package com.twitter.ann.scalding.offline

import com.twitter.ann.common.\_

import com.twitter.ann.hnsw.{HnswParams, TypedHnswIndex}

import com.twitter.bijection.Injection

import com.twitter.cortex.ml.embeddings.common.{EntityKind, Helpers, UserKind}

import com.twitter.entityembeddings.neighbors.thriftscala.{EntityKey, NearestNeighbors, Neighbor}

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

import com.twitter.ml.api.embedding.EmbeddingMath.{Float => math}

import com.twitter.ml.featurestore.lib.embedding.EmbeddingWithEntity

import com.twitter.ml.featurestore.lib.{EntityId, UserId}

import com.twitter.scalding.typed.{TypedPipe, UnsortedGrouped}

import com.twitter.scalding.{Args, DateRange, Stat, TextLine, UniqueID}

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

import com.twitter.util.{Await, FuturePool}

import scala.util.Random

case class Index[T, D <: Distance[D]](

injection: Injection[T, Array[Byte]],

metric: Metric[D],

dimension: Int,

directory: AbstractFile) {

lazy val annIndex = TypedHnswIndex.loadIndex[T, D](

dimension,

metric,

injection,

ReadWriteFuturePool(FuturePool.immediatePool),

directory

)

}

object KnnHelper {

def getFilteredUserEmbeddings(

args: Args,

filterPath: Option[String],

reducers: Int,

useHashJoin: Boolean

)(

implicit dateRange: DateRange

): TypedPipe[EmbeddingWithEntity[UserId]] = {

val userEmbeddings: TypedPipe[EmbeddingWithEntity[UserId]] =

UserKind.parser.getEmbeddingFormat(args, "consumer").getEmbeddings

filterPath match {

case Some(fileName: String) =>

val filterUserIds: TypedPipe[UserId] = TypedPipe

.from(TextLine(fileName))

.flatMap { idLine =>

Helpers.optionalToLong(idLine)

}

.map { id =>

UserId(id)

}

Helpers

.adjustableJoin(

left = userEmbeddings.groupBy(\_.entityId),

right = filterUserIds.asKeys,

useHashJoin = useHashJoin,

reducers = Some(reducers)

).map {

case (\_, (embedding, \_)) => embedding

}

case None => userEmbeddings

}

}

def getNeighborsPipe[T <: EntityId, D <: Distance[D]](

args: Args,

uncastEntityKind: EntityKind[\_],

uncastMetric: Metric[\_],

ef: Int,

consumerEmbeddings: TypedPipe[EmbeddingWithEntity[UserId]],

abstractFile: Option[AbstractFile],

reducers: Int,

numNeighbors: Int,

dimension: Int

)(

implicit dateRange: DateRange

): TypedPipe[(EntityKey, NearestNeighbors)] = {

val entityKind = uncastEntityKind.asInstanceOf[EntityKind[T]]

val injection = entityKind.byteInjection

val metric = uncastMetric.asInstanceOf[Metric[D]]

abstractFile match {

case Some(directory: AbstractFile) =>

val index = Index(injection, metric, dimension, directory)

consumerEmbeddings

.map { embedding =>

val knn = Await.result(

index.annIndex.queryWithDistance(

Embedding(embedding.embedding.toArray),

numNeighbors,

HnswParams(ef)

)

)

val neighborList = knn

.filter(\_.neighbor.toString != embedding.entityId.userId.toString)

.map(nn =>

Neighbor(

neighbor = EntityKey(nn.neighbor.toString),

similarity = Some(1 - nn.distance.distance)))

EntityKey(embedding.entityId.toString) -> NearestNeighbors(neighborList)

}

case None =>

val producerEmbeddings: TypedPipe[EmbeddingWithEntity[UserId]] =

UserKind.parser.getEmbeddingFormat(args, "producer").getEmbeddings

bruteForceNearestNeighbors(

consumerEmbeddings,

producerEmbeddings,

numNeighbors,

reducers

)

}

}

def bruteForceNearestNeighbors(

consumerEmbeddings: TypedPipe[EmbeddingWithEntity[UserId]],

producerEmbeddings: TypedPipe[EmbeddingWithEntity[UserId]],

numNeighbors: Int,

reducers: Int

): TypedPipe[(EntityKey, NearestNeighbors)] = {

consumerEmbeddings

.cross(producerEmbeddings)

