package com.twitter.ann.faiss

import com.twitter.ann.common.Cosine

import com.twitter.ann.common.Distance

import com.twitter.ann.common.EmbeddingType.EmbeddingVector

import com.twitter.ann.common.Metric

import com.twitter.ann.common.NeighborWithDistance

import com.twitter.ann.common.Queryable

import com.twitter.ml.api.embedding.EmbeddingMath

import com.twitter.search.common.file.AbstractFile

import com.twitter.search.common.file.FileUtils

import com.twitter.util.Future

import com.twitter.util.logging.Logging

import java.io.File

import java.util.concurrent.locks.ReentrantReadWriteLock

object QueryableIndexAdapter extends Logging {

// swigfaiss.read\_index doesn't support hdfs files, hence a copy to temporary directory

def loadJavaIndex(directory: AbstractFile): Index = {

val indexFile = directory.getChild("faiss.index")

val tmpFile = File.createTempFile("faiss.index", ".tmp")

val tmpAbstractFile = FileUtils.getFileHandle(tmpFile.toString)

indexFile.copyTo(tmpAbstractFile)

val index = swigfaiss.read\_index(tmpAbstractFile.getPath)

if (!tmpFile.delete()) {

error(s"Failed to delete ${tmpFile.toString}")

}

index

}

}

trait QueryableIndexAdapter[T, D <: Distance[D]] extends Queryable[T, FaissParams, D] {

this: Logging =>

private val MAX\_COSINE\_DISTANCE = 1f

protected def index: Index

protected val metric: Metric[D]

protected val dimension: Int

private def maybeNormalizeEmbedding(embeddingVector: EmbeddingVector): EmbeddingVector = {

// There is no direct support for Cosine, but l2norm + ip == Cosine by definition

if (metric == Cosine) {

EmbeddingMath.Float.normalize(embeddingVector)

} else {

embeddingVector

}

}

private def maybeTranslateToCosineDistanceInplace(array: floatArray, len: Int): Unit = {

// Faiss reports Cosine similarity while we need Cosine distance.

if (metric == Cosine) {

for (index <- 0 until len) {

val similarity = array.getitem(index)

if (similarity < 0 || similarity > 1) {

warn(s"Expected similarity to be between 0 and 1, got ${similarity} instead")

array.setitem(index, MAX\_COSINE\_DISTANCE)

} else {

array.setitem(index, 1 - similarity)

}

}

}

}

private val paramsLock = new ReentrantReadWriteLock()

private var currentParams: Option[String] = None

// Assume that parameters rarely change and try read lock first

private def ensuringParams[R](parameterString: String, f: () => R): R = {

paramsLock.readLock().lock()

try {

if (currentParams.contains(parameterString)) {

return f()

}

} finally {

paramsLock.readLock().unlock()

}

paramsLock.writeLock().lock()

try {

currentParams = Some(parameterString)

new ParameterSpace().set\_index\_parameters(index, parameterString)

f()

} finally {

paramsLock.writeLock().unlock()

}

}

def replaceIndex(f: () => Unit): Unit = {

paramsLock.writeLock().lock()

try {

currentParams = None

f()

} finally {

paramsLock.writeLock().unlock()

}

}

def query(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: FaissParams

): Future[List[T]] = {

Future.value(

ensuringParams(

runtimeParams.toLibraryString,

() => {

val distances = new floatArray(numOfNeighbors)

val indexes = new LongVector()

indexes.resize(numOfNeighbors)

val normalizedEmbedding = maybeNormalizeEmbedding(embedding)

index.search(

// Number of query embeddings

1,

// Array of query embeddings

toFloatArray(normalizedEmbedding).cast(),

// Number of neighbours to return

numOfNeighbors,

// Location to store neighbour distances

distances.cast(),

// Location to store neighbour identifiers

indexes

)

// This is a shortcoming of current swig bindings

// Nothing prevents JVM from freeing distances while inside index.search

// This might be removed once we start passing FloatVector

// Why java.lang.ref.Reference.reachabilityFence doesn't compile?

debug(distances)

toSeq(indexes, numOfNeighbors).toList.asInstanceOf[List[T]]

}

))

}

def queryWithDistance(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: FaissParams

): Future[List[NeighborWithDistance[T, D]]] = {

Future.value(

ensuringParams(

runtimeParams.toLibraryString,

() => {

val distances = new floatArray(numOfNeighbors)

val indexes = new LongVector()

indexes.resize(numOfNeighbors)

val normalizedEmbedding = maybeNormalizeEmbedding(embedding)

index.search(

// Number of query embeddings

1,

// Array of query embeddings

toFloatArray(normalizedEmbedding).cast(),

// Number of neighbours to return

numOfNeighbors,

// Location to store neighbour distances

distances.cast(),

// Location to store neighbour identifiers

indexes

)

val ids = toSeq(indexes, numOfNeighbors).toList.asInstanceOf[List[T]]

maybeTranslateToCosineDistanceInplace(distances, numOfNeighbors)

val distancesSeq = toSeq(distances, numOfNeighbors)

ids.zip(distancesSeq).map {

case (id, distance) =>

NeighborWithDistance(id, metric.fromAbsoluteDistance(distance))

}

}

))

}

private def toFloatArray(emb: EmbeddingVector): floatArray = {

val nativeArray = new floatArray(emb.length)

for ((value, aIdx) <- emb.iterator.zipWithIndex) {

nativeArray.setitem(aIdx, value)

}

nativeArray

}

private def toSeq(vector: LongVector, len: Long): Seq[Long] = {

(0L until len).map(vector.at)

}

private def toSeq(array: floatArray, len: Int): Seq[Float] = {

(0 until len).map(array.getitem)

}

}