package com.twitter.follow\_recommendations.common.candidate\_sources.sims\_expansion

import com.twitter.follow\_recommendations.common.candidate\_sources.base.TwoHopExpansionCandidateSource

import com.twitter.follow\_recommendations.common.candidate\_sources.sims.SwitchingSimsSource

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

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

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

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

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

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

import scala.math.\_

case class SimilarUser(candidateId: Long, similarTo: Long, score: Double)

abstract class SimsExpansionBasedCandidateSource[-Target <: HasParams](

switchingSimsSource: SwitchingSimsSource)

extends TwoHopExpansionCandidateSource[Target, CandidateUser, SimilarUser, CandidateUser] {

// max number secondary degree nodes per first degree node

def maxSecondaryDegreeNodes(req: Target): Int

// max number output results

def maxResults(req: Target): Int

// scorer to score candidate based on first and second degree node scores

def scoreCandidate(source: Double, similarToScore: Double): Double

def calibrateDivisor(req: Target): Double

def calibrateScore(candidateScore: Double, req: Target): Double = {

candidateScore / calibrateDivisor(req)

}

override def secondaryDegreeNodes(req: Target, node: CandidateUser): Stitch[Seq[SimilarUser]] = {

switchingSimsSource(new HasParams with HasSimilarToContext {

override val similarToUserIds = Seq(node.id)

override val params = (req.params)

}).map(\_.take(maxSecondaryDegreeNodes(req)).map { candidate =>

SimilarUser(

candidate.id,

node.id,

(node.score, candidate.score) match {

// only calibrated sims expanded candidates scores

case (Some(nodeScore), Some(candidateScore)) =>

calibrateScore(scoreCandidate(nodeScore, candidateScore), req)

case (Some(nodeScore), \_) => nodeScore

// NewFollowingSimilarUser will enter this case

case \_ => calibrateScore(candidate.score.getOrElse(0.0), req)

}

)

})

}

override def aggregateAndScore(

request: Target,

firstDegreeToSecondDegreeNodesMap: Map[CandidateUser, Seq[SimilarUser]]

): Stitch[Seq[CandidateUser]] = {

val inputNodes = firstDegreeToSecondDegreeNodesMap.keys.map(\_.id).toSet

val aggregator = request.params(SimsExpansionSourceParams.Aggregator) match {

case SimsExpansionSourceAggregatorId.Max =>

SimsExpansionBasedCandidateSource.ScoreAggregator.Max

case SimsExpansionSourceAggregatorId.Sum =>

SimsExpansionBasedCandidateSource.ScoreAggregator.Sum

case SimsExpansionSourceAggregatorId.MultiDecay =>

SimsExpansionBasedCandidateSource.ScoreAggregator.MultiDecay

}

val groupedCandidates = firstDegreeToSecondDegreeNodesMap.values.flatten

.filterNot(c => inputNodes.contains(c.candidateId))

.groupBy(\_.candidateId)

.map {

case (id, candidates) =>

// Different aggregators for final score

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

val proofs = candidates.map(\_.similarTo).toSet

CandidateUser(

id = id,

score = Some(finalScore),

reason =

Some(Reason(Some(AccountProof(similarToProof = Some(SimilarToProof(proofs.toSeq))))))

).withCandidateSource(identifier)

}

.toSeq

.sortBy(-\_.score.getOrElse(0.0d))

.take(maxResults(request))

Stitch.value(groupedCandidates)

}

}

object SimsExpansionBasedCandidateSource {

object ScoreAggregator {

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

if (candidateScores.size > 0) candidateScores.max else 0.0

}

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

candidateScores.sum

}

val MultiDecay: Seq[Double] => Double = (candidateScores: Seq[Double]) => {

val alpha = 0.1

val beta = 0.1

val gamma = 0.8

val decay\_scores: Seq[Double] =

candidateScores

.sorted(Ordering[Double].reverse)

.zipWithIndex

.map(x => x.\_1 \* pow(gamma, x.\_2))

alpha \* candidateScores.max + decay\_scores.sum + beta \* candidateScores.size

}

}

}