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

import com.twitter.follow\_recommendations.common.candidate\_sources.base.SimilarUserExpanderParams.DefaultEnableImplicitEngagedExpansion

import com.twitter.follow\_recommendations.common.candidate\_sources.base.SimilarUserExpanderParams.DefaultExpansionInputCount

import com.twitter.follow\_recommendations.common.candidate\_sources.base.SimilarUserExpanderParams.DefaultFinalCandidatesReturnedCount

import com.twitter.follow\_recommendations.common.candidate\_sources.base.SimilarUserExpanderParams.EnableNonDirectFollowExpansion

import com.twitter.follow\_recommendations.common.candidate\_sources.base.SimilarUserExpanderParams.EnableSimsExpandSeedAccountsSort

import com.twitter.follow\_recommendations.common.candidate\_sources.base.SimilarUserExpanderRepository.DefaultCandidateBuilder

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

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

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

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

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

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

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

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

import com.twitter.hermit.candidate.thriftscala.Candidates

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.timelines.configapi.FSBoundedParam

import com.twitter.timelines.configapi.FSParam

import com.twitter.timelines.configapi.HasParams

import com.twitter.timelines.configapi.Params

case class SecondDegreeCandidate(userId: Long, score: Double, socialProof: Option[Seq[Long]])

abstract class SimilarUserExpanderRepository[-Request <: HasParams](

override val identifier: CandidateSourceIdentifier,

similarToCandidatesFetcher: Fetcher[

Long,

Unit,

Candidates

],

expansionInputSizeParam: FSBoundedParam[Int] = DefaultExpansionInputCount,

candidatesReturnedSizeParam: FSBoundedParam[Int] = DefaultFinalCandidatesReturnedCount,

enableImplicitEngagedExpansion: FSParam[Boolean] = DefaultEnableImplicitEngagedExpansion,

thresholdToAvoidExpansion: Int = 30,

maxExpansionPerCandidate: Option[Int] = None,

includingOriginalCandidates: Boolean = false,

scorer: (Double, Double) => Double = SimilarUserExpanderRepository.DefaultScorer,

aggregator: (Seq[Double]) => Double = ScoreAggregator.Max,

candidateBuilder: (Long, CandidateSourceIdentifier, Double, CandidateUser) => CandidateUser =

DefaultCandidateBuilder)

extends TwoHopExpansionCandidateSource[

Request,

CandidateUser,

SecondDegreeCandidate,

CandidateUser

] {

val originalCandidateSource: CandidateSource[Request, CandidateUser]

val backupOriginalCandidateSource: Option[CandidateSource[Request, CandidateUser]] = None

override def firstDegreeNodes(request: Request): Stitch[Seq[CandidateUser]] = {

val originalCandidatesStitch: Stitch[Seq[CandidateUser]] =

originalCandidateSource(request)

val backupCandidatesStitch: Stitch[Seq[CandidateUser]] =

if (request.params(EnableNonDirectFollowExpansion)) {

backupOriginalCandidateSource.map(\_.apply(request)).getOrElse(Stitch.Nil)

} else {

Stitch.Nil

}

val firstDegreeCandidatesCombinedStitch: Stitch[Seq[CandidateUser]] =

Stitch

.join(originalCandidatesStitch, backupCandidatesStitch).map {

case (firstDegreeOrigCandidates, backupFirstDegreeCandidates) =>

if (request.params(EnableSimsExpandSeedAccountsSort)) {

firstDegreeOrigCandidates ++ backupFirstDegreeCandidates sortBy {

-\_.score.getOrElse(DefaultScore)

}

} else {

firstDegreeOrigCandidates ++ backupFirstDegreeCandidates

}

}

val candidatesAfterImplicitEngagementsRemovalStitch: Stitch[Seq[CandidateUser]] =

getCandidatesAfterImplicitEngagementFiltering(

request.params,

firstDegreeCandidatesCombinedStitch)

val firstDegreeCandidatesCombinedTrimmed = candidatesAfterImplicitEngagementsRemovalStitch.map {

candidates: Seq[CandidateUser] =>

candidates.take(request.params(expansionInputSizeParam))

