package com.twitter.cr\_mixer.util

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

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

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

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

import com.twitter.cr\_mixer.param.BlenderParams.BlendGroupingMethodEnum

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

import com.twitter.simclusters\_v2.thriftscala.InternalId

object CountWeightedInterleaveUtil {

/\*\*

\* Grouping key for interleaving candidates

\*

\* @param sourceInfoOpt optional SourceInfo, containing the source information

\* @param similarityEngineTypeOpt optional SimilarityEngineType, containing similarity engine

\* information

\* @param modelIdOpt optional modelId, containing the model ID

\* @param authorIdOpt optional authorId, containing the tweet author ID

\* @param groupIdOpt optional groupId, containing the ID corresponding to the blending group

\*/

case class GroupingKey(

sourceInfoOpt: Option[SourceInfo],

similarityEngineTypeOpt: Option[SimilarityEngineType],

modelIdOpt: Option[String],

authorIdOpt: Option[Long],

groupIdOpt: Option[Int])

/\*\*

\* Converts candidates to grouping key based upon the feature that we interleave with.

\*/

def toGroupingKey[CandidateType <: Candidate](

candidate: CandidateType,

interleaveFeature: Option[BlendGroupingMethodEnum.Value],

groupId: Option[Int],

): GroupingKey = {

val grouping: GroupingKey = candidate match {

case c: RankedCandidate =>

interleaveFeature.getOrElse(BlendGroupingMethodEnum.SourceKeyDefault) match {

case BlendGroupingMethodEnum.SourceKeyDefault =>

GroupingKey(

sourceInfoOpt = c.reasonChosen.sourceInfoOpt,

similarityEngineTypeOpt =

Some(c.reasonChosen.similarityEngineInfo.similarityEngineType),

modelIdOpt = c.reasonChosen.similarityEngineInfo.modelId,

authorIdOpt = None,

groupIdOpt = groupId

)

// Some candidate sources don't have a sourceType, so it defaults to similarityEngine

case BlendGroupingMethodEnum.SourceTypeSimilarityEngine =>

val sourceInfoOpt = c.reasonChosen.sourceInfoOpt.map(\_.sourceType).map { sourceType =>

SourceInfo(

sourceType = sourceType,

internalId = InternalId.UserId(0),

sourceEventTime = None)

}

GroupingKey(

sourceInfoOpt = sourceInfoOpt,

similarityEngineTypeOpt =

Some(c.reasonChosen.similarityEngineInfo.similarityEngineType),

modelIdOpt = c.reasonChosen.similarityEngineInfo.modelId,

authorIdOpt = None,

groupIdOpt = groupId

)

case BlendGroupingMethodEnum.AuthorId =>

GroupingKey(

sourceInfoOpt = None,

similarityEngineTypeOpt = None,

modelIdOpt = None,

authorIdOpt = Some(c.tweetInfo.authorId),

groupIdOpt = groupId

)

case \_ =>

throw new UnsupportedOperationException(

s"Unsupported interleave feature: $interleaveFeature")

}

case \_ =>

GroupingKey(

sourceInfoOpt = None,

similarityEngineTypeOpt = None,

modelIdOpt = None,

authorIdOpt = None,

groupIdOpt = groupId

)

}

grouping

}

/\*\*

\* Rather than manually calculating and maintaining the weights to rank with, we instead

\* calculate the weights on the fly, based upon the frequencies of the candidates within each

\* group. To ensure that diversity of the feature is maintained, we additionally employ a

\* 'shrinkage' parameter which enforces more diversity by moving the weights closer to uniformity.

\* More details are available at go/weighted-interleave.

\*

\* @param candidateSeqKeyByFeature candidate to key.

\* @param rankerWeightShrinkage value between [0, 1] with 1 being complete uniformity.

\* @return Interleaving weights keyed by feature.

\*/

private def calculateWeightsKeyByFeature[CandidateType <: Candidate](

candidateSeqKeyByFeature: Map[GroupingKey, Seq[CandidateType]],

rankerWeightShrinkage: Double

): Map[GroupingKey, Double] = {

val maxNumberCandidates: Double = candidateSeqKeyByFeature.values

.map { candidates =>

candidates.size

}.max.toDouble

candidateSeqKeyByFeature.map {

case (featureKey: GroupingKey, candidateSeq: Seq[CandidateType]) =>

val observedWeight: Double = candidateSeq.size.toDouble / maxNumberCandidates

// How much to shrink empirical estimates to 1 (Default is to make all weights 1).

val finalWeight =

(1.0 - rankerWeightShrinkage) \* observedWeight + rankerWeightShrinkage \* 1.0

featureKey -> finalWeight

}

}

/\*\*

\* Builds out the groups and weights for weighted interleaving of the candidates.

\* More details are available at go/weighted-interleave.

\*

\* @param rankedCandidateSeq candidates to interleave.

\* @param rankerWeightShrinkage value between [0, 1] with 1 being complete uniformity.

\* @return Candidates grouped by feature key and with calculated interleaving weights.

\*/

def buildRankedCandidatesWithWeightKeyByFeature(

rankedCandidateSeq: Seq[RankedCandidate],

rankerWeightShrinkage: Double,

interleaveFeature: BlendGroupingMethodEnum.Value

): Seq[(Seq[RankedCandidate], Double)] = {

// 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 candidateSeqKeyByFeature: Map[GroupingKey, Seq[RankedCandidate]] =

rankedCandidateSeq.groupBy { candidate: RankedCandidate =>

toGroupingKey(candidate, Some(interleaveFeature), None)

}

// These weights [0, 1] are used to do weighted interleaving

// The default value of 1.0 ensures the group is always sampled.

val candidateWeightsKeyByFeature: Map[GroupingKey, Double] =

calculateWeightsKeyByFeature(candidateSeqKeyByFeature, rankerWeightShrinkage)

candidateSeqKeyByFeature.map {

case (groupingKey: GroupingKey, candidateSeq: Seq[RankedCandidate]) =>

Tuple2(

candidateSeq.sortBy(-\_.predictionScore),

candidateWeightsKeyByFeature.getOrElse(groupingKey, 1.0))

}.toSeq

}

/\*\*

\* Takes current grouping (as implied by the outer Seq) and computes blending weights.

\*

\* @param initialCandidatesSeqSeq grouped candidates to interleave.

\* @param rankerWeightShrinkage value between [0, 1] with 1 being complete uniformity.

\* @return Grouped candidates with calculated interleaving weights.

\*/

def buildInitialCandidatesWithWeightKeyByFeature(

initialCandidatesSeqSeq: Seq[Seq[InitialCandidate]],

rankerWeightShrinkage: Double,

): Seq[(Seq[InitialCandidate], Double)] = {

val candidateSeqKeyByFeature: Map[GroupingKey, Seq[InitialCandidate]] =

initialCandidatesSeqSeq.zipWithIndex.map(\_.swap).toMap.map {

case (groupId: Int, initialCandidatesSeq: Seq[InitialCandidate]) =>

toGroupingKey(initialCandidatesSeq.head, None, Some(groupId)) -> initialCandidatesSeq

}

// These weights [0, 1] are used to do weighted interleaving

// The default value of 1.0 ensures the group is always sampled.

val candidateWeightsKeyByFeature =

calculateWeightsKeyByFeature(candidateSeqKeyByFeature, rankerWeightShrinkage)

candidateSeqKeyByFeature.map {

case (groupingKey: GroupingKey, candidateSeq: Seq[InitialCandidate]) =>

Tuple2(candidateSeq, candidateWeightsKeyByFeature.getOrElse(groupingKey, 1.0))

}.toSeq

}

}