package com.twitter.cr\_mixer.blender

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

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

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

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

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.timelines.configapi.Params

import com.twitter.util.Future

import com.twitter.util.Time

import javax.inject.Inject

case class ContentSignalBlender @Inject() (globalStats: StatsReceiver) {

private val name: String = this.getClass.getCanonicalName

private val stats: StatsReceiver = globalStats.scope(name)

/\*\*

\* Exposes multiple types of sorting relying only on Content Based signals

\* Candidate Recency, Random, FavoriteCount and finally Standardized, which standardizes the scores

\* that come from the active SimilarityEngine and then sort on the standardized scores.

\*/

def blend(

params: Params,

inputCandidates: Seq[Seq[InitialCandidate]],

): Future[Seq[BlendedCandidate]] = {

// Filter out empty candidate sequence

val candidates = inputCandidates.filter(\_.nonEmpty)

val sortedCandidates = params(BlenderParams.ContentBlenderTypeSortingAlgorithmParam) match {

case BlenderParams.ContentBasedSortingAlgorithmEnum.CandidateRecency =>

candidates.flatten.sortBy(c => getSnowflakeTimeStamp(c.tweetId)).reverse

case BlenderParams.ContentBasedSortingAlgorithmEnum.RandomSorting =>

candidates.flatten.sortBy(\_ => scala.util.Random.nextDouble())

case BlenderParams.ContentBasedSortingAlgorithmEnum.FavoriteCount =>

candidates.flatten.sortBy(-\_.tweetInfo.favCount)

case BlenderParams.ContentBasedSortingAlgorithmEnum.SimilarityToSignalSorting =>

standardizeAndSortByScore(flattenAndGroupByEngineTypeOrFirstContribEngine(candidates))

case \_ =>

candidates.flatten.sortBy(-\_.tweetInfo.favCount)

}

stats.stat("candidates").add(sortedCandidates.size)

val blendedCandidates =

BlendedCandidatesBuilder.build(inputCandidates, removeDuplicates(sortedCandidates))

Future.value(blendedCandidates)

}

private def removeDuplicates(candidates: Seq[InitialCandidate]): Seq[InitialCandidate] = {

val seen = collection.mutable.Set.empty[Long]

candidates.filter { c =>

if (seen.contains(c.tweetId)) {

false

} else {

seen += c.tweetId

true

}

}

}

private def groupByEngineTypeOrFirstContribEngine(

candidates: Seq[InitialCandidate]

): Map[SimilarityEngineType, Seq[InitialCandidate]] = {

val grouped = candidates.groupBy { candidate =>

val contrib = candidate.candidateGenerationInfo.contributingSimilarityEngines

if (contrib.nonEmpty) {

contrib.head.similarityEngineType

} else {

candidate.candidateGenerationInfo.similarityEngineInfo.similarityEngineType

}

}

grouped

}

private def flattenAndGroupByEngineTypeOrFirstContribEngine(

candidates: Seq[Seq[InitialCandidate]]

): Seq[Seq[InitialCandidate]] = {

val flat = candidates.flatten

val grouped = groupByEngineTypeOrFirstContribEngine(flat)

grouped.values.toSeq

}

private def standardizeAndSortByScore(

candidates: Seq[Seq[InitialCandidate]]

): Seq[InitialCandidate] = {

candidates

.map { innerSeq =>

val meanScore = innerSeq

.map(c => c.candidateGenerationInfo.similarityEngineInfo.score.getOrElse(0.0))

.sum / innerSeq.length

val stdDev = scala.math

.sqrt(

innerSeq

.map(c => c.candidateGenerationInfo.similarityEngineInfo.score.getOrElse(0.0))

.map(a => a - meanScore)

.map(a => a \* a)

.sum / innerSeq.length)

innerSeq

.map(c =>

(

c,

c.candidateGenerationInfo.similarityEngineInfo.score

.map { score =>

if (stdDev != 0) (score - meanScore) / stdDev

else 0.0

}

.getOrElse(0.0)))

}.flatten.sortBy { case (\_, standardizedScore) => -standardizedScore }

.map { case (candidate, \_) => candidate }

}

private def getSnowflakeTimeStamp(tweetId: Long): Time = {

val isSnowflake = SnowflakeId.isSnowflakeId(tweetId)

if (isSnowflake) {

SnowflakeId(tweetId).time

} else {

Time.fromMilliseconds(0L)

}

}

}