package com.twitter.simclusters\_v2.common

import com.twitter.simclusters\_v2.thriftscala.SimClusterWithScore

import com.twitter.simclusters\_v2.thriftscala.{SimClustersEmbedding => ThriftSimClustersEmbedding}

import scala.collection.mutable

import scala.language.implicitConversions

import scala.util.hashing.MurmurHash3.arrayHash

import scala.util.hashing.MurmurHash3.productHash

import scala.math.\_

/\*\*

\* A representation of a SimClusters Embedding, designed for low memory footprint and performance.

\* For services that cache millions of embeddings, we found this to significantly reduce allocations,

\* memory footprint and overall performance.

\*

\* Embedding data is stored in pre-sorted arrays rather than structures which use a lot of pointers

\* (e.g. Map). A minimal set of lazily-constructed intermediate data is kept.

\*

\* Be wary of adding further `val` or `lazy val`s to this class; materializing and storing more data

\* on these objects could significantly affect in-memory cache performance.

\*

\* Also, if you are using this code in a place where you care about memory footprint, be careful

\* not to materialize any of the lazy vals unless you need them.

\*/

sealed trait SimClustersEmbedding extends Equals {

import SimClustersEmbedding.\_

/\*\*

\* Any compliant implementation of the SimClustersEmbedding trait must ensure that:

\* - the cluster and score arrays are ordered as described below

\* - the cluster and score arrays are treated as immutable (.hashCode is memoized)

\* - the size of all cluster and score arrays is the same

\* - all cluster scores are > 0

\* - cluster ids are unique

\*/

// In descending score order - this is useful for truncation, where we care most about the highest scoring elements

private[simclusters\_v2] val clusterIds: Array[ClusterId]

private[simclusters\_v2] val scores: Array[Double]

// In ascending cluster order. This is useful for operations where we try to find the same cluster in another embedding, e.g. dot product

private[simclusters\_v2] val sortedClusterIds: Array[ClusterId]

private[simclusters\_v2] val sortedScores: Array[Double]

/\*\*

\* Build and return a Set of all clusters in this embedding

\*/

lazy val clusterIdSet: Set[ClusterId] = sortedClusterIds.toSet

/\*\*

\* Build and return Seq representation of this embedding

\*/

lazy val embedding: Seq[(ClusterId, Double)] =

sortedClusterIds.zip(sortedScores).sortBy(-\_.\_2).toSeq

/\*\*

\* Build and return a Map representation of this embedding

\*/

lazy val map: Map[ClusterId, Double] = sortedClusterIds.zip(sortedScores).toMap

lazy val l1norm: Double = CosineSimilarityUtil.l1NormArray(sortedScores)

lazy val l2norm: Double = CosineSimilarityUtil.normArray(sortedScores)

lazy val logNorm: Double = CosineSimilarityUtil.logNormArray(sortedScores)

lazy val expScaledNorm: Double =

CosineSimilarityUtil.expScaledNormArray(sortedScores, DefaultExponent)

/\*\*

\* The L2 Normalized Embedding. Optimize for Cosine Similarity Calculation.

\*/

lazy val normalizedSortedScores: Array[Double] =

CosineSimilarityUtil.applyNormArray(sortedScores, l2norm)

lazy val logNormalizedSortedScores: Array[Double] =

CosineSimilarityUtil.applyNormArray(sortedScores, logNorm)

lazy val expScaledNormalizedSortedScores: Array[Double] =

CosineSimilarityUtil.applyNormArray(sortedScores, expScaledNorm)

/\*\*

\* The Standard Deviation of an Embedding.

\*/

lazy val std: Double = {

if (scores.isEmpty) {

0.0

} else {

val sum = scores.sum

val mean = sum / scores.length

var variance: Double = 0.0

for (i <- scores.indices) {

val v = scores(i) - mean

variance += (v \* v)

}

math.sqrt(variance / scores.length)

}

}

/\*\*

\* Return the score of a given clusterId.

