package com.twitter.simclusters\_v2.scalding

import com.twitter.algebird.{Monoid, Semigroup}

import com.twitter.scalding.\_

object UpdateKnownFor {

/\*\*

\* Convenience datastructure that can summarize key stats about a node's set of

\* immediate neighbors.

\*

\* @param nodeCount number of nodes

\* @param sumOfEdgeWeights sum of weights on edges in the neighborhood.

\* @param sumOfMembershipWeightedEdgeWeights sum of { edge weight \* membership weight } for each node

\* in the neighborhood. Membership weight to what is not

\* specified in this case class and is instead part of the

\* context.

\* @param sumOfMembershipWeights sum of membership weight for each node in the

\* neighborhood. Membership weight to what is not

\* specified in this case class and is instead part of

\* the context.

\*/

case class NeighborhoodInformation(

nodeCount: Int,

sumOfEdgeWeights: Float,

sumOfMembershipWeightedEdgeWeights: Float,

sumOfMembershipWeights: Float)

object NeighborhoodInformationMonoid extends Monoid[NeighborhoodInformation] {

override val zero: NeighborhoodInformation = NeighborhoodInformation(0, 0f, 0f, 0f)

override def plus(l: NeighborhoodInformation, r: NeighborhoodInformation) =

NeighborhoodInformation(

l.nodeCount + r.nodeCount,

l.sumOfEdgeWeights + r.sumOfEdgeWeights,

l.sumOfMembershipWeightedEdgeWeights + r.sumOfMembershipWeightedEdgeWeights,

l.sumOfMembershipWeights + r.sumOfMembershipWeights

)

}

case class NodeInformation(

originalClusters: List[Int],

overallStats: NeighborhoodInformation,

statsOfClustersInNeighborhood: Map[Int, NeighborhoodInformation])

object NodeInformationSemigroup extends Semigroup[NodeInformation] {

implicit val ctsMonoid: Monoid[NeighborhoodInformation] = NeighborhoodInformationMonoid

override def plus(l: NodeInformation, r: NodeInformation) =

NodeInformation(

l.originalClusters ++ r.originalClusters,

ctsMonoid.plus(l.overallStats, r.overallStats),

Monoid

.mapMonoid[Int, NeighborhoodInformation].plus(

l.statsOfClustersInNeighborhood,

r.statsOfClustersInNeighborhood)

)

}

case class ClusterScoresForNode(

sumScoreIgnoringMembershipScores: Double,

ratioScoreIgnoringMembershipScores: Double,

ratioScoreUsingMembershipScores: Double)

/\*\*

\* Given a user and a cluster:

\* True positive weight = sum of edge weights to neighbors who belong to that cluster.

\* False negative weight = sum of edge weights to neighbors who don’t belong to that cluster.

\* False positive weight = (number of users in the cluster who are not neighbors of the node) \* globalAvgEdgeWeight

\* Membership-weighted true positive weight = for neighbors who are also in the cluster, sum of edge weight times user membership score in the cluster.

\* Membership-weighted false negative weight = for neighbors who are not in the cluster, sum of edge weight times avg membership score across the whole knownFor input.

\* Membership-weighted false positive weight = for users in the cluster who are not neighbors of the node, avg global edge weight times user membership score for the cluster.

\*

\* Ignoring membership scores, sum formula:

\* truePositiveWtFactor\*(True positive weight) - false negative weight - false positive weight

\* Ignoring membership scores, ratio formula:

\* True positive weight / (true positive weight + false negative weight + false positive weight)

\* Using membership scores

\* Membership-weighted true positive weight / (Membership-weighted true positive weight + Membership-weighted false negative weight + Membership-weighted false positive weight)

