package com.twitter.simclusters\_v2.common

object CosineSimilarityUtil {

/\*\*

\* Sum of squared elements for a given vector v

\*/

def sumOfSquares[T](v: Map[T, Double]): Double = {

v.values.foldLeft(0.0) { (sum, value) => sum + value \* value }

}

/\*\*

\* Sum of squared elements for a given vector v

\*/

def sumOfSquaresArray(v: Array[Double]): Double = {

v.foldLeft(0.0) { (sum, value) => sum + value \* value }

}

/\*\*

\* Calculate the l2Norm score

\*/

def norm[T](v: Map[T, Double]): Double = {

math.sqrt(sumOfSquares(v))

}

/\*\*

\* Calculate the l2Norm score

\*/

def normArray(v: Array[Double]): Double = {

math.sqrt(sumOfSquaresArray(v))

}

/\*\*

\* Calculate the logNorm score

\*/

def logNorm[T](v: Map[T, Double]): Double = {

math.log(sumOfSquares(v) + 1)

}

/\*\*

\* Calculate the logNorm score

\*/

def logNormArray(v: Array[Double]): Double = {

math.log(sumOfSquaresArray(v) + 1)

}

/\*\*

\* Calculate the exp scaled norm score

\* \*/

def expScaledNorm[T](v: Map[T, Double], exponent: Double): Double = {

math.pow(sumOfSquares(v), exponent)

}

/\*\*

\* Calculate the exp scaled norm score

\* \*/

def expScaledNormArray(v: Array[Double], exponent: Double): Double = {

math.pow(sumOfSquaresArray(v), exponent)

}

/\*\*

\* Calculate the l1Norm score

\*/

def l1Norm[T](v: Map[T, Double]): Double = {

v.values.foldLeft(0.0) { (sum, value) => sum + Math.abs(value) }

}

/\*\*

\* Calculate the l1Norm score

\*/

def l1NormArray(v: Array[Double]): Double = {

v.foldLeft(0.0) { (sum, value) => sum + Math.abs(value) }

}

/\*\*

\* Divide the weight vector with the applied norm

\* Return the original object if the norm is 0

\*

\* @param v a map from cluster id to its weight

\* @param norm a calculated norm from the given map v

\*

\* @return a map with normalized weight

\*/

def applyNorm[T](v: Map[T, Double], norm: Double): Map[T, Double] = {

if (norm == 0) v else v.mapValues(x => x / norm)

}

/\*\*

\* Divide the weight vector with the applied norm

\* Return the original object if the norm is 0

\*

\* @param v a an array of weights

\* @param norm a calculated norm from the given array v

\*

\* @return an array with normalized weight in the same order as v

\*/

def applyNormArray(v: Array[Double], norm: Double): Array[Double] = {

if (norm == 0) v else v.map(\_ / norm)

}

/\*\*

\* Normalize the weight vector for easy cosine similarity calculation. If the input weight vector

\* is empty or its norm is 0, return the original map.

\*

\* @param v a map from cluster id to its weight

\*

\* @return a map with normalized weight (the norm of the weight vector is 1)

\*/

def normalize[T](v: Map[T, Double], maybeNorm: Option[Double] = None): Map[T, Double] = {

val norm = maybeNorm.getOrElse(CosineSimilarityUtil.norm(v))

applyNorm(v, norm)

}

/\*\*

\* Normalize the weight vector for easy cosine similarity calculation. If the input weight vector

\* is empty or its norm is 0, return the original array.

\*

\* @param v an array of weights

\*

\* @return an array with normalized weight (the norm of the weight vector is 1), in the same order as v

\*/

def normalizeArray(

v: Array[Double],

maybeNorm: Option[Double] = None

): Array[Double] = {

val norm = maybeNorm.getOrElse(CosineSimilarityUtil.normArray(v))

applyNormArray(v, norm)

}

/\*\*

\* Normalize the weight vector with log norm. If the input weight vector

\* is empty or its norm is 0, return the original map.