\*/

def get(clusterId: ClusterId): Option[Double] = {

var i = 0

while (i < sortedClusterIds.length) {

val thisId = sortedClusterIds(i)

if (clusterId == thisId) return Some(sortedScores(i))

if (thisId > clusterId) return None

i += 1

}

None

}

/\*\*

\* Return the score of a given clusterId. If not exist, return default.

\*/

def getOrElse(clusterId: ClusterId, default: Double = 0.0): Double = {

require(default >= 0.0)

var i = 0

while (i < sortedClusterIds.length) {

val thisId = sortedClusterIds(i)

if (clusterId == thisId) return sortedScores(i)

if (thisId > clusterId) return default

i += 1

}

default

}

/\*\*

\* Return the cluster ids

\*/

def getClusterIds(): Array[ClusterId] = clusterIds

/\*\*

\* Return the cluster ids with the highest scores

\*/

def topClusterIds(size: Int): Seq[ClusterId] = clusterIds.take(size)

/\*\*

\* Return true if this embedding contains a given clusterId

\*/

def contains(clusterId: ClusterId): Boolean = clusterIdSet.contains(clusterId)

def sum(another: SimClustersEmbedding): SimClustersEmbedding = {

if (another.isEmpty) this

else if (this.isEmpty) another

else {

var i1 = 0

var i2 = 0

val l = scala.collection.mutable.ArrayBuffer.empty[(Int, Double)]

while (i1 < sortedClusterIds.length && i2 < another.sortedClusterIds.length) {

if (sortedClusterIds(i1) == another.sortedClusterIds(i2)) {

l += Tuple2(sortedClusterIds(i1), sortedScores(i1) + another.sortedScores(i2))

i1 += 1

i2 += 1

} else if (sortedClusterIds(i1) > another.sortedClusterIds(i2)) {

l += Tuple2(another.sortedClusterIds(i2), another.sortedScores(i2))

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

i2 += 1

} else {

l += Tuple2(sortedClusterIds(i1), sortedScores(i1))

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

i1 += 1

}

}

if (i1 == sortedClusterIds.length && i2 != another.sortedClusterIds.length)

// this was shorter. Prepend remaining elements from another

l ++= another.sortedClusterIds.drop(i2).zip(another.sortedScores.drop(i2))

else if (i1 != sortedClusterIds.length && i2 == another.sortedClusterIds.length)

// another was shorter. Prepend remaining elements from this

l ++= sortedClusterIds.drop(i1).zip(sortedScores.drop(i1))

SimClustersEmbedding(l)

}

}

def scalarMultiply(multiplier: Double): SimClustersEmbedding = {

require(multiplier > 0.0, "SimClustersEmbedding.scalarMultiply requires multiplier > 0.0")

DefaultSimClustersEmbedding(

clusterIds,

scores.map(\_ \* multiplier),

sortedClusterIds,

sortedScores.map(\_ \* multiplier)

)

}

def scalarDivide(divisor: Double): SimClustersEmbedding = {

require(divisor > 0.0, "SimClustersEmbedding.scalarDivide requires divisor > 0.0")

DefaultSimClustersEmbedding(

clusterIds,

scores.map(\_ / divisor),

sortedClusterIds,

sortedScores.map(\_ / divisor)

)

}

def dotProduct(another: SimClustersEmbedding): Double = {

CosineSimilarityUtil.dotProductForSortedClusterAndScores(

sortedClusterIds,

sortedScores,

another.sortedClusterIds,

another.sortedScores)

}

def cosineSimilarity(another: SimClustersEmbedding): Double = {

CosineSimilarityUtil.dotProductForSortedClusterAndScores(

sortedClusterIds,

normalizedSortedScores,

another.sortedClusterIds,

another.normalizedSortedScores)

}

def logNormCosineSimilarity(another: SimClustersEmbedding): Double = {

CosineSimilarityUtil.dotProductForSortedClusterAndScores(

sortedClusterIds,

logNormalizedSortedScores,

another.sortedClusterIds,

another.logNormalizedSortedScores)

}

def expScaledCosineSimilarity(another: SimClustersEmbedding): Double = {

CosineSimilarityUtil.dotProductForSortedClusterAndScores(

sortedClusterIds,

expScaledNormalizedSortedScores,

another.sortedClusterIds,

another.expScaledNormalizedSortedScores)

}

/\*\*

\* Return true if this is an empty embedding

\*/

def isEmpty: Boolean = sortedClusterIds.isEmpty

/\*\*

\* Return the Jaccard Similarity Score between two embeddings.

