package com.twitter.follow\_recommendations.common.candidate\_sources.base

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.util.DefaultTimer

import com.twitter.follow\_recommendations.common.candidate\_sources.base.RealGraphExpansionRepository.DefaultScore

import com.twitter.follow\_recommendations.common.candidate\_sources.base.RealGraphExpansionRepository.MaxNumIntermediateNodesToKeep

import com.twitter.follow\_recommendations.common.candidate\_sources.base.RealGraphExpansionRepository.FirstDegreeCandidatesTimeout

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

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

import com.twitter.onboarding.relevance.features.ymbii.ExpansionCandidateScores

import com.twitter.onboarding.relevance.features.ymbii.RawYMBIICandidateFeatures

import com.twitter.onboarding.relevance.store.thriftscala.CandidatesFollowedV1

import com.twitter.product\_mixer.core.functional\_component.candidate\_source.CandidateSource

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

import com.twitter.stitch.Stitch

import com.twitter.strato.client.Fetcher

import com.twitter.util.Duration

import scala.collection.immutable

import scala.util.control.NonFatal

private final case class InterestExpansionCandidate(

userID: Long,

score: Double,

features: RawYMBIICandidateFeatures)

abstract class RealGraphExpansionRepository[Request](

realgraphExpansionStore: Fetcher[

Long,

Unit,

CandidatesFollowedV1

],

override val identifier: CandidateSourceIdentifier,

statsReceiver: StatsReceiver = NullStatsReceiver,

maxUnderlyingCandidatesToQuery: Int = 50,

maxCandidatesToReturn: Int = 40,

overrideUnderlyingTimeout: Option[Duration] = None,

appendSocialProof: Boolean = false)

extends CandidateSource[

Request,

CandidateUser

] {

val underlyingCandidateSource: Seq[

CandidateSource[

Request,

CandidateUser

]

]

private val stats = statsReceiver.scope(this.getClass.getSimpleName).scope(identifier.name)

private val underlyingCandidateSourceFailureStats =

stats.scope("underlying\_candidate\_source\_failure")

def apply(

request: Request,

): Stitch[Seq[CandidateUser]] = {

val candidatesFromUnderlyingSourcesStitch: Seq[Stitch[Seq[CandidateUser]]] =

underlyingCandidateSource.map { candidateSource =>

candidateSource

.apply(request)

.within(overrideUnderlyingTimeout.getOrElse(FirstDegreeCandidatesTimeout))(

DefaultTimer

)

.handle {

case NonFatal(e) =>

underlyingCandidateSourceFailureStats

.counter(candidateSource.identifier.name, e.getClass.getSimpleName).incr()

Seq.empty

}

}

for {

underlyingCandidatesFromEachAlgo <- Stitch.collect(candidatesFromUnderlyingSourcesStitch)

// The first algorithm in the list has the highest priority. Depending on if its not

// populated, fall back to other algorithms. Once a particular algorithm is chosen, only

// take the top few candidates from the underlying store for expansion.

underlyingCandidatesTuple =

underlyingCandidatesFromEachAlgo

.zip(underlyingCandidateSource)

.find(\_.\_1.nonEmpty)

underlyingAlgorithmUsed: Option[CandidateSourceIdentifier] = underlyingCandidatesTuple.map {

case (\_, candidateSource) => candidateSource.identifier

}

// Take maxUnderlyingCandidatesToQuery to query realgraphExpansionStore

underlyingCandidates =

underlyingCandidatesTuple

.map {

case (candidates, candidateSource) =>

stats

.scope("underlyingAlgorithmUsedScope").counter(

candidateSource.identifier.name).incr()

candidates

}

.getOrElse(Seq.empty)

.sortBy(\_.score.getOrElse(DefaultScore))(Ordering.Double.reverse)

.take(maxUnderlyingCandidatesToQuery)

underlyingCandidateMap: Map[Long, Double] = underlyingCandidates.map { candidate =>

