package com.twitter.cr\_mixer.candidate\_generation

import com.twitter.cr\_mixer.blender.AdsBlender

import com.twitter.cr\_mixer.logging.AdsRecommendationsScribeLogger

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

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

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

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

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

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

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

import com.twitter.cr\_mixer.source\_signal.RealGraphInSourceGraphFetcher

import com.twitter.cr\_mixer.source\_signal.SourceFetcher.FetcherQuery

import com.twitter.cr\_mixer.source\_signal.UssSourceSignalFetcher

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.util.StatsUtil

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

import com.twitter.util.Future

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

class AdsCandidateGenerator @Inject() (

ussSourceSignalFetcher: UssSourceSignalFetcher,

realGraphInSourceGraphFetcher: RealGraphInSourceGraphFetcher,

adsCandidateSourceRouter: AdsCandidateSourcesRouter,

adsBlender: AdsBlender,

scribeLogger: AdsRecommendationsScribeLogger,

globalStats: StatsReceiver) {

private val stats: StatsReceiver = globalStats.scope(this.getClass.getCanonicalName)

private val fetchSourcesStats = stats.scope("fetchSources")

private val fetchRealGraphSeedsStats = stats.scope("fetchRealGraphSeeds")

private val fetchCandidatesStats = stats.scope("fetchCandidates")

private val interleaveStats = stats.scope("interleave")

private val rankStats = stats.scope("rank")

def get(query: AdsCandidateGeneratorQuery): Future[Seq[RankedAdsCandidate]] = {

val allStats = stats.scope("all")

val perProductStats = stats.scope("perProduct", query.product.toString)

StatsUtil.trackItemsStats(allStats) {

StatsUtil.trackItemsStats(perProductStats) {

for {

// fetch source signals

sourceSignals <- StatsUtil.trackBlockStats(fetchSourcesStats) {

fetchSources(query)

}

realGraphSeeds <- StatsUtil.trackItemMapStats(fetchRealGraphSeedsStats) {

fetchSeeds(query)

}

// get initial candidates from similarity engines

// hydrate lineItemInfo and filter out non active ads

initialCandidates <- StatsUtil.trackBlockStats(fetchCandidatesStats) {

fetchCandidates(query, sourceSignals, realGraphSeeds)

}

// blend candidates

blendedCandidates <- StatsUtil.trackItemsStats(interleaveStats) {

interleave(initialCandidates)

}

rankedCandidates <- StatsUtil.trackItemsStats(rankStats) {

rank(

blendedCandidates,

query.params(AdsParams.EnableScoreBoost),

query.params(AdsParams.AdsCandidateGenerationScoreBoostFactor),

rankStats)

}

} yield {

rankedCandidates.take(query.maxNumResults)

}

}

}

}

def fetchSources(

query: AdsCandidateGeneratorQuery

): Future[Set[SourceInfo]] = {

val fetcherQuery =

FetcherQuery(query.userId, query.product, query.userState, query.params)

ussSourceSignalFetcher.get(fetcherQuery).map(\_.getOrElse(Seq.empty).toSet)

}

private def fetchCandidates(

query: AdsCandidateGeneratorQuery,

sourceSignals: Set[SourceInfo],

realGraphSeeds: Map[UserId, Double]

): Future[Seq[Seq[InitialAdsCandidate]]] = {

scribeLogger.scribeInitialAdsCandidates(

query,

adsCandidateSourceRouter

.fetchCandidates(query.userId, sourceSignals, realGraphSeeds, query.params),

query.params(AdsParams.EnableScribe)

)

}

private def fetchSeeds(

query: AdsCandidateGeneratorQuery

): Future[Map[UserId, Double]] = {

if (query.params(ConsumersBasedUserAdGraphParams.EnableSourceParam)) {

realGraphInSourceGraphFetcher

.get(FetcherQuery(query.userId, query.product, query.userState, query.params))

.map(\_.map(\_.seedWithScores).getOrElse(Map.empty))

} else Future.value(Map.empty[UserId, Double])

}

private def interleave(

candidates: Seq[Seq[InitialAdsCandidate]]

): Future[Seq[BlendedAdsCandidate]] = {

adsBlender

.blend(candidates)

}

private def rank(

candidates: Seq[BlendedAdsCandidate],

enableScoreBoost: Boolean,

scoreBoostFactor: Double,

statsReceiver: StatsReceiver,

): Future[Seq[RankedAdsCandidate]] = {

val candidateSize = candidates.size

val rankedCandidates = candidates.zipWithIndex.map {

case (candidate, index) =>

val score = 0.5 + 0.5 \* ((candidateSize - index).toDouble / candidateSize)

val boostedScore = if (enableScoreBoost) {

statsReceiver.stat("boostedScore").add((100.0 \* score \* scoreBoostFactor).toFloat)

score \* scoreBoostFactor

} else {

statsReceiver.stat("score").add((100.0 \* score).toFloat)

score

}

candidate.toRankedAdsCandidate(boostedScore)

}

Future.value(rankedCandidates)

}

}