package com.twitter.ann.faiss

import com.google.common.base.Preconditions

import com.twitter.ann.common.Cosine

import com.twitter.ann.common.Distance

import com.twitter.ann.common.EntityEmbedding

import com.twitter.ann.common.IndexOutputFile

import com.twitter.ann.common.InnerProduct

import com.twitter.ann.common.L2

import com.twitter.ann.common.Metric

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

import com.twitter.scalding.Execution

import com.twitter.scalding.TypedPipe

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

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

import com.twitter.util.logging.Logging

import java.io.File

import scala.util.Random

trait FaissIndexer extends Logging {

/\*\*

\* Produce faiss index file specified by factory string

\*

\* @param pipe Embeddings to be indexed

\* @param sampleRate Fraction of embeddings used for training. Regardless of this parameter, all embeddings are present in the output.

\* @param factoryString Faiss factory string, see https://github.com/facebookresearch/faiss/wiki/The-index-factory

\* @param metric Metric to use

\* @param outputDirectory Directory where \_SUCCESS and faiss.index will be written.

\*/

def build[D <: Distance[D]](

pipe: TypedPipe[EntityEmbedding[Long]],

sampleRate: Float,

factoryString: String,

metric: Metric[D],

outputDirectory: AbstractFile

): Execution[Unit] = {

outputDirectory.mkdirs()

Preconditions.checkState(

outputDirectory.canRead,

"Failed to create parent directories for %s",

outputDirectory.toString)

val maybeNormalizedPipe = if (l2Normalize(metric)) {

pipe.map { idAndEmbedding =>

EntityEmbedding(idAndEmbedding.id, EmbeddingMath.Float.normalize(idAndEmbedding.embedding))

}

} else {

pipe

}

maybeNormalizedPipe.toIterableExecution.flatMap { annEmbeddings =>

logger.info(s"${factoryString}")

val t1 = System.nanoTime

buildAndWriteFaissIndex(

Random.shuffle(annEmbeddings),

sampleRate,

factoryString,

metric,

new IndexOutputFile(outputDirectory))

val duration = (System.nanoTime - t1) / 1e9d

logger.info(s"It took ${duration}s to build and index")

Execution.unit

}

}

def buildAndWriteFaissIndex[D <: Distance[D]](

entities: Iterable[EntityEmbedding[Long]],

sampleRate: Float,

factoryString: String,

metricType: Metric[D],

outputDirectory: IndexOutputFile

): Unit = {

val metric = parseMetric(metricType)

val datasetSize = entities.size.toLong

val dimensions = entities.head.embedding.length

logger.info(s"There are $datasetSize embeddings")

logger.info(s"Faiss compile options are ${swigfaiss.get\_compile\_options()}")

logger.info(s"OMP threads count is ${swigfaiss.omp\_get\_max\_threads()}")

val index = swigfaiss.index\_factory(dimensions, factoryString, metric)

index.setVerbose(true)

val idMap = new IndexIDMap(index)

val trainingSetSize = Math.min(datasetSize, Math.round(datasetSize \* sampleRate))

val ids = toIndexVector(entities)

val fullDataset = toFloatVector(dimensions, entities)

logger.info("Finished bridging full dataset")

idMap.train(trainingSetSize, fullDataset.data())

logger.info("Finished training")

idMap.add\_with\_ids(datasetSize, fullDataset.data(), ids)

logger.info("Added data to the index")

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

swigfaiss.write\_index(idMap, tmpFile.toString)

logger.info(s"Wrote to tmp file ${tmpFile.toString}")

copyToOutputAndCreateSuccess(FileUtils.getFileHandle(tmpFile.toString), outputDirectory)

logger.info("Copied file")

}

private def copyToOutputAndCreateSuccess(

tmpFile: AbstractFile,

outputDirectory: IndexOutputFile

) = {

val outputFile = outputDirectory.createFile("faiss.index")

logger.info(s"Final output file is ${outputFile.getPath()}")

outputFile.copyFrom(tmpFile.getByteSource.openStream())

outputDirectory.createSuccessFile()

}

private def toFloatVector(

dimensions: Int,

entities: Iterable[EntityEmbedding[Long]]

): FloatVector = {

require(entities.nonEmpty)

val vector = new FloatVector()

vector.reserve(dimensions.toLong \* entities.size.toLong)

for (entity <- entities) {

for (value <- entity.embedding) {

vector.push\_back(value)

}

}

vector

}

private def toIndexVector(embeddings: Iterable[EntityEmbedding[Long]]): LongVector = {

require(embeddings.nonEmpty)

val vector = new LongVector()

vector.reserve(embeddings.size)

for (embedding <- embeddings) {

vector.push\_back(embedding.id)

}

vector

}

private def parseMetric[D <: Distance[D]](metric: Metric[D]): MetricType = metric match {

case L2 => MetricType.METRIC\_L2

case InnerProduct => MetricType.METRIC\_INNER\_PRODUCT

case Cosine => MetricType.METRIC\_INNER\_PRODUCT

case \_ => throw new AbstractMethodError(s"Not implemented for metric ${metric}")

}

private def l2Normalize[D <: Distance[D]](metric: Metric[D]): Boolean = metric match {

case Cosine => true

case \_ => false

}

}

object FaissIndexer extends FaissIndexer {}