\*

\* @param overallNeighborhoodStats

\* @param statsForCluster

\* @param clusterSize

\* @param sumOfClusterMembershipScores

\* @param globalAvgEdgeWeight

\* @param truePositiveWtFactor

\*

\* @return

\*/

def getScoresForCluster(

overallNeighborhoodStats: NeighborhoodInformation,

statsForCluster: NeighborhoodInformation,

clusterSize: Int,

sumOfClusterMembershipScores: Double,

globalAvgEdgeWeight: Double,

truePositiveWtFactor: Double

): ClusterScoresForNode = {

val truePositiveWt = statsForCluster.sumOfEdgeWeights

val falseNegativeWt = overallNeighborhoodStats.sumOfEdgeWeights - truePositiveWt

val falsePositiveWt = (clusterSize - statsForCluster.nodeCount) \* globalAvgEdgeWeight

val membershipWeightedTruePositiveWt = statsForCluster.sumOfMembershipWeightedEdgeWeights

val membershipWeightedFalseNegativeWt =

overallNeighborhoodStats.sumOfMembershipWeightedEdgeWeights - membershipWeightedTruePositiveWt

val membershipWeightedFalsePositiveWt =

(sumOfClusterMembershipScores - statsForCluster.sumOfMembershipWeights) \* globalAvgEdgeWeight

val sumScore =

truePositiveWtFactor \* statsForCluster.sumOfEdgeWeights - falseNegativeWt - falsePositiveWt

val ratioScore = truePositiveWt / (truePositiveWt + falseNegativeWt + falsePositiveWt)

val ratioUsingMemberships =

membershipWeightedTruePositiveWt / (membershipWeightedTruePositiveWt +

membershipWeightedFalsePositiveWt + membershipWeightedFalseNegativeWt)

ClusterScoresForNode(sumScore, ratioScore, ratioUsingMemberships)

}

def pickBestCluster(

overallNeighborhoodStats: NeighborhoodInformation,

statsOfClustersInNeighborhood: Map[Int, NeighborhoodInformation],

clusterOverallStatsMap: Map[Int, NeighborhoodInformation],

globalAvgEdgeWeight: Double,

truePositiveWtFactor: Double,

clusterScoresToFinalScore: ClusterScoresForNode => Double,

minNeighborsInCluster: Int

): Option[(Int, Double)] = {

val clusterToScores = statsOfClustersInNeighborhood.toList.flatMap {

case (clusterId, statsInNeighborhood) =>

val clusterOverallStats = clusterOverallStatsMap(clusterId)

if (statsInNeighborhood.nodeCount >= minNeighborsInCluster) {

Some(

(

clusterId,

clusterScoresToFinalScore(

getScoresForCluster(

overallNeighborhoodStats,

statsInNeighborhood,

clusterOverallStats.nodeCount,

clusterOverallStats.sumOfMembershipWeights,

globalAvgEdgeWeight,

truePositiveWtFactor

)

)

)

)

} else {

None

}

}

if (clusterToScores.nonEmpty) {

Some(clusterToScores.maxBy(\_.\_2))

} else None

}

def updateGeneric(

graph: TypedPipe[(Long, Map[Long, Float])],

inputUserToClusters: TypedPipe[(Long, Array[(Int, Float)])],

clusterOverallStatsMap: Map[Int, NeighborhoodInformation],

minNeighborsInCluster: Int,

globalAvgWeight: Double,

avgMembershipScore: Double,

truePositiveWtFactor: Double,

clusterScoresToFinalScore: ClusterScoresForNode => Double

)(

implicit uniqId: UniqueID

): TypedPipe[(Long, Array[(Int, Float)])] = {

val emptyToSomething = Stat("no\_assignment\_to\_some")

val somethingToEmpty = Stat("some\_assignment\_to\_none")

val emptyToEmpty = Stat("empty\_to\_empty")

val sameCluster = Stat("same\_cluster")

val diffCluster = Stat("diff\_cluster")

val nodesWithSmallDegree = Stat("nodes\_with\_degree\_lt\_" + minNeighborsInCluster)

collectInformationPerNode(graph, inputUserToClusters, avgMembershipScore)

.mapValues {

case NodeInformation(originalClusters, overallStats, statsOfClustersInNeighborhood) =>

val newClusterWithScoreOpt = if (overallStats.nodeCount < minNeighborsInCluster) {

nodesWithSmallDegree.inc()

None

} else {

pickBestCluster(

overallStats,

statsOfClustersInNeighborhood,

clusterOverallStatsMap,

globalAvgWeight,

truePositiveWtFactor,

clusterScoresToFinalScore,

minNeighborsInCluster

)

