package com.twitter.ann.common

import com.google.common.collect.ImmutableBiMap

import com.twitter.ann.common.EmbeddingType.\_

import com.twitter.ann.common.thriftscala.DistanceMetric

import com.twitter.ann.common.thriftscala.{CosineDistance => ServiceCosineDistance}

import com.twitter.ann.common.thriftscala.{Distance => ServiceDistance}

import com.twitter.ann.common.thriftscala.{InnerProductDistance => ServiceInnerProductDistance}

import com.twitter.ann.common.thriftscala.{EditDistance => ServiceEditDistance}

import com.twitter.ann.common.thriftscala.{L2Distance => ServiceL2Distance}

import com.twitter.bijection.Injection

import scala.util.Failure

import scala.util.Success

import scala.util.Try

// Ann distance metrics

trait Distance[D] extends Any with Ordered[D] {

def distance: Float

}

case class L2Distance(distance: Float) extends AnyVal with Distance[L2Distance] {

override def compare(that: L2Distance): Int =

Ordering.Float.compare(this.distance, that.distance)

}

case class CosineDistance(distance: Float) extends AnyVal with Distance[CosineDistance] {

override def compare(that: CosineDistance): Int =

Ordering.Float.compare(this.distance, that.distance)

}

case class InnerProductDistance(distance: Float)

extends AnyVal

with Distance[InnerProductDistance] {

override def compare(that: InnerProductDistance): Int =

Ordering.Float.compare(this.distance, that.distance)

}

case class EditDistance(distance: Float) extends AnyVal with Distance[EditDistance] {

override def compare(that: EditDistance): Int =

Ordering.Float.compare(this.distance, that.distance)

}

object Metric {

private[this] val thriftMetricMapping = ImmutableBiMap.of(

L2,

DistanceMetric.L2,

Cosine,

DistanceMetric.Cosine,

InnerProduct,

DistanceMetric.InnerProduct,

Edit,

DistanceMetric.EditDistance

)

def fromThrift(metric: DistanceMetric): Metric[\_ <: Distance[\_]] = {

thriftMetricMapping.inverse().get(metric)

}

def toThrift(metric: Metric[\_ <: Distance[\_]]): DistanceMetric = {

thriftMetricMapping.get(metric)

}

def fromString(metricName: String): Metric[\_ <: Distance[\_]]

with Injection[\_, ServiceDistance] = {

metricName match {

case "Cosine" => Cosine

case "L2" => L2

case "InnerProduct" => InnerProduct

case "EditDistance" => Edit

case \_ =>

throw new IllegalArgumentException(s"No Metric with the name $metricName")

}

}

}

sealed trait Metric[D <: Distance[D]] {

def distance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): D

def absoluteDistance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): Float

def fromAbsoluteDistance(distance: Float): D

}

case object L2 extends Metric[L2Distance] with Injection[L2Distance, ServiceDistance] {

override def distance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): L2Distance = {

fromAbsoluteDistance(MetricUtil.l2distance(embedding1, embedding2).toFloat)

}

override def fromAbsoluteDistance(distance: Float): L2Distance = {

L2Distance(distance)

}

override def absoluteDistance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): Float = distance(embedding1, embedding2).distance

override def apply(scalaDistance: L2Distance): ServiceDistance = {

ServiceDistance.L2Distance(ServiceL2Distance(scalaDistance.distance))

}

override def invert(serviceDistance: ServiceDistance): Try[L2Distance] = {

serviceDistance match {

case ServiceDistance.L2Distance(l2Distance) =>

Success(L2Distance(l2Distance.distance.toFloat))

case distance =>

Failure(new IllegalArgumentException(s"Expected an l2 distance but got $distance"))

}

}

}

case object Cosine extends Metric[CosineDistance] with Injection[CosineDistance, ServiceDistance] {

override def distance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): CosineDistance = {

fromAbsoluteDistance(1 - MetricUtil.cosineSimilarity(embedding1, embedding2))

}

override def fromAbsoluteDistance(distance: Float): CosineDistance = {

CosineDistance(distance)

}

override def absoluteDistance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): Float = distance(embedding1, embedding2).distance

override def apply(scalaDistance: CosineDistance): ServiceDistance = {

ServiceDistance.CosineDistance(ServiceCosineDistance(scalaDistance.distance))