\* Note: this implementation should be optimized if we start to use it in production

\*/

def jaccardSimilarity(another: SimClustersEmbedding): Double = {

if (this.isEmpty || another.isEmpty) {

0.0

} else {

val intersect = clusterIdSet.intersect(another.clusterIdSet).size

val union = clusterIdSet.union(another.clusterIdSet).size

intersect.toDouble / union

}

}

/\*\*

\* Return the Fuzzy Jaccard Similarity Score between two embeddings.

\* Treat each Simclusters embedding as fuzzy set, calculate the fuzzy set similarity

\* metrics of two embeddings

\*

\* Paper 2.2.1: https://openreview.net/pdf?id=SkxXg2C5FX

\*/

def fuzzyJaccardSimilarity(another: SimClustersEmbedding): Double = {

if (this.isEmpty || another.isEmpty) {

0.0

} else {

val v1C = sortedClusterIds

val v1S = sortedScores

val v2C = another.sortedClusterIds

val v2S = another.sortedScores

require(v1C.length == v1S.length)

require(v2C.length == v2S.length)

var i1 = 0

var i2 = 0

var numerator = 0.0

var denominator = 0.0

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

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

numerator += min(v1S(i1), v2S(i2))

denominator += max(v1S(i1), v2S(i2))

i1 += 1

i2 += 1

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

denominator += v2S(i2)

i2 += 1

} else {

denominator += v1S(i1)

i1 += 1

}

}

while (i1 < v1C.length) {

denominator += v1S(i1)

i1 += 1

}

while (i2 < v2C.length) {

denominator += v2S(i2)

i2 += 1

}

numerator / denominator

}

}

/\*\*

\* Return the Euclidean Distance Score between two embeddings.

\* Note: this implementation should be optimized if we start to use it in production

\*/

def euclideanDistance(another: SimClustersEmbedding): Double = {

val unionClusters = clusterIdSet.union(another.clusterIdSet)

val variance = unionClusters.foldLeft(0.0) {

case (sum, clusterId) =>

val distance = math.abs(this.getOrElse(clusterId) - another.getOrElse(clusterId))

sum + distance \* distance

}

math.sqrt(variance)

}

/\*\*

\* Return the Manhattan Distance Score between two embeddings.

\* Note: this implementation should be optimized if we start to use it in production

\*/

def manhattanDistance(another: SimClustersEmbedding): Double = {

val unionClusters = clusterIdSet.union(another.clusterIdSet)

unionClusters.foldLeft(0.0) {

case (sum, clusterId) =>

sum + math.abs(this.getOrElse(clusterId) - another.getOrElse(clusterId))

}

}

/\*\*

\* Return the number of overlapping clusters between two embeddings.

\*/

def overlappingClusters(another: SimClustersEmbedding): Int = {

var i1 = 0

var i2 = 0

var count = 0

while (i1 < sortedClusterIds.length && i2 < another.sortedClusterIds.length) {

if (sortedClusterIds(i1) == another.sortedClusterIds(i2)) {

count += 1

i1 += 1

i2 += 1

} else if (sortedClusterIds(i1) > another.sortedClusterIds(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

}

}

count

}

/\*\*

\* Return the largest product cluster scores

\*/

def maxElementwiseProduct(another: SimClustersEmbedding): Double = {

var i1 = 0

var i2 = 0

var maxProduct: Double = 0.0

while (i1 < sortedClusterIds.length && i2 < another.sortedClusterIds.length) {

if (sortedClusterIds(i1) == another.sortedClusterIds(i2)) {

val product = sortedScores(i1) \* another.sortedScores(i2)

if (product > maxProduct) maxProduct = product

i1 += 1

i2 += 1

} else if (sortedClusterIds(i1) > another.sortedClusterIds(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

}

}

maxProduct

}

/\*\*

\* Return a new SimClustersEmbedding with Max Embedding Size.

