package com.twitter.ann.service.loadtest

import com.google.common.annotations.VisibleForTesting

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

import com.twitter.ann.common.thriftscala.AnnQueryService

import com.twitter.ann.common.thriftscala.NearestNeighborQuery

import com.twitter.ann.common.thriftscala.NearestNeighborResult

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

import com.twitter.ann.common.thriftscala.{RuntimeParams => ServiceRuntimeParams}

import com.twitter.ann.common.Distance

import com.twitter.ann.common.EntityEmbedding

import com.twitter.ann.common.Queryable

import com.twitter.ann.common.RuntimeParams

import com.twitter.ann.common.ServiceClientQueryable

import com.twitter.bijection.Injection

import com.twitter.cortex.ml.embeddings.common.EntityKind

import com.twitter.finagle.builder.ClientBuilder

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.mtls.client.MtlsStackClient.MtlsThriftMuxClientSyntax

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.thrift.ClientId

import com.twitter.finagle.Service

import com.twitter.finagle.ThriftMux

import com.twitter.ml.api.embedding.Embedding

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

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

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

import com.twitter.search.common.file.LocalFile

import com.twitter.util.Future

import com.twitter.util.logging.Logger

import java.io.File

import scala.collection.JavaConversions.\_

import scala.collection.mutable

import scala.util.Random

object LoadTestUtils {

lazy val Log = Logger(getClass.getName)

private[this] val LocalPath = "."

private[this] val RNG = new Random(100)

private[loadtest] def getTruthSetMap[Q, I](

directory: String,

queryIdType: String,

indexIdType: String

): Map[Q, Seq[I]] = {

Log.info(s"Loading truth set from ${directory}")

val queryConverter = getKeyConverter[Q](queryIdType)

val indexConverter = getKeyConverter[I](indexIdType)

val res = loadKnnDirFileToMap(

getLocalFileHandle(directory),

// Knn truth file tsv format: [id neighbor:distance neighbor:distance ...]

arr => { arr.map(str => indexConverter(str.substring(0, str.lastIndexOf(":")))).toSeq },

queryConverter

)

assert(res.nonEmpty, s"Must have some something in the truth set ${directory}")

res

}

private[this] def getLocalFileHandle(

directory: String

): AbstractFile = {

val fileHandle = FileUtils.getFileHandle(directory)

if (fileHandle.isInstanceOf[LocalFile]) {

fileHandle

} else {

val localFileHandle =

FileUtils.getFileHandle(s"${LocalPath}${File.separator}${fileHandle.getName}")

fileHandle.copyTo(localFileHandle)

localFileHandle

}

}

private[loadtest] def getEmbeddingsSet[T](

directory: String,

idType: String

): Seq[EntityEmbedding[T]] = {

Log.info(s"Loading embeddings from ${directory}")

val res = loadKnnDirFileToMap(

getLocalFileHandle(directory),

arr => { arr.map(\_.toFloat) },

getKeyConverter[T](idType)

).map { case (key, value) => EntityEmbedding[T](key, Embedding(value.toArray)) }.toSeq

assert(res.nonEmpty, s"Must have some something in the embeddings set ${directory}")

res

}

private[this] def loadKnnDirFileToMap[K, V](

directory: AbstractFile,

f: Array[String] => Seq[V],

converter: String => K

): Map[K, Seq[V]] = {

val map = mutable.HashMap[K, Seq[V]]()

directory

.listFiles(new Filter {

override def accept(file: AbstractFile): Boolean =

file.getName != AbstractFile.SUCCESS\_FILE\_NAME

}).foreach { file =>

asScalaBuffer(file.readLines()).foreach { line =>

addToMapFromKnnString(line, f, map, converter)

}

}

map.toMap

}

// Generating random float with value range bounded between minValue and maxValue

private[loadtest] def getRandomQuerySet(

dimension: Int,

totalQueries: Int,

minValue: Float,

maxValue: Float

): Seq[EmbeddingVector] = {

Log.info(

s"Generating $totalQueries random queries for dimension $dimension with value between $minValue and $maxValue...")

assert(totalQueries > 0, s"Total random queries $totalQueries should be greater than 0")

assert(

maxValue > minValue,

s"Random embedding max value should be greater than min value. min: $minValue max: $maxValue")

(1 to totalQueries).map { \_ =>

val embedding = Array.fill(dimension)(minValue + (maxValue - minValue) \* RNG.nextFloat())

Embedding(embedding)

}

}

private[this] def getKeyConverter[T](idType: String): String => T = {

val converter = idType match {

case "long" =>

(s: String) => s.toLong

case "string" =>

(s: String) => s

case "int" =>

(s: String) => s.toInt

case entityKind =>

(s: String) => EntityKind.getEntityKind(entityKind).stringInjection.invert(s).get

}

converter.asInstanceOf[String => T]

}

private[loadtest] def buildRemoteServiceQueryClient[T, P <: RuntimeParams, D <: Distance[D]](

destination: String,

clientId: String,

statsReceiver: StatsReceiver,

serviceIdentifier: ServiceIdentifier,

runtimeParamInjection: Injection[P, ServiceRuntimeParams],

distanceInjection: Injection[D, ServiceDistance],

indexIdInjection: Injection[T, Array[Byte]]

): Future[Queryable[T, P, D]] = {

val client: AnnQueryService.MethodPerEndpoint = new AnnQueryService.FinagledClient(

service = ClientBuilder()

.reportTo(statsReceiver)

.dest(destination)

.stack(ThriftMux.client.withMutualTls(serviceIdentifier).withClientId(ClientId(clientId)))

.build(),

stats = statsReceiver

)

val service = new Service[NearestNeighborQuery, NearestNeighborResult] {

override def apply(request: NearestNeighborQuery): Future[NearestNeighborResult] =

client.query(request)

}

Future.value(

new ServiceClientQueryable[T, P, D](

service,

runtimeParamInjection,

distanceInjection,

indexIdInjection

)

)

}

// helper method to convert a line in KNN file output format into map

@VisibleForTesting

def addToMapFromKnnString[K, V](

line: String,

f: Array[String] => Seq[V],

map: mutable.HashMap[K, Seq[V]],

converter: String => K

): Unit = {

val items = line.split("\t")

map += converter(items(0)) -> f(items.drop(1))

}

def printResults(

inMemoryBuildRecorder: InMemoryLoadTestBuildRecorder,

queryTimeConfigurations: Seq[QueryTimeConfiguration[\_, \_]]

): Seq[String] = {

val queryTimeConfigStrings = queryTimeConfigurations.map { config =>

config.printResults

}

Seq(

"Build results",

"indexingTimeSecs\ttoQueryableTimeMs\tindexSize",

s"${inMemoryBuildRecorder.indexLatency.inSeconds}\t${inMemoryBuildRecorder.toQueryableLatency.inMilliseconds}\t${inMemoryBuildRecorder.indexSize}",

"Query results",

QueryTimeConfiguration.ResultHeader

) ++ queryTimeConfigStrings

}

}