}

override def invert(serviceDistance: ServiceDistance): Try[CosineDistance] = {

serviceDistance match {

case ServiceDistance.CosineDistance(cosineDistance) =>

Success(CosineDistance(cosineDistance.distance.toFloat))

case distance =>

Failure(new IllegalArgumentException(s"Expected a cosine distance but got $distance"))

}

}

}

case object InnerProduct

extends Metric[InnerProductDistance]

with Injection[InnerProductDistance, ServiceDistance] {

override def distance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): InnerProductDistance = {

fromAbsoluteDistance(1 - MetricUtil.dot(embedding1, embedding2))

}

override def fromAbsoluteDistance(distance: Float): InnerProductDistance = {

InnerProductDistance(distance)

}

override def absoluteDistance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): Float = distance(embedding1, embedding2).distance

override def apply(scalaDistance: InnerProductDistance): ServiceDistance = {

ServiceDistance.InnerProductDistance(ServiceInnerProductDistance(scalaDistance.distance))

}

override def invert(

serviceDistance: ServiceDistance

): Try[InnerProductDistance] = {

serviceDistance match {

case ServiceDistance.InnerProductDistance(cosineDistance) =>

Success(InnerProductDistance(cosineDistance.distance.toFloat))

case distance =>

Failure(

new IllegalArgumentException(s"Expected a inner product distance but got $distance")

)

}

}

}

case object Edit extends Metric[EditDistance] with Injection[EditDistance, ServiceDistance] {

private def intDistance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector,

pos1: Int,

pos2: Int,

precomputedDistances: scala.collection.mutable.Map[(Int, Int), Int]

): Int = {

// return the remaining characters of other String

if (pos1 == 0) return pos2

if (pos2 == 0) return pos1

// To check if the recursive tree

// for given n & m has already been executed

precomputedDistances.getOrElse(

(pos1, pos2), {

// We might want to change this so that capitals are considered the same.

// Also maybe some characters that look similar should also be the same.

val computed = if (embedding1(pos1 - 1) == embedding2(pos2 - 1)) {

intDistance(embedding1, embedding2, pos1 - 1, pos2 - 1, precomputedDistances)

} else { // If characters are nt equal, we need to

// find the minimum cost out of all 3 operations.

val insert = intDistance(embedding1, embedding2, pos1, pos2 - 1, precomputedDistances)

val del = intDistance(embedding1, embedding2, pos1 - 1, pos2, precomputedDistances)

val replace =

intDistance(embedding1, embedding2, pos1 - 1, pos2 - 1, precomputedDistances)

1 + Math.min(insert, Math.min(del, replace))

}

precomputedDistances.put((pos1, pos2), computed)

computed

}

)

}

override def distance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): EditDistance = {

val editDistance = intDistance(

embedding1,

embedding2,

embedding1.length,

embedding2.length,

scala.collection.mutable.Map[(Int, Int), Int]()

)

EditDistance(editDistance)

}

override def fromAbsoluteDistance(distance: Float): EditDistance = {

EditDistance(distance.toInt)

}

override def absoluteDistance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): Float = distance(embedding1, embedding2).distance

override def apply(scalaDistance: EditDistance): ServiceDistance = {

ServiceDistance.EditDistance(ServiceEditDistance(scalaDistance.distance.toInt))

}

override def invert(

serviceDistance: ServiceDistance

): Try[EditDistance] = {

serviceDistance match {

case ServiceDistance.EditDistance(cosineDistance) =>

Success(EditDistance(cosineDistance.distance.toFloat))

case distance =>

Failure(

new IllegalArgumentException(s"Expected a inner product distance but got $distance")

)

}

}

}

object MetricUtil {

private[ann] def dot(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): Float = {

math.dotProduct(embedding1, embedding2)

}

private[ann] def l2distance(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): Double = {

math.l2Distance(embedding1, embedding2)

}

private[ann] def cosineSimilarity(

embedding1: EmbeddingVector,

embedding2: EmbeddingVector

): Float = {

math.cosineSimilarity(embedding1, embedding2).toFloat

}

private[ann] def norm(

embedding: EmbeddingVector

): EmbeddingVector = {

math.normalize(embedding)

}

}