\*

\* @param v a map from cluster id to its weight

\*

\* @return a map with log normalized weight

\* \*/

def logNormalize[T](v: Map[T, Double], maybeNorm: Option[Double] = None): Map[T, Double] = {

val norm = maybeNorm.getOrElse(CosineSimilarityUtil.logNorm(v))

applyNorm(v, norm)

}

/\*\*

\* Normalize the weight vector with log norm. If the input weight vector

\* is empty or its norm is 0, return the original array.

\*

\* @param v an array of weights

\*

\* @return an array with log normalized weight, in the same order as v

\* \*/

def logNormalizeArray(

v: Array[Double],

maybeNorm: Option[Double] = None

): Array[Double] = {

val norm = maybeNorm.getOrElse(CosineSimilarityUtil.logNormArray(v))

applyNormArray(v, norm)

}

/\*\*

\* Normalize the weight vector with exponentially scaled norm. If the input weight vector

\* is empty or its norm is 0, return the original map.

\*

\* @param v a map from cluster id to its weight

\* @param exponent the exponent we apply to the weight vector's norm

\*

\* @return a map with exp scaled normalized weight

\* \*/

def expScaledNormalize[T](

v: Map[T, Double],

exponent: Option[Double] = None,

maybeNorm: Option[Double] = None

): Map[T, Double] = {

val norm = maybeNorm.getOrElse(CosineSimilarityUtil.expScaledNorm(v, exponent.getOrElse(0.3)))

applyNorm(v, norm)

}

/\*\*

\* Normalize the weight vector with exponentially scaled norm. If the input weight vector

\* is empty or its norm is 0, return the original map.

\*

\* @param v an array of weights

\* @param exponent the exponent we apply to the weight vector's norm

\*

\* @return an array with exp scaled normalized weight, in the same order as v

\* \*/

def expScaledNormalizeArray(

v: Array[Double],

exponent: Double,

maybeNorm: Option[Double] = None

): Array[Double] = {

val norm = maybeNorm.getOrElse(CosineSimilarityUtil.expScaledNormArray(v, exponent))

applyNormArray(v, norm)

}

/\*\*

\* Given two sparse vectors, calculate its dot product.

\*

\* @param v1 the first map from cluster id to its weight

\* @param v2 the second map from cluster id to its weight

\*

\* @return the dot product of above two sparse vector

\*/

def dotProduct[T](v1: Map[T, Double], v2: Map[T, Double]): Double = {

val comparer = v1.size - v2.size

val smaller = if (comparer > 0) v2 else v1

val bigger = if (comparer > 0) v1 else v2

smaller.foldLeft(0.0) {

case (sum, (id, value)) =>

sum + bigger.getOrElse(id, 0.0) \* value

}

}

/\*\*

\* Given two sparse vectors, calculate its dot product.

\*

\* @param v1C an array of cluster ids. Must be sorted in ascending order

\* @param v1S an array of corresponding cluster scores, of the same length and order as v1c

\* @param v2C an array of cluster ids. Must be sorted in ascending order

\* @param v2S an array of corresponding cluster scores, of the same length and order as v2c

\*

\* @return the dot product of above two sparse vector

\*/

def dotProductForSortedClusterAndScores(

v1C: Array[Int],

v1S: Array[Double],

v2C: Array[Int],

v2S: Array[Double]

): Double = {

require(v1C.size == v1S.size)

require(v2C.size == v2S.size)

var i1 = 0

var i2 = 0

var product: Double = 0.0

while (i1 < v1C.size && i2 < v2C.size) {

if (v1C(i1) == v2C(i2)) {

product += v1S(i1) \* v2S(i2)

i1 += 1

i2 += 1

} else if (v1C(i1) > v2C(i2)) {

// v2 cluster is lower. Increment it to see if the next one matches v1's

i2 += 1

} else {

// v1 cluster is lower. Increment it to see if the next one matches v2's

i1 += 1

}

}

product

}

}