(candidate.id, candidate.score.getOrElse(DefaultScore))

}.toMap

expansionCandidates <-

Stitch

.traverse(underlyingCandidateMap.keySet.toSeq) { candidateId =>

Stitch.join(

Stitch.value(candidateId),

realgraphExpansionStore.fetch(candidateId).map(\_.v))

}.map(\_.toMap)

rerankedCandidates: Seq[InterestExpansionCandidate] =

rerankCandidateExpansions(underlyingCandidateMap, expansionCandidates)

rerankedCandidatesFiltered = rerankedCandidates.take(maxCandidatesToReturn)

} yield {

rerankedCandidatesFiltered.map { candidate =>

val socialProofReason = if (appendSocialProof) {

val socialProofIds = candidate.features.expansionCandidateScores

.map(\_.intermediateCandidateId)

Some(

Reason(Some(

AccountProof(followProof = Some(FollowProof(socialProofIds, socialProofIds.size))))))

} else {

None

}

CandidateUser(

id = candidate.userID,

score = Some(candidate.score),

reason = socialProofReason,

userCandidateSourceDetails = Some(

UserCandidateSourceDetails(

primaryCandidateSource = Some(identifier),

candidateSourceFeatures = Map(identifier -> Seq(candidate.features))

))

).addAddressBookMetadataIfAvailable(underlyingAlgorithmUsed.toSeq)

}

}

}

/\*\*

\* Expands underlying candidates, returning them in sorted order.

\*

\* @param underlyingCandidatesMap A map from underlying candidate id to score

\* @param expansionCandidateMap A map from underlying candidate id to optional expansion candidates

\* @return A sorted sequence of expansion candidates and associated scores

\*/

private def rerankCandidateExpansions(

underlyingCandidatesMap: Map[Long, Double],

expansionCandidateMap: Map[Long, Option[CandidatesFollowedV1]]

): Seq[InterestExpansionCandidate] = {

// extract features

val candidates: Seq[(Long, ExpansionCandidateScores)] = for {

(underlyingCandidateId, underlyingCandidateScore) <- underlyingCandidatesMap.toSeq

expansionCandidates =

expansionCandidateMap

.get(underlyingCandidateId)

.flatten

.map(\_.candidatesFollowed)

.getOrElse(Seq.empty)

expansionCandidate <- expansionCandidates

} yield expansionCandidate.candidateID -> ExpansionCandidateScores(

underlyingCandidateId,

Some(underlyingCandidateScore),

Some(expansionCandidate.score)

)

// merge intermediate nodes for the same candidate

val dedupedCandidates: Seq[(Long, Seq[ExpansionCandidateScores])] =

candidates.groupBy(\_.\_1).mapValues(\_.map(\_.\_2).sortBy(\_.intermediateCandidateId)).toSeq

// score the candidate

val candidatesWithTotalScore: Seq[((Long, Seq[ExpansionCandidateScores]), Double)] =

dedupedCandidates.map { candidate: (Long, Seq[ExpansionCandidateScores]) =>

(

candidate,

candidate.\_2.map { ieScore: ExpansionCandidateScores =>

ieScore.scoreFromUserToIntermediateCandidate.getOrElse(DefaultScore) \*

ieScore.scoreFromIntermediateToExpansionCandidate.getOrElse(DefaultScore)

}.sum)

}

// sort candidate by score

for {

((candidate, edges), score) <- candidatesWithTotalScore.sortBy(\_.\_2)(Ordering[Double].reverse)

} yield InterestExpansionCandidate(

candidate,

score,

RawYMBIICandidateFeatures(

edges.size,

edges.take(MaxNumIntermediateNodesToKeep).to[immutable.Seq])

)

}

}

object RealGraphExpansionRepository {

private val FirstDegreeCandidatesTimeout: Duration = 250.milliseconds

private val MaxNumIntermediateNodesToKeep = 20

private val DefaultScore = 0.0d

}