}

firstDegreeCandidatesCombinedTrimmed.map { firstDegreeResults: Seq[CandidateUser] =>

if (firstDegreeResults.nonEmpty && firstDegreeResults.size < thresholdToAvoidExpansion) {

firstDegreeResults

.groupBy(\_.id).mapValues(

\_.maxBy(\_.score)

).values.toSeq

} else {

Nil

}

}

}

override def secondaryDegreeNodes(

request: Request,

firstDegreeCandidate: CandidateUser

): Stitch[Seq[SecondDegreeCandidate]] = {

similarToCandidatesFetcher.fetch(firstDegreeCandidate.id).map(\_.v).map { candidateListOption =>

candidateListOption

.map { candidatesList =>

candidatesList.candidates.map(candidate =>

SecondDegreeCandidate(candidate.userId, candidate.score, candidate.socialProof))

}.getOrElse(Nil)

}

}

override def aggregateAndScore(

req: Request,

firstDegreeToSecondDegreeNodesMap: Map[CandidateUser, Seq[SecondDegreeCandidate]]

): Stitch[Seq[CandidateUser]] = {

val similarExpanderResults = firstDegreeToSecondDegreeNodesMap.flatMap {

case (firstDegreeCandidate, seqOfSecondDegreeCandidates) =>

val sourceScore = firstDegreeCandidate.score.getOrElse(DefaultScore)

val results: Seq[CandidateUser] = seqOfSecondDegreeCandidates.map { secondDegreeCandidate =>

val score = scorer(sourceScore, secondDegreeCandidate.score)

candidateBuilder(secondDegreeCandidate.userId, identifier, score, firstDegreeCandidate)

}

maxExpansionPerCandidate match {

case None => results

case Some(limit) => results.sortBy(-\_.score.getOrElse(DefaultScore)).take(limit)

}

}.toSeq

val allCandidates = {

if (includingOriginalCandidates)

firstDegreeToSecondDegreeNodesMap.keySet.toSeq

else

Nil

} ++ similarExpanderResults

val groupedCandidates: Seq[CandidateUser] = allCandidates

.groupBy(\_.id)

.flatMap {

case (\_, candidates) =>

val finalScore = aggregator(candidates.map(\_.score.getOrElse(DefaultScore)))

val candidateSourceDetailsCombined = aggregateCandidateSourceDetails(candidates)

val accountSocialProofcombined = aggregateAccountSocialProof(candidates)

candidates.headOption.map(

\_.copy(

score = Some(finalScore),

reason = accountSocialProofcombined,

userCandidateSourceDetails = candidateSourceDetailsCombined)

.withCandidateSource(identifier))

}

.toSeq

Stitch.value(

groupedCandidates

.sortBy { -\_.score.getOrElse(DefaultScore) }.take(req.params(candidatesReturnedSizeParam))

)

}

def aggregateCandidateSourceDetails(

candidates: Seq[CandidateUser]

): Option[UserCandidateSourceDetails] = {

candidates

.map { candidate =>

candidate.userCandidateSourceDetails.map(\_.candidateSourceScores).getOrElse(Map.empty)

}.reduceLeftOption { (scoreMap1, scoreMap2) =>

scoreMap1 ++ scoreMap2

}.map {

UserCandidateSourceDetails(primaryCandidateSource = None, \_)

}

}

def aggregateAccountSocialProof(candidates: Seq[CandidateUser]): Option[Reason] = {

candidates

.map { candidate =>

(

candidate.reason

.flatMap(\_.accountProof.flatMap(\_.similarToProof.map(\_.similarTo))).getOrElse(Nil),

candidate.reason

.flatMap(\_.accountProof.flatMap(\_.followProof.map(\_.followedBy))).getOrElse(Nil),

candidate.reason

.flatMap(\_.accountProof.flatMap(\_.followProof.map(\_.numIds))).getOrElse(0)

)