.map {

case (cEmbed: EmbeddingWithEntity[UserId], pEmbed: EmbeddingWithEntity[UserId]) =>

// Cosine similarity

val cEmbedNorm = math.l2Norm(cEmbed.embedding).toFloat

val pEmbedNorm = math.l2Norm(pEmbed.embedding).toFloat

val distance: Float = -math.dotProduct(

(math.scalarProduct(cEmbed.embedding, 1 / cEmbedNorm)),

math.scalarProduct(pEmbed.embedding, 1 / pEmbedNorm))

(

UserKind.stringInjection(cEmbed.entityId),

(distance, UserKind.stringInjection(pEmbed.entityId)))

}

.groupBy(\_.\_1).withReducers(reducers)

.sortWithTake(numNeighbors) {

case ((\_: String, (sim1: Float, \_: String)), (\_: String, (sim2: Float, \_: String))) =>

sim1 < sim2

}

.map {

case (consumerId: String, (prodSims: Seq[(String, (Float, String))])) =>

EntityKey(consumerId) -> NearestNeighbors(

prodSims.map {

case (consumerId: String, (sim: Float, prodId: String)) =>

Neighbor(neighbor = EntityKey(prodId), similarity = Some(-sim.toDouble))

}

)

}

}

/\*\*

\* Calculate the nearest neighbors exhaustively between two entity embeddings using one as query and other as the search space.

\* @param queryEmbeddings entity embeddings for queries

\* @param searchSpaceEmbeddings entity embeddings for search space

\* @param metric distance metric

\* @param numNeighbors number of neighbors

\* @param queryIdsFilter optional query ids to filter to query entity embeddings

\* @param reducers number of reducers for grouping

\* @param isSearchSpaceLarger Used for optimization: Is the search space larger than the query space? Ignored if numOfSearchGroups > 1.

\* @param numOfSearchGroups we divide the search space into these groups (randomly). Useful when the search space is too large. Overrides isSearchSpaceLarger.

\* @param numReplicas Each search group will be responsible for 1/numReplicas queryEmebeddings.

\* This might speed up the search when the size of the index embeddings is

\* large.

\* @tparam A type of query entity

\* @tparam B type of search space entity

\* @tparam D type of distance

\*/

def findNearestNeighbours[A <: EntityId, B <: EntityId, D <: Distance[D]](

queryEmbeddings: TypedPipe[EmbeddingWithEntity[A]],

searchSpaceEmbeddings: TypedPipe[EmbeddingWithEntity[B]],

metric: Metric[D],

numNeighbors: Int = 10,

queryIdsFilter: Option[TypedPipe[A]] = Option.empty,

reducers: Int = 100,

mappers: Int = 100,

isSearchSpaceLarger: Boolean = true,

numOfSearchGroups: Int = 1,

numReplicas: Int = 1,

useCounters: Boolean = true

)(

implicit ordering: Ordering[A],

uid: UniqueID

): TypedPipe[(A, Seq[(B, D)])] = {

val filteredQueryEmbeddings = queryIdsFilter match {

case Some(filter) => {

queryEmbeddings.groupBy(\_.entityId).hashJoin(filter.asKeys).map {

case (x, (embedding, \_)) => embedding

}

}

case None => queryEmbeddings

}

if (numOfSearchGroups > 1) {

val indexingStrategy = BruteForceIndexingStrategy(metric)

findNearestNeighboursWithIndexingStrategy(

queryEmbeddings,

searchSpaceEmbeddings,

numNeighbors,

numOfSearchGroups,

indexingStrategy,

numReplicas,

Some(reducers),

useCounters = useCounters

)

} else {

findNearestNeighboursViaCross(

filteredQueryEmbeddings,

searchSpaceEmbeddings,

metric,

numNeighbors,

reducers,

mappers,

isSearchSpaceLarger)

}

}

/\*\*

\* Calculate the nearest neighbors using the specified indexing strategy between two entity

\* embeddings using one as query and other as the search space.

\* @param queryEmbeddings entity embeddings for queries

\* @param searchSpaceEmbeddings entity embeddings for search space. You should be able to fit

\* searchSpaceEmbeddings.size / numOfSearchGroups into memory.

\* @param numNeighbors number of neighbors

\* @param reducersOption number of reducers for the final sortedTake.