}

newClusterWithScoreOpt match {

case Some((newClusterId, score)) =>

if (originalClusters.isEmpty) {

emptyToSomething.inc()

} else if (originalClusters.contains(newClusterId)) {

sameCluster.inc()

} else {

diffCluster.inc()

}

Array((newClusterId, score.toFloat))

case None =>

if (originalClusters.isEmpty) {

emptyToEmpty.inc()

} else {

somethingToEmpty.inc()

}

Array.empty[(Int, Float)]

}

}

}

/\*\*

\* Assembles the information we need at a node in order to decide what the new cluster should be.

\* So this is where we assemble what the overall

\*

\* This function is where all the crucial steps take place. First get the cluster that each

\* node belongs to, and then broadcast information about this node and cluster membership to each

\* of it's neighbors. Now bring together all records with the same nodeId as the key and create

\* the NodeInformation dataset.

\* @param graph symmetric graph i.e. if u is in v's adj list, then v is in u's adj list.

\* @param userToClusters current knownFor.

\* @param avgMembershipScore avg. membership score of a node in the knownFor we're updating.

\* Useful to deal with nodes which don't belong to any knownFor.

\* @return pipe with node information for each node

\*/

def collectInformationPerNode(

graph: TypedPipe[(Long, Map[Long, Float])],

userToClusters: TypedPipe[(Long, Array[(Int, Float)])],

avgMembershipScore: Double

): TypedPipe[(Long, NodeInformation)] = {

implicit val nisg: Semigroup[NodeInformation] = NodeInformationSemigroup

graph

.leftJoin(userToClusters)

// uncomment for adhoc job

//.withReducers(200)

.flatMap {

case (nodeId, (adjList, assignedClustersOpt)) =>

val assignedClusters =

assignedClustersOpt.map(\_.toList).getOrElse(Nil)

val res = adjList.toList.flatMap {

case (neighborId, neighborWeight) =>

if (assignedClusters.nonEmpty) {

assignedClusters.map {

case (clusterId, membershipScore) =>

val neighborhoodInformationForCluster = NeighborhoodInformation(

1,

neighborWeight,

membershipScore \* neighborWeight,

membershipScore)

val originalClusters =

if (neighborId == nodeId) List(clusterId)

else List.empty[Int]

(

neighborId,

NodeInformation(

originalClusters,

neighborhoodInformationForCluster,

Map(clusterId -> neighborhoodInformationForCluster)))

}

} else {

List(

(

neighborId,

NodeInformation(

Nil,

NeighborhoodInformation(

1,

neighborWeight,

(avgMembershipScore \* neighborWeight).toFloat,

avgMembershipScore.toFloat),

Map.empty[Int, NeighborhoodInformation]

)))

}

}

res

}

.sumByKey

// uncomment for adhoc job

//.withReducers(100)

}

/\*\*

\* Replace incoming knownFor scores with ratioScoreIgnoringMembershipScores

\* @param knownFor

\* @param simsGraphWithoutSelfLoops

\* @param globalAvgWeight

\* @param clusterStats

\* @param avgMembershipScore

\* @return

\*/

def newKnownForScores(

knownFor: TypedPipe[(Long, Array[(Int, Float)])],

simsGraphWithoutSelfLoops: TypedPipe[(Long, Map[Long, Float])],

globalAvgWeight: Double,

clusterStats: Map[Int, NeighborhoodInformation],

avgMembershipScore: Double

): TypedPipe[(Long, Array[(Int, Float)])] = {

collectInformationPerNode(simsGraphWithoutSelfLoops, knownFor, avgMembershipScore)

.mapValues {

case NodeInformation(originalClusters, overallStats, statsOfClustersInNeighborhood) =>

originalClusters.map { clusterId =>

(

clusterId,

getScoresForCluster(

overallStats,

statsOfClustersInNeighborhood(clusterId),

clusterStats(clusterId).nodeCount,

clusterStats(clusterId).sumOfMembershipWeights,

globalAvgWeight,

0

).ratioScoreIgnoringMembershipScores.toFloat)

}.toArray

}

}

}