package com.twitter.follow\_recommendations.common.rankers.ml\_ranker.scoring

import com.twitter.cortex.deepbird.thriftjava.DeepbirdPredictionService

import com.twitter.cortex.deepbird.thriftjava.ModelSelector

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.follow\_recommendations.common.models.CandidateUser

import com.twitter.follow\_recommendations.common.models.HasDebugOptions

import com.twitter.follow\_recommendations.common.models.HasDisplayLocation

import com.twitter.follow\_recommendations.common.models.Score

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.Feature

import com.twitter.ml.api.RichDataRecord

import com.twitter.ml.prediction\_service.{BatchPredictionRequest => JBatchPredictionRequest}

import com.twitter.product\_mixer.core.model.marshalling.request.HasClientContext

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

import com.twitter.util.Future

import com.twitter.util.TimeoutException

import scala.collection.JavaConversions.\_

import scala.collection.JavaConverters.\_

/\*\*

\* Generic trait that implements the scoring given a deepbirdClient

\* To test out a new model, create a scorer extending this trait, override the modelName and inject the scorer

\*/

trait DeepbirdScorer extends Scorer {

def modelName: String

def predictionFeature: Feature.Continuous

// Set a default batchSize of 100 when making model prediction calls to the Deepbird V2 prediction server

def batchSize: Int = 100

def deepbirdClient: DeepbirdPredictionService.ServiceToClient

def baseStats: StatsReceiver

def modelSelector: ModelSelector = new ModelSelector().setId(modelName)

def stats: StatsReceiver = baseStats.scope(this.getClass.getSimpleName).scope(modelName)

private def requestCount = stats.counter("requests")

private def emptyRequestCount = stats.counter("empty\_requests")

private def successCount = stats.counter("success")

private def failureCount = stats.counter("failures")

private def inputRecordsStat = stats.stat("input\_records")

private def outputRecordsStat = stats.stat("output\_records")

// Counters for tracking batch-prediction statistics when making DBv2 prediction calls

//

// numBatchRequests tracks the number of batch prediction requests made to DBv2 prediction servers

private def numBatchRequests = stats.counter("batches")

// numEmptyBatchRequests tracks the number of batch prediction requests made to DBv2 prediction servers

// that had an empty input DataRecord

private def numEmptyBatchRequests = stats.counter("empty\_batches")

// numTimedOutBatchRequests tracks the number of batch prediction requests made to DBv2 prediction servers

// that had timed-out

private def numTimedOutBatchRequests = stats.counter("timeout\_batches")

private def batchPredictionLatency = stats.stat("batch\_prediction\_latency")

private def predictionLatency = stats.stat("prediction\_latency")

private def numEmptyModelPredictions = stats.counter("empty\_model\_predictions")

private def numNonEmptyModelPredictions = stats.counter("non\_empty\_model\_predictions")

private val DefaultPredictionScore = 0.0

/\*\*

\* NOTE: For instances of [[DeepbirdScorer]] this function SHOULD NOT be used.

\* Please use [[score(records: Seq[DataRecord])]] instead.

\*/

@Deprecated

def score(

target: HasClientContext with HasParams with HasDisplayLocation with HasDebugOptions,

candidates: Seq[CandidateUser]

): Seq[Option[Score]] =

throw new UnsupportedOperationException(

"For instances of DeepbirdScorer this operation is not defined. Please use " +

"`def score(records: Seq[DataRecord]): Stitch[Seq[Score]]` " +

"instead.")

override def score(records: Seq[DataRecord]): Stitch[Seq[Score]] = {

requestCount.incr()

if (records.isEmpty) {

emptyRequestCount.incr()

Stitch.Nil

} else {

inputRecordsStat.add(records.size)

Stitch.callFuture(

batchPredict(records, batchSize)

.map { recordList =>

val scores = recordList.map { record =>

Score(

value = record.getOrElse(DefaultPredictionScore),

rankerId = Some(id),

scoreType = scoreType)

}

outputRecordsStat.add(scores.size)

scores

}.onSuccess(\_ => successCount.incr())

.onFailure(\_ => failureCount.incr()))

}

}

def batchPredict(

dataRecords: Seq[DataRecord],

batchSize: Int

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

Stat

.timeFuture(predictionLatency) {

val batchedDataRecords = dataRecords.grouped(batchSize).toSeq

numBatchRequests.incr(batchedDataRecords.size)

Future

.collect(batchedDataRecords.map(batch => predict(batch)))

.map(res => res.reduce(\_ ++ \_))

}

}

def predict(dataRecords: Seq[DataRecord]): Future[Seq[Option[Double]]] = {

Stat

.timeFuture(batchPredictionLatency) {

if (dataRecords.isEmpty) {

numEmptyBatchRequests.incr()

Future.Nil

} else {

deepbirdClient

.batchPredictFromModel(new JBatchPredictionRequest(dataRecords.asJava), modelSelector)

.map { response =>

response.predictions.toSeq.map { prediction =>

val predictionFeatureOption = Option(

new RichDataRecord(prediction).getFeatureValue(predictionFeature)

)

predictionFeatureOption match {

case Some(predictionValue) =>

numNonEmptyModelPredictions.incr()

Option(predictionValue.toDouble)

case None =>

numEmptyModelPredictions.incr()

Option(DefaultPredictionScore)

}

}

}

.rescue {

case e: TimeoutException => // DBv2 prediction calls that timed out

numTimedOutBatchRequests.incr()

stats.counter(e.getClass.getSimpleName).incr()

Future.value(dataRecords.map(\_ => Option(DefaultPredictionScore)))

case e: Exception => // other generic DBv2 prediction call failures

stats.counter(e.getClass.getSimpleName).incr()

Future.value(dataRecords.map(\_ => Option(DefaultPredictionScore)))

}

}

}

}

}