGenerationInfo.sourceInfoOpt,

candidateGenerationInfo.similarityEngineInfo.similarityEngineType,

candidateGenerationInfo.similarityEngineInfo.modelId

)

}

}

case class InterleaveWeights(weight: Double, var summed\_weight: Double)