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

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

import com.twitter.product\_mixer.core.model.marshalling.request.HasClientContext

import com.twitter.stitch.Stitch

import com.twitter.strato.generated.client.onboarding.userrecs.StrongTiePredictionFeaturesOnUserClientColumn

import com.twitter.timelines.configapi.HasParams

import com.twitter.wtf.scalding.jobs.strong\_tie\_prediction.FirstDegreeEdge

import com.twitter.wtf.scalding.jobs.strong\_tie\_prediction.SecondDegreeEdge

import com.twitter.wtf.scalding.jobs.strong\_tie\_prediction.SecondDegreeEdgeInfo

import javax.inject.Inject

import javax.inject.Singleton

// Link to code functionality we're migrating

@Singleton

class STPSecondDegreeFetcher @Inject() (

strongTiePredictionFeaturesOnUserClientColumn: StrongTiePredictionFeaturesOnUserClientColumn) {

private def scoreSecondDegreeEdge(edge: SecondDegreeEdge): (Int, Int, Int) = {

def bool2int(b: Boolean): Int = if (b) 1 else 0

(

-edge.edgeInfo.numMutualFollowPath,

-edge.edgeInfo.numLowTweepcredFollowPath,

-(bool2int(edge.edgeInfo.forwardEmailPath) + bool2int(edge.edgeInfo.reverseEmailPath) +

bool2int(edge.edgeInfo.forwardPhonePath) + bool2int(edge.edgeInfo.reversePhonePath))

)

}

// Use each first-degree edge(w/ candidateId) to expand and find mutual follows.

// Then, with the mutual follows, group-by candidateId and join edge information

// to create secondDegree edges.

def getSecondDegreeEdges(

target: HasClientContext with HasParams,

firstDegreeEdges: Seq[FirstDegreeEdge]

): Stitch[Seq[SecondDegreeEdge]] = {

target.getOptionalUserId

.map { userId =>

val firstDegreeConnectingIds = firstDegreeEdges.map(\_.dstId)

val firstDegreeEdgeInfoMap = firstDegreeEdges.map(e => (e.dstId, e.edgeInfo)).toMap

val intermediateSecondDegreeEdgesStitch = Stitch

.traverse(firstDegreeConnectingIds) { connectingId =>

val stpFeaturesOptStitch = strongTiePredictionFeaturesOnUserClientColumn.fetcher

.fetch(connectingId)

.map(\_.v)

stpFeaturesOptStitch.map { stpFeatureOpt =>

val intermediateSecondDegreeEdges = for {

edgeInfo <- firstDegreeEdgeInfoMap.get(connectingId)

stpFeatures <- stpFeatureOpt

topSecondDegreeUserIds =

stpFeatures.topMutualFollows

.getOrElse(Nil)

.map(\_.userId)

.take(STPSecondDegreeFetcher.MaxNumOfMutualFollows)

} yield topSecondDegreeUserIds.map(

IntermediateSecondDegreeEdge(connectingId, \_, edgeInfo))

intermediateSecondDegreeEdges.getOrElse(Nil)

}

}.map(\_.flatten)

intermediateSecondDegreeEdgesStitch.map { intermediateSecondDegreeEdges =>

val secondaryDegreeEdges = intermediateSecondDegreeEdges.groupBy(\_.candidateId).map {

case (candidateId, intermediateEdges) =>

SecondDegreeEdge(

srcId = userId,

dstId = candidateId,

edgeInfo = SecondDegreeEdgeInfo(

numMutualFollowPath = intermediateEdges.count(\_.edgeInfo.mutualFollow),

numLowTweepcredFollowPath =

intermediateEdges.count(\_.edgeInfo.lowTweepcredFollow),

forwardEmailPath = intermediateEdges.exists(\_.edgeInfo.forwardEmail),

reverseEmailPath = intermediateEdges.exists(\_.edgeInfo.reverseEmail),

forwardPhonePath = intermediateEdges.exists(\_.edgeInfo.forwardPhone),

reversePhonePath = intermediateEdges.exists(\_.edgeInfo.reversePhone),

socialProof = intermediateEdges

.filter { e => e.edgeInfo.mutualFollow || e.edgeInfo.lowTweepcredFollow }

.sortBy(-\_.edgeInfo.realGraphWeight)

.take(3)

.map { c => (c.connectingId, c.edgeInfo.realGraphWeight) }

)

)

}

secondaryDegreeEdges.toSeq

.sortBy(scoreSecondDegreeEdge)

.take(STPSecondDegreeFetcher.MaxNumSecondDegreeEdges)

}

}.getOrElse(Stitch.Nil)

}

}

object STPSecondDegreeFetcher {

val MaxNumSecondDegreeEdges = 200

val MaxNumOfMutualFollows = 50

}