package com.twitter.ann.scalding.offline

import com.twitter.ann.common.Metric

import com.twitter.bijection.scrooge.BinaryScalaCodec

import com.twitter.ml.featurestore.lib.UserId

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

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

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

import com.twitter.scalding.commons.source.VersionedKeyValSource

import com.twitter.scalding.typed.TypedPipe

import com.twitter.scalding.{Args, DateOps, DateParser, DateRange, Execution, TypedTsv, UniqueID}

import com.twitter.scalding\_internal.job.TwitterExecutionApp

import com.twitter.search.common.file.{AbstractFile, LocalFile}

import java.util.TimeZone

/\*\*

\* Generates the nearest neighbour for users and store them in Manhattan format i.e sequence files.

\* See README for oscar usage.

\*/

object KnnOfflineJob extends TwitterExecutionApp {

override def job: Execution[Unit] = Execution.withId { implicit uniqueId =>

Execution.getArgs.flatMap { args: Args =>

val knnDirectoryOpt: Option[String] = args.optional("knn\_directory")

knnDirectoryOpt match {

case Some(knnDirectory) =>

Execution.withCachedFile(knnDirectory) { directory =>

execute(args, Some(new LocalFile(directory.file)))

}

case None =>

execute(args, None)

}

}

}

/\*\*

\* Execute KnnOfflineJob

\* @param args: The args object for this job

\* @param abstractFile: An optional of producer embedding path

\*/

def execute(

args: Args,

abstractFile: Option[AbstractFile]

)(

implicit uniqueID: UniqueID

): Execution[Unit] = {

implicit val tz: TimeZone = TimeZone.getDefault()

implicit val dp: DateParser = DateParser.default

implicit val dateRange = DateRange.parse(args.list("date"))(DateOps.UTC, DateParser.default)

implicit val keyInject = BinaryScalaCodec(EntityKey)

implicit val valueInject = BinaryScalaCodec(NearestNeighbors)

val entityKind = EntityKind.getEntityKind(args("producer\_entity\_kind"))

val metric = Metric.fromString(args("metric"))

val outputPath: String = args("output\_path")

val numNeighbors: Int = args("neighbors").toInt

val ef = args.getOrElse("ef", numNeighbors.toString).toInt

val reducers: Int = args("reducers").toInt

val knnDimension: Int = args("dimension").toInt

val debugOutputPath: Option[String] = args.optional("debug\_output\_path")

val filterPath: Option[String] = args.optional("users\_filter\_path")

val shards: Int = args.getOrElse("shards", "100").toInt

val useHashJoin: Boolean = args.getOrElse("use\_hash\_join", "false").toBoolean

val mhOutput = VersionedKeyValSource[EntityKey, NearestNeighbors](

path = outputPath,

sourceVersion = None,

sinkVersion = None,

maxFailures = 0,

versionsToKeep = 1

)

val consumerEmbeddings: TypedPipe[EmbeddingWithEntity[UserId]] =

KnnHelper.getFilteredUserEmbeddings(

args,

filterPath,

reducers,

useHashJoin

)

val neighborsPipe: TypedPipe[(EntityKey, NearestNeighbors)] = KnnHelper.getNeighborsPipe(

args,

entityKind,

metric,

ef,

consumerEmbeddings,

abstractFile,

reducers,

numNeighbors,

knnDimension

)

val neighborsExecution: Execution[Unit] = neighborsPipe

.writeExecution(mhOutput)

// Write manual Inspection

debugOutputPath match {

case Some(path: String) =>

val debugExecution: Execution[Unit] = KnnDebug

.getDebugTable(

neighborsPipe = neighborsPipe,

shards = shards,

reducers = reducers

)

.writeExecution(TypedTsv(path))

Execution.zip(debugExecution, neighborsExecution).unit

case None => neighborsExecution

}

}

}