package com.twitter.simclusters\_v2.common.clustering

import com.twitter.eventdetection.common.louvain.LouvainDriver

import com.twitter.eventdetection.common.louvain.NetworkFactory

import com.twitter.eventdetection.common.model.Entity

import com.twitter.eventdetection.common.model.NetworkInput

import com.twitter.eventdetection.common.model.TextEntityValue

import com.twitter.util.Stopwatch

import scala.collection.JavaConverters.\_

import scala.math.max

/\*\*

\* Groups entities by the Louvain clustering method.

\* @param similarityThreshold: When building the edges between entities, edges with weight

\* less than or equal to this threshold will be filtered out.

\* @param appliedResolutionFactor: If present, will be used to multiply the applied resolution

\* parameter of the Louvain method by this factor.

\* Note that the DEFAULT\_MAX\_RESOLUTION will not be applied.

\*/

class LouvainClusteringMethod(

similarityThreshold: Double,

appliedResolutionFactor: Option[Double])

extends ClusteringMethod {

import ClusteringStatistics.\_

def cluster[T](

embeddings: Map[Long, T],

similarityFn: (T, T) => Double,

recordStatCallback: (String, Long) => Unit = (\_, \_) => ()

): Set[Set[Long]] = {

// 1. Build the graph on which to run Louvain:

// - Weigh edges by the similarity between the 2 embeddings,

// - Filter out edges with weight <= threshold.

val timeSinceGraphBuildStart = Stopwatch.start()

val edges: Seq[((Long, Long), Double)] = embeddings.toSeq

.combinations(2)

.map { pair: Seq[(Long, T)] => // pair of 2

val (user1, embedding1) = pair.head

val (user2, embedding2) = pair(1)

val similarity = similarityFn(embedding1, embedding2)

recordStatCallback(

StatComputedSimilarityBeforeFilter,

(similarity \* 100).toLong // preserve up to two decimal places

)

((user1, user2), similarity)

}

.filter(\_.\_2 > similarityThreshold)

.toSeq

recordStatCallback(StatSimilarityGraphTotalBuildTime, timeSinceGraphBuildStart().inMilliseconds)

// check if some entities do not have any incoming / outgoing edge

// these are size-1 clusters (i.e. their own)

val individualClusters: Set[Long] = embeddings.keySet -- edges.flatMap {

case ((user1, user2), \_) => Set(user1, user2)

}.toSet

// 2. LouvainDriver uses "Entity" as input, so build 2 mappings

// - Long (entity id) -> Entity

// - Entity -> Long (entity id)

val embeddingIdToEntity: Map[Long, Entity] = embeddings.map {

case (id, \_) => id -> Entity(TextEntityValue(id.toString, Some(id.toString)), None)

}

val entityToEmbeddingId: Map[Entity, Long] = embeddingIdToEntity.map {

case (id, e) => e -> id

}

// 3. Create the list of NetworkInput on which to run LouvainDriver

val networkInputList = edges

.map {

case ((fromUserId: Long, toUserId: Long), weight: Double) =>

new NetworkInput(embeddingIdToEntity(fromUserId), embeddingIdToEntity(toUserId), weight)

}.toList.asJava

val timeSinceClusteringAlgRunStart = Stopwatch.start()

val networkDictionary = NetworkFactory.buildDictionary(networkInputList)

val network = NetworkFactory.buildNetwork(networkInputList, networkDictionary)

if (networkInputList.size() == 0) {

// handle case if no edge at all (only one entity or all entities are too far apart)

embeddings.keySet.map(e => Set(e))

} else {

// 4. Run clustering algorithm

val clusteredIds = appliedResolutionFactor match {

case Some(res) =>

LouvainDriver.clusterAppliedResolutionFactor(network, networkDictionary, res)

case None => LouvainDriver.cluster(network, networkDictionary)

}

recordStatCallback(

StatClusteringAlgorithmRunTime,

timeSinceClusteringAlgRunStart().inMilliseconds)

// 5. Post-processing

val atLeast2MembersClusters: Set[Set[Long]] = clusteredIds.asScala

.groupBy(\_.\_2)

.mapValues(\_.map { case (e, \_) => entityToEmbeddingId(e) }.toSet)

.values.toSet

atLeast2MembersClusters ++ individualClusters.map { e => Set(e) }

}

}

def clusterWithSilhouette[T](

embeddings: Map[Long, T],

similarityFn: (T, T) => Double,

similarityFnForSil: (T, T) => Double,

recordStatCallback: (String, Long) => Unit = (\_, \_) => ()

