package com.twitter.cr\_mixer.similarity\_engine

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

import com.twitter.ann.common.thriftscala.Distance

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

import com.twitter.ann.hnsw.HnswCommon

import com.twitter.ann.hnsw.HnswParams

import com.twitter.bijection.Injection

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

import com.twitter.cr\_mixer.model.SimilarityEngineInfo

import com.twitter.cr\_mixer.model.TweetWithScore

import com.twitter.cr\_mixer.similarity\_engine.SimilarityEngine.MemCacheConfig

import com.twitter.cr\_mixer.similarity\_engine.SimilarityEngine.SimilarityEngineConfig

import com.twitter.cr\_mixer.thriftscala.SimilarityEngineType

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.util.StatsUtil

import com.twitter.mediaservices.commons.codec.ArrayByteBufferCodec

import com.twitter.ml.api.thriftscala.{Embedding => ThriftEmbedding}

import com.twitter.ml.featurestore.lib

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

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi.Params

import com.twitter.util.Future

case class HnswANNEngineQuery(

modelId: String,

sourceId: InternalId,

params: Params,

) {

val cacheKey: String = s"${modelId}\_${sourceId.toString}"

}

/\*\*

\* This Engine looks for tweets whose similarity is close to a Source Dense Embedding.

\* Only support Long based embedding lookup. UserId or TweetId.

\*

\* It provides HNSW specific implementations

\*

\* @param memCacheConfigOpt If specified, it will wrap the underlying store with a MemCache layer

\* You should only enable this for cacheable queries, e.x. TweetIds.

\* consumer based UserIds are generally not possible to cache.

\*/

class HnswANNSimilarityEngine(

embeddingStoreLookUpMap: Map[String, ReadableStore[InternalId, ThriftEmbedding]],

annServiceLookUpMap: Map[String, AnnQueryService.MethodPerEndpoint],

globalStats: StatsReceiver,

override val identifier: SimilarityEngineType,

engineConfig: SimilarityEngineConfig,

memCacheConfigOpt: Option[MemCacheConfig[HnswANNEngineQuery]] = None)

extends SimilarityEngine[HnswANNEngineQuery, TweetWithScore] {

private val MaxNumResults: Int = 200

private val ef: Int = 800

private val TweetIdByteInjection: Injection[lib.TweetId, Array[Byte]] = TweetKind.byteInjection

private val scopedStats = globalStats.scope("similarityEngine", identifier.toString)

def getScopedStats: StatsReceiver = scopedStats

private def fetchEmbedding(

query: HnswANNEngineQuery,

): Future[Option[ThriftEmbedding]] = {

val embeddingStore = embeddingStoreLookUpMap.getOrElse(

query.modelId,

throw new IllegalArgumentException(

s"${this.getClass.getSimpleName} ${identifier.toString}: " +

s"ModelId ${query.modelId} does not exist for embeddingStore"

)

)

embeddingStore.get(query.sourceId)

}

private def fetchCandidates(

query: HnswANNEngineQuery,

embedding: ThriftEmbedding

): Future[Seq[TweetWithScore]] = {

val annService = annServiceLookUpMap.getOrElse(

query.modelId,

throw new IllegalArgumentException(

s"${this.getClass.getSimpleName} ${identifier.toString}: " +

s"ModelId ${query.modelId} does not exist for annStore"

)

)

val hnswParams = HnswCommon.RuntimeParamsInjection.apply(HnswParams(ef))

val annQuery =

NearestNeighborQuery(embedding, withDistance = true, hnswParams, MaxNumResults)

annService

.query(annQuery)

.map(

\_.nearestNeighbors

.map { nearestNeighbor =>

val candidateId = TweetIdByteInjection

.invert(ArrayByteBufferCodec.decode(nearestNeighbor.id))

.toOption

.map(\_.tweetId)

(candidateId, nearestNeighbor.distance)

}.collect {

case (Some(candidateId), Some(distance)) =>

TweetWithScore(candidateId, toScore(distance))

})

}

// Convert Distance to a score such that higher scores mean more similar.

def toScore(distance: Distance): Double = {

distance match {

case Distance.EditDistance(editDistance) =>

// (-Infinite, 0.0]

0.0 - editDistance.distance

case Distance.L2Distance(l2Distance) =>

// (-Infinite, 0.0]

0.0 - l2Distance.distance

case Distance.CosineDistance(cosineDistance) =>

// [0.0 - 1.0]

1.0 - cosineDistance.distance

case Distance.InnerProductDistance(innerProductDistance) =>

// (-Infinite, Infinite)

1.0 - innerProductDistance.distance

case Distance.UnknownUnionField(\_) =>

throw new IllegalStateException(

s"${this.getClass.getSimpleName} does not recognize $distance.toString"

)

}

}

private[similarity\_engine] def getEmbeddingAndCandidates(

query: HnswANNEngineQuery

): Future[Option[Seq[TweetWithScore]]] = {

val fetchEmbeddingStat = scopedStats.scope(query.modelId).scope("fetchEmbedding")

val fetchCandidatesStat = scopedStats.scope(query.modelId).scope("fetchCandidates")

for {

embeddingOpt <- StatsUtil.trackOptionStats(fetchEmbeddingStat) { fetchEmbedding(query) }

candidates <- StatsUtil.trackItemsStats(fetchCandidatesStat) {

embeddingOpt match {

case Some(embedding) => fetchCandidates(query, embedding)

case None => Future.Nil

}

}

} yield {

Some(candidates)

}

}

// Add memcache wrapper, if specified

private val store = {

val uncachedStore = ReadableStore.fromFnFuture(getEmbeddingAndCandidates)

memCacheConfigOpt match {

case Some(config) =>

SimilarityEngine.addMemCache(

underlyingStore = uncachedStore,

memCacheConfig = config,

statsReceiver = scopedStats

)

case \_ => uncachedStore

}

}

def toSimilarityEngineInfo(

query: HnswANNEngineQuery,

score: Double

): SimilarityEngineInfo = {

SimilarityEngineInfo(

similarityEngineType = this.identifier,

modelId = Some(query.modelId),

score = Some(score))

}

override def getCandidates(

engineQuery: HnswANNEngineQuery

): Future[Option[Seq[TweetWithScore]]] = {

val versionedStats = globalStats.scope(engineQuery.modelId)

SimilarityEngine.getFromFn(

store.get,

engineQuery,

engineConfig,

engineQuery.params,

versionedStats

)

}

}