\* @param numOfSearchGroups we divide the search space into these groups (randomly). Useful when

\* the search space is too large. Search groups are shards. Choose this

\* number by ensuring searchSpaceEmbeddings.size / numOfSearchGroups

\* embeddings will fit into memory.

\* @param numReplicas Each search group will be responsible for 1/numReplicas queryEmebeddings.

\* By increasing this number, we can parallelize the work and reduce end to end

\* running times.

\* @param indexingStrategy How we will search for nearest neighbors within a search group

\* @param queryShards one step we have is to fan out the query embeddings. We create one entry

\* per search group. If numOfSearchGroups is large, then this fan out can take

\* a long time. You can shard the query shard first to parallelize this

\* process. One way to estimate what value to use:

\* queryEmbeddings.size \* numOfSearchGroups / queryShards should be around 1GB.

\* @param searchSpaceShards this param is similar to queryShards. Except it shards the search

\* space when numReplicas is too large. One way to estimate what value

\* to use: searchSpaceEmbeddings.size \* numReplicas / searchSpaceShards

\* should be around 1GB.

\* @tparam A type of query entity

\* @tparam B type of search space entity

\* @tparam D type of distance

\* @return a pipe keyed by the index embedding. The values are the list of numNeighbors nearest

\* neighbors along with their distances.

\*/

def findNearestNeighboursWithIndexingStrategy[A <: EntityId, B <: EntityId, D <: Distance[D]](

queryEmbeddings: TypedPipe[EmbeddingWithEntity[A]],

searchSpaceEmbeddings: TypedPipe[EmbeddingWithEntity[B]],

numNeighbors: Int,

numOfSearchGroups: Int,

indexingStrategy: IndexingStrategy[D],

numReplicas: Int = 1,

reducersOption: Option[Int] = None,

queryShards: Option[Int] = None,

searchSpaceShards: Option[Int] = None,

useCounters: Boolean = true

)(

implicit ordering: Ordering[A],

uid: UniqueID

): UnsortedGrouped[A, Seq[(B, D)]] = {

implicit val ord: Ordering[NNKey] = Ordering.by(NNKey.unapply)

val entityEmbeddings = searchSpaceEmbeddings.map { embedding: EmbeddingWithEntity[B] =>

val entityEmbedding =

EntityEmbedding(embedding.entityId, Embedding(embedding.embedding.toArray))

entityEmbedding

}

val shardedSearchSpace = shard(entityEmbeddings, searchSpaceShards)

val groupedSearchSpaceEmbeddings = shardedSearchSpace

.flatMap { entityEmbedding =>

val searchGroup = Random.nextInt(numOfSearchGroups)

(0 until numReplicas).map { replica =>

(NNKey(searchGroup, replica, Some(numReplicas)), entityEmbedding)

}

}

val shardedQueries = shard(queryEmbeddings, queryShards)

val groupedQueryEmbeddings = shardedQueries

.flatMap { entity =>

val replica = Random.nextInt(numReplicas)

(0 until numOfSearchGroups).map { searchGroup =>

(NNKey(searchGroup, replica, Some(numReplicas)), entity)

}

}.group

.withReducers(reducersOption.getOrElse(numOfSearchGroups \* numReplicas))

val numberAnnIndexQueries = Stat("NumberAnnIndexQueries")

val annIndexQueryTotalMs = Stat("AnnIndexQueryTotalMs")

val numberIndexBuilds = Stat("NumberIndexBuilds")

val annIndexBuildTotalMs = Stat("AnnIndexBuildTotalMs")

val groupedKnn = groupedQueryEmbeddings

.cogroup(groupedSearchSpaceEmbeddings) {

case (\_, queryIter, searchSpaceIter) =>

// This index build happens numReplicas times. Ideally we could serialize the queryable.

// And only build the index once per search group.

// The issues with that now are:

// - The HNSW queryable is not serializable in scalding

// - The way that map reduce works requires that there is a job that write out the search

// space embeddings numReplicas times. In the current setup, we can do that by sharding

// the embeddings first and then fanning out. But if we had a single queryable, we would

// not be able to shard it easily and writing this out would take a long time.

val indexBuildStartTime = System.currentTimeMillis()

val queryable = indexingStrategy.buildIndex(searchSpaceIter)

if (useCounters) {

numberIndexBuilds.inc()

annIndexBuildTotalMs.incBy(System.currentTimeMillis() - indexBuildStartTime)

