package com.twitter.product\_mixer.component\_library.scorer.cortex

import com.google.protobuf.ByteString

import com.twitter.ml.prediction\_service.BatchPredictionRequest

import com.twitter.ml.prediction\_service.BatchPredictionResponse

import com.twitter.product\_mixer.component\_library.scorer.common.ManagedModelClient

import com.twitter.product\_mixer.component\_library.scorer.common.ModelSelector

import com.twitter.product\_mixer.core.feature.Feature

import com.twitter.product\_mixer.core.feature.datarecord.BaseDataRecordFeature

import com.twitter.product\_mixer.core.feature.datarecord.TensorDataRecordCompatible

import com.twitter.product\_mixer.core.feature.featuremap.FeatureMap

import com.twitter.product\_mixer.core.feature.featuremap.datarecord.DataRecordConverter

import com.twitter.product\_mixer.core.feature.featuremap.datarecord.DataRecordExtractor

import com.twitter.product\_mixer.core.feature.featuremap.datarecord.FeaturesScope

import com.twitter.product\_mixer.core.functional\_component.scorer.Scorer

import com.twitter.product\_mixer.core.model.common.CandidateWithFeatures

import com.twitter.product\_mixer.core.model.common.UniversalNoun

import com.twitter.product\_mixer.core.model.common.identifier.ScorerIdentifier

import com.twitter.product\_mixer.core.pipeline.PipelineQuery

import com.twitter.product\_mixer.core.pipeline.pipeline\_failure.IllegalStateFailure

import inference.GrpcService

import inference.GrpcService.ModelInferRequest

import inference.GrpcService.ModelInferResponse

import com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailure

import com.twitter.stitch.Stitch

import org.apache.thrift.TDeserializer

import org.apache.thrift.TSerializer

import scala.collection.JavaConverters.\_

private[cortex] class CortexManagedDataRecordScorer[

Query <: PipelineQuery,

Candidate <: UniversalNoun[Any],

QueryFeatures <: BaseDataRecordFeature[Query, \_],

CandidateFeatures <: BaseDataRecordFeature[Candidate, \_],

ResultFeatures <: BaseDataRecordFeature[Candidate, \_] with TensorDataRecordCompatible[\_]

](

override val identifier: ScorerIdentifier,

modelSignature: String,

modelSelector: ModelSelector[Query],

modelClient: ManagedModelClient,

queryFeatures: FeaturesScope[QueryFeatures],

candidateFeatures: FeaturesScope[CandidateFeatures],

resultFeatures: Set[ResultFeatures])

extends Scorer[Query, Candidate] {

require(resultFeatures.nonEmpty, "Result features cannot be empty")

override val features: Set[Feature[\_, \_]] = resultFeatures.asInstanceOf[Set[Feature[\_, \_]]]

private val queryDataRecordAdapter = new DataRecordConverter(queryFeatures)

private val candidatesDataRecordAdapter = new DataRecordConverter(candidateFeatures)

private val resultDataRecordExtractor = new DataRecordExtractor(resultFeatures)

private val localTSerializer = new ThreadLocal[TSerializer] {

override protected def initialValue: TSerializer = new TSerializer()

}

private val localTDeserializer = new ThreadLocal[TDeserializer] {

override protected def initialValue: TDeserializer = new TDeserializer()

}

override def apply(

query: Query,

candidates: Seq[CandidateWithFeatures[Candidate]]

): Stitch[Seq[FeatureMap]] = {

modelClient.score(buildRequest(query, candidates)).map(buildResponse(candidates, \_))

}

/\*\*

\* Takes candidates to be scored and converts it to a ModelInferRequest that can be passed to the

\* managed ML service

\*/

private def buildRequest(

query: Query,

scorerCandidates: Seq[CandidateWithFeatures[Candidate]]

): ModelInferRequest = {

// Convert the feature maps to thrift data records and construct thrift request.

val thriftDataRecords = scorerCandidates.map { candidate =>

candidatesDataRecordAdapter.toDataRecord(candidate.features)

}

val batchRequest = new BatchPredictionRequest(thriftDataRecords.asJava)

query.features.foreach { featureMap =>

batchRequest.setCommonFeatures(queryDataRecordAdapter.toDataRecord(featureMap))

}

val serializedBatchRequest = localTSerializer.get().serialize(batchRequest)

// Build Tensor Request

val requestBuilder = ModelInferRequest

.newBuilder()

modelSelector.apply(query).foreach { modelName =>

requestBuilder.setModelName(modelName) // model name in the model config

}

val inputTensorBuilder = ModelInferRequest.InferInputTensor

.newBuilder()

.setName("request")

.setDatatype("UINT8")

.addShape(serializedBatchRequest.length)

val inferParameter = GrpcService.InferParameter

.newBuilder()

.setStringParam(modelSignature) // signature of exported tf function

.build()

requestBuilder

.addInputs(inputTensorBuilder)

.addRawInputContents(ByteString.copyFrom(serializedBatchRequest))

.putParameters("signature\_name", inferParameter)

.build()

}

private def buildResponse(

scorerCandidates: Seq[CandidateWithFeatures[Candidate]],

response: ModelInferResponse

): Seq[FeatureMap] = {

val responseByteString = if (response.getRawOutputContentsList.isEmpty()) {

throw PipelineFailure(

IllegalStateFailure,

"Model inference response has empty raw outputContents")

} else {

response.getRawOutputContents(0)

}

val batchPredictionResponse: BatchPredictionResponse = new BatchPredictionResponse()

localTDeserializer.get().deserialize(batchPredictionResponse, responseByteString.toByteArray)

// get the prediction values from the batch prediction response

val resultScoreMaps =

batchPredictionResponse.predictions.asScala.map(resultDataRecordExtractor.fromDataRecord)

if (resultScoreMaps.size != scorerCandidates.size) {

throw PipelineFailure(IllegalStateFailure, "Result Size mismatched candidates size")

}

resultScoreMaps

}

}