\*

\* Prefer to truncate on embedding construction where possible. Doing so is cheaper.

\*/

def truncate(size: Int): SimClustersEmbedding = {

if (clusterIds.length <= size) {

this

} else {

val truncatedClusterIds = clusterIds.take(size)

val truncatedScores = scores.take(size)

val (sortedClusterIds, sortedScores) =

truncatedClusterIds.zip(truncatedScores).sortBy(\_.\_1).unzip

DefaultSimClustersEmbedding(

truncatedClusterIds,

truncatedScores,

sortedClusterIds,

sortedScores)

}

}

def toNormalized: SimClustersEmbedding = {

// Additional safety check. Only EmptyEmbedding's l2norm is 0.0.

if (l2norm == 0.0) {

EmptyEmbedding

} else {

this.scalarDivide(l2norm)

}

}

implicit def toThrift: ThriftSimClustersEmbedding = {

ThriftSimClustersEmbedding(

embedding.map {

case (clusterId, score) =>

SimClusterWithScore(clusterId, score)

}

)

}

def canEqual(a: Any): Boolean = a.isInstanceOf[SimClustersEmbedding]

/\* We define equality as having the same clusters and scores.

\* This implementation is arguably incorrect in this case:

\* (1 -> 1.0, 2 -> 0.0) == (1 -> 1.0) // equals returns false

\* However, compliant implementations of SimClustersEmbedding should not include zero-weight

\* clusters, so this implementation should work correctly.

\*/

override def equals(that: Any): Boolean =

that match {

case that: SimClustersEmbedding =>

that.canEqual(this) &&

this.sortedClusterIds.sameElements(that.sortedClusterIds) &&

this.sortedScores.sameElements(that.sortedScores)

case \_ => false

}

/\*\*

\* hashcode implementation based on the contents of the embedding. As a lazy val, this relies on

\* the embedding contents being immutable.

\*/

override lazy val hashCode: Int = {

/\* Arrays uses object id as hashCode, so different arrays with the same contents hash

\* differently. To provide a stable hash code, we take the same approach as how a

\* `case class(clusters: Seq[Int], scores: Seq[Double])` would be hashed. See

\* ScalaRunTime.\_hashCode and MurmurHash3.productHash

\* https://github.com/scala/scala/blob/2.12.x/src/library/scala/runtime/ScalaRunTime.scala#L167

\* https://github.com/scala/scala/blob/2.12.x/src/library/scala/util/hashing/MurmurHash3.scala#L64

\*

\* Note that the hashcode is arguably incorrect in this case:

\* (1 -> 1.0, 2 -> 0.0).hashcode == (1 -> 1.0).hashcode // returns false

\* However, compliant implementations of SimClustersEmbedding should not include zero-weight

\* clusters, so this implementation should work correctly.

\*/

productHash((arrayHash(sortedClusterIds), arrayHash(sortedScores)))

}

}

object SimClustersEmbedding {

val EmptyEmbedding: SimClustersEmbedding =

DefaultSimClustersEmbedding(Array.empty, Array.empty, Array.empty, Array.empty)

val DefaultExponent: Double = 0.3

// Descending by score then ascending by ClusterId

implicit val order: Ordering[(ClusterId, Double)] =

(a: (ClusterId, Double), b: (ClusterId, Double)) => {

b.\_2 compare a.\_2 match {

case 0 => a.\_1 compare b.\_1

case c => c

}

}

/\*\*

\* Constructors

\*

\* These constructors:

\* - do not make assumptions about the ordering of the cluster/scores.