}

queryIter.flatMap { query =>

val queryStartTime = System.currentTimeMillis()

val embedding = Embedding(query.embedding.toArray)

val result = Await.result(

queryable.queryWithDistance(embedding, numNeighbors)

)

val queryToTopNeighbors = result

.map { neighbor =>

(query.entityId, (neighbor.neighbor, neighbor.distance))

}

if (useCounters) {

numberAnnIndexQueries.inc()

annIndexQueryTotalMs.incBy(System.currentTimeMillis() - queryStartTime)

}

queryToTopNeighbors

}

}

.values

.group

val groupedKnnWithReducers = reducersOption

.map { reducers =>

groupedKnn

.withReducers(reducers)

}.getOrElse(groupedKnn)

groupedKnnWithReducers

.sortedTake(numNeighbors) {

Ordering

.by[(B, D), D] {

case (\_, distance) => distance

}

}

}

private[this] def shard[T](

pipe: TypedPipe[T],

numberOfShards: Option[Int]

): TypedPipe[T] = {

numberOfShards

.map { shards =>

pipe.shard(shards)

}.getOrElse(pipe)

}

private[this] def findNearestNeighboursViaCross[A <: EntityId, B <: EntityId, D <: Distance[D]](

queryEmbeddings: TypedPipe[EmbeddingWithEntity[A]],

searchSpaceEmbeddings: TypedPipe[EmbeddingWithEntity[B]],

metric: Metric[D],

numNeighbors: Int,

reducers: Int,

mappers: Int,

isSearchSpaceLarger: Boolean

)(

implicit ordering: Ordering[A]

): TypedPipe[(A, Seq[(B, D)])] = {

val crossed: TypedPipe[(A, (B, D))] = if (isSearchSpaceLarger) {

searchSpaceEmbeddings

.shard(mappers)

.cross(queryEmbeddings).map {

case (searchSpaceEmbedding, queryEmbedding) =>

val distance = metric.distance(searchSpaceEmbedding.embedding, queryEmbedding.embedding)

(queryEmbedding.entityId, (searchSpaceEmbedding.entityId, distance))

}

} else {

queryEmbeddings

.shard(mappers)

.cross(searchSpaceEmbeddings).map {

case (queryEmbedding, searchSpaceEmbedding) =>

val distance = metric.distance(searchSpaceEmbedding.embedding, queryEmbedding.embedding)

(queryEmbedding.entityId, (searchSpaceEmbedding.entityId, distance))

}

}

crossed

.groupBy(\_.\_1)

.withReducers(reducers)

.sortedTake(numNeighbors) {

Ordering

.by[(A, (B, D)), D] {

case (\_, (\_, distance)) => distance

} // Sort by distance metric in ascending order

}.map {

case (queryId, neighbors) =>

(queryId, neighbors.map(\_.\_2))

}

}

/\*\*

\* Convert nearest neighbors to string format.

\* By default format would be (queryId neighbourId:distance neighbourId:distance .....) in ascending order of distance.

\* @param nearestNeighbors nearest neighbors tuple in form of (queryId, Seq[(neighborId, distance)]

\* @param queryEntityKind entity kind of query

\* @param neighborEntityKind entity kind of search space/neighbors

\* @param idDistanceSeparator String separator to separate a single neighborId and distance. Default to colon (:)

\* @param neighborSeparator String operator to separate neighbors. Default to tab

\* @tparam A type of query entity

\* @tparam B type of search space entity

\* @tparam D type of distance

\*/

def nearestNeighborsToString[A <: EntityId, B <: EntityId, D <: Distance[D]](

nearestNeighbors: (A, Seq[(B, D)]),

queryEntityKind: EntityKind[A],

neighborEntityKind: EntityKind[B],

idDistanceSeparator: String = ":",

neighborSeparator: String = "\t"

): String = {

val (queryId, neighbors) = nearestNeighbors

val formattedNeighbors = neighbors.map {

case (neighbourId, distance) =>

s"${neighborEntityKind.stringInjection.apply(neighbourId)}$idDistanceSeparator${distance.distance}"

}

(queryEntityKind.stringInjection.apply(queryId) +: formattedNeighbors)

.mkString(neighborSeparator)

}

private[this] case class NNKey(

searchGroup: Int,

replica: Int,

maxReplica: Option[Int] = None) {

override def hashCode(): Int =

maxReplica.map(\_ \* searchGroup + replica).getOrElse(super.hashCode())

}

}