}.reduceLeftOption { (accountProofOne, accountProofTwo) =>

(

// merge similarToIds

accountProofOne.\_1 ++ accountProofTwo.\_1,

// merge followedByIds

accountProofOne.\_2 ++ accountProofTwo.\_2,

// add numIds

accountProofOne.\_3 + accountProofTwo.\_3)

}.map { proofs =>

Reason(accountProof = Some(

AccountProof(

similarToProof = Some(SimilarToProof(proofs.\_1)),

followProof = if (proofs.\_2.nonEmpty) Some(FollowProof(proofs.\_2, proofs.\_3)) else None

)))

}

}

def getCandidatesAfterImplicitEngagementFiltering(

params: Params,

firstDegreeCandidatesStitch: Stitch[Seq[CandidateUser]]

): Stitch[Seq[CandidateUser]] = {

if (!params(enableImplicitEngagedExpansion)) {

/\*\*

\* Remove candidates whose engagement types only contain implicit engagements

\* (e.g. Profile View, Tweet Click) and only expand those candidates who contain explicit

\* engagements.

\*/

firstDegreeCandidatesStitch.map { candidates =>

candidates.filter { cand =>

cand.engagements.exists(engage =>

engage == EngagementType.Like || engage == EngagementType.Retweet || engage == EngagementType.Mention)

}

}

} else {

firstDegreeCandidatesStitch

}

}

}

object SimilarUserExpanderRepository {

val DefaultScorer: (Double, Double) => Double = (sourceScore: Double, similarScore: Double) =>

similarScore

val MultiplyScorer: (Double, Double) => Double = (sourceScore: Double, similarScore: Double) =>

sourceScore \* similarScore

val SourceScorer: (Double, Double) => Double = (sourceScore: Double, similarScore: Double) =>

sourceScore

val DefaultScore = 0.0d

val DefaultCandidateBuilder: (

Long,

CandidateSourceIdentifier,

Double,

CandidateUser

) => CandidateUser =

(

userId: Long,

\_: CandidateSourceIdentifier,

score: Double,

candidate: CandidateUser

) => {

val originalCandidateSourceDetails =

candidate.userCandidateSourceDetails.flatMap { candSourceDetails =>

candSourceDetails.primaryCandidateSource.map { primaryCandidateSource =>

UserCandidateSourceDetails(

primaryCandidateSource = None,

candidateSourceScores = Map(primaryCandidateSource -> candidate.score))

}

}

CandidateUser(

id = userId,

score = Some(score),

userCandidateSourceDetails = originalCandidateSourceDetails,

reason =

Some(Reason(Some(AccountProof(similarToProof = Some(SimilarToProof(Seq(candidate.id)))))))

)

}

val FollowClusterCandidateBuilder: (

Long,

CandidateSourceIdentifier,

Double,

CandidateUser

) => CandidateUser =

(userId: Long, \_: CandidateSourceIdentifier, score: Double, candidate: CandidateUser) => {

val originalCandidateSourceDetails =

candidate.userCandidateSourceDetails.flatMap { candSourceDetails =>

candSourceDetails.primaryCandidateSource.map { primaryCandidateSource =>

UserCandidateSourceDetails(

primaryCandidateSource = None,

candidateSourceScores = Map(primaryCandidateSource -> candidate.score))

}

}

val originalFollowCluster = candidate.reason

.flatMap(\_.accountProof.flatMap(\_.followProof.map(\_.followedBy)))

CandidateUser(

id = userId,

score = Some(score),

userCandidateSourceDetails = originalCandidateSourceDetails,

reason = Some(

Reason(

Some(

AccountProof(

similarToProof = Some(SimilarToProof(Seq(candidate.id))),

followProof = originalFollowCluster.map(follows =>

FollowProof(follows, follows.size)))))

)

)

}

}

object ScoreAggregator {

// aggregate the same candidates with same id by taking the one with largest score

val Max: Seq[Double] => Double = (candidateScores: Seq[Double]) => { candidateScores.max }

// aggregate the same candidates with same id by taking the sum of the scores

val Sum: Seq[Double] => Double = (candidateScores: Seq[Double]) => { candidateScores.sum }

}