\* - do assume that cluster ids are unique

\* - ignore (drop) any cluster whose score is <= 0

\*/

def apply(embedding: (ClusterId, Double)\*): SimClustersEmbedding =

buildDefaultSimClustersEmbedding(embedding)

def apply(embedding: Iterable[(ClusterId, Double)]): SimClustersEmbedding =

buildDefaultSimClustersEmbedding(embedding)

def apply(embedding: Iterable[(ClusterId, Double)], size: Int): SimClustersEmbedding =

buildDefaultSimClustersEmbedding(embedding, truncate = Some(size))

implicit def apply(thriftEmbedding: ThriftSimClustersEmbedding): SimClustersEmbedding =

buildDefaultSimClustersEmbedding(thriftEmbedding.embedding.map(\_.toTuple))

def apply(thriftEmbedding: ThriftSimClustersEmbedding, truncate: Int): SimClustersEmbedding =

buildDefaultSimClustersEmbedding(

thriftEmbedding.embedding.map(\_.toTuple),

truncate = Some(truncate))

private def buildDefaultSimClustersEmbedding(

embedding: Iterable[(ClusterId, Double)],

truncate: Option[Int] = None

): SimClustersEmbedding = {

val truncatedIdAndScores = {

val idsAndScores = embedding.filter(\_.\_2 > 0.0).toArray.sorted(order)

truncate match {

case Some(t) => idsAndScores.take(t)

case \_ => idsAndScores

}

}

if (truncatedIdAndScores.isEmpty) {

EmptyEmbedding

} else {

val (clusterIds, scores) = truncatedIdAndScores.unzip

val (sortedClusterIds, sortedScores) = truncatedIdAndScores.sortBy(\_.\_1).unzip

DefaultSimClustersEmbedding(clusterIds, scores, sortedClusterIds, sortedScores)

}

}

/\*\* \*\*\*\*\* Aggregation Methods \*\*\*\*\*\*/

/\*\*

\* A high performance version of Sum a list of SimClustersEmbeddings.

\* Suggest using in Online Services to avoid the unnecessary GC.

\* For offline or streaming. Please check [[SimClustersEmbeddingMonoid]]

\*/

def sum(simClustersEmbeddings: Iterable[SimClustersEmbedding]): SimClustersEmbedding = {

if (simClustersEmbeddings.isEmpty) {

EmptyEmbedding

} else {

val sum = simClustersEmbeddings.foldLeft(mutable.Map[ClusterId, Double]()) {

(sum, embedding) =>

for (i <- embedding.sortedClusterIds.indices) {

val clusterId = embedding.sortedClusterIds(i)

sum.put(clusterId, embedding.sortedScores(i) + sum.getOrElse(clusterId, 0.0))

}

sum

}

SimClustersEmbedding(sum)

}

}

/\*\*

\* Support a fixed size SimClustersEmbedding Sum

\*/

def sum(

simClustersEmbeddings: Iterable[SimClustersEmbedding],

maxSize: Int

): SimClustersEmbedding = {

sum(simClustersEmbeddings).truncate(maxSize)

}

/\*\*

\* A high performance version of Mean a list of SimClustersEmbeddings.

\* Suggest using in Online Services to avoid the unnecessary GC.

\*/

def mean(simClustersEmbeddings: Iterable[SimClustersEmbedding]): SimClustersEmbedding = {

if (simClustersEmbeddings.isEmpty) {

EmptyEmbedding

} else {

sum(simClustersEmbeddings).scalarDivide(simClustersEmbeddings.size)

}

}

/\*\*

\* Support a fixed size SimClustersEmbedding Mean

\*/

def mean(

simClustersEmbeddings: Iterable[SimClustersEmbedding],

maxSize: Int

): SimClustersEmbedding = {

mean(simClustersEmbeddings).truncate(maxSize)

}

}

case class DefaultSimClustersEmbedding(

override val clusterIds: Array[ClusterId],

override val scores: Array[Double],

override val sortedClusterIds: Array[ClusterId],

override val sortedScores: Array[Double])

extends SimClustersEmbedding {

override def toString: String =

s"DefaultSimClustersEmbedding(${clusterIds.zip(scores).mkString(",")})"

}

object DefaultSimClustersEmbedding {

// To support existing code which builds embeddings from a Seq

def apply(embedding: Seq[(ClusterId, Double)]): SimClustersEmbedding = SimClustersEmbedding(

embedding)

}