): (Set[Set[Long]], Set[Set[(Long, Double)]]) = {

// 1. Build the graph on which to run Louvain:

// - Weigh edges by the similarity between the 2 embeddings,

// - Filter out edges with weight <= threshold.

val timeSinceGraphBuildStart = Stopwatch.start()

val edgesSimilarityMap = collection.mutable.Map[(Long, Long), Double]()

val edges: Seq[((Long, Long), Double)] = embeddings.toSeq

.combinations(2)

.map { pair: Seq[(Long, T)] => // pair of 2

val (user1, embedding1) = pair.head

val (user2, embedding2) = pair(1)

val similarity = similarityFn(embedding1, embedding2)

val similarityForSil = similarityFnForSil(embedding1, embedding2)

edgesSimilarityMap.put((user1, user2), similarityForSil)

edgesSimilarityMap.put((user2, user1), similarityForSil)

recordStatCallback(

StatComputedSimilarityBeforeFilter,

(similarity \* 100).toLong // preserve up to two decimal places

)

((user1, user2), similarity)

}

.filter(\_.\_2 > similarityThreshold)

.toSeq

recordStatCallback(StatSimilarityGraphTotalBuildTime, timeSinceGraphBuildStart().inMilliseconds)

// check if some entities do not have any incoming / outgoing edge

// these are size-1 clusters (i.e. their own)

val individualClusters: Set[Long] = embeddings.keySet -- edges.flatMap {

case ((user1, user2), \_) => Set(user1, user2)

}.toSet

// 2. LouvainDriver uses "Entity" as input, so build 2 mappings

// - Long (entity id) -> Entity

// - Entity -> Long (entity id)

val embeddingIdToEntity: Map[Long, Entity] = embeddings.map {

case (id, \_) => id -> Entity(TextEntityValue(id.toString, Some(id.toString)), None)

}

val entityToEmbeddingId: Map[Entity, Long] = embeddingIdToEntity.map {

case (id, e) => e -> id

}

// 3. Create the list of NetworkInput on which to run LouvainDriver

val networkInputList = edges

.map {

case ((fromUserId: Long, toUserId: Long), weight: Double) =>

new NetworkInput(embeddingIdToEntity(fromUserId), embeddingIdToEntity(toUserId), weight)

}.toList.asJava

val timeSinceClusteringAlgRunStart = Stopwatch.start()

val networkDictionary = NetworkFactory.buildDictionary(networkInputList)

val network = NetworkFactory.buildNetwork(networkInputList, networkDictionary)

val clusters = if (networkInputList.size() == 0) {

// handle case if no edge at all (only one entity or all entities are too far apart)

embeddings.keySet.map(e => Set(e))

} else {

// 4. Run clustering algorithm

val clusteredIds = appliedResolutionFactor match {

case Some(res) =>

LouvainDriver.clusterAppliedResolutionFactor(network, networkDictionary, res)

case None => LouvainDriver.cluster(network, networkDictionary)

}

recordStatCallback(

StatClusteringAlgorithmRunTime,

timeSinceClusteringAlgRunStart().inMilliseconds)

// 5. Post-processing

val atLeast2MembersClusters: Set[Set[Long]] = clusteredIds.asScala

.groupBy(\_.\_2)

.mapValues(\_.map { case (e, \_) => entityToEmbeddingId(e) }.toSet)

.values.toSet

atLeast2MembersClusters ++ individualClusters.map { e => Set(e) }

}

// Calculate silhouette metrics

val contactIdWithSilhouette = clusters.map {

case cluster =>

val otherClusters = clusters - cluster

cluster.map {

case contactId =>

if (otherClusters.isEmpty) {

(contactId, 0.0)

} else {

val otherSameClusterContacts = cluster - contactId

if (otherSameClusterContacts.isEmpty) {

(contactId, 0.0)

} else {

// calculate similarity of given userId with all other users in the same cluster

val a\_i = otherSameClusterContacts.map {

case sameClusterContact =>

edgesSimilarityMap((contactId, sameClusterContact))

}.sum / otherSameClusterContacts.size

// calculate similarity of given userId to all other clusters, find the best nearest cluster

val b\_i = otherClusters.map {

case otherCluster =>

otherCluster.map {

case otherClusterContact =>

edgesSimilarityMap((contactId, otherClusterContact))

}.sum / otherCluster.size

}.max

// silhouette (value) of one userId i

val s\_i = (a\_i - b\_i) / max(a\_i, b\_i)

(contactId, s\_i)

}

}

}

}

(clusters, contactIdWithSilhouette)

}

}