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

import com.twitter.finagle.stats.StatsReceiver

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

import com.twitter.product\_mixer.component\_library.scorer.tensorbuilder.ModelInferRequestBuilder

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

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

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

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

import com.twitter.stitch.Stitch

import com.twitter.util.logging.Logging

import inference.GrpcService.ModelInferRequest

import inference.GrpcService.ModelInferResponse.InferOutputTensor

import scala.collection.convert.ImplicitConversions.`collection AsScalaIterable`

private[scorer] class CortexManagedInferenceServiceTensorScorer[

Query <: PipelineQuery,

Candidate <: UniversalNoun[Any]

](

override val identifier: ScorerIdentifier,

modelInferRequestBuilder: ModelInferRequestBuilder[

Query,

Candidate

],

resultFeatureExtractors: Seq[FeatureWithExtractor[Query, Candidate, \_]],

client: MLModelInferenceClient,

statsReceiver: StatsReceiver)

extends Scorer[Query, Candidate]

with Logging {

require(resultFeatureExtractors.nonEmpty, "Result Extractors cannot be empty")

private val managedServiceRequestFailures = statsReceiver.counter("managedServiceRequestFailures")

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

resultFeatureExtractors.map(\_.feature).toSet.asInstanceOf[Set[Feature[\_, \_]]]

override def apply(

query: Query,

candidates: Seq[CandidateWithFeatures[Candidate]]

): Stitch[Seq[FeatureMap]] = {

val batchInferRequest: ModelInferRequest = modelInferRequestBuilder(query, candidates)

val managedServiceResponse: Stitch[Seq[InferOutputTensor]] =

client.score(batchInferRequest).map(\_.getOutputsList.toSeq).onFailure { e =>

error(s"request to ML Managed Service Failed: $e")

managedServiceRequestFailures.incr()

}

managedServiceResponse.map { responses =>

extractResponse(query, candidates.map(\_.candidate), responses)

}

}

def extractResponse(

query: Query,

candidates: Seq[Candidate],

tensorOutput: Seq[InferOutputTensor]

): Seq[FeatureMap] = {

val featureMapBuilders = candidates.map { \_ => FeatureMapBuilder.apply() }

// Extract the feature for each candidate from the tensor outputs

resultFeatureExtractors.foreach {

case FeatureWithExtractor(feature, extractor) =>

val extractedValues = extractor.apply(query, tensorOutput)

if (candidates.size != extractedValues.size) {

throw PipelineFailure(

IllegalStateFailure,

s"Managed Service returned a different number of $feature than the number of candidates." +

s"Returned ${extractedValues.size} scores but there were ${candidates.size} candidates."

)

}

// Go through the extracted features list one by one and update the feature map result for each candidate.

featureMapBuilders.zip(extractedValues).foreach {

case (builder, value) =>

builder.add(feature, Some(value))

}

}

featureMapBuilders.map(\_.build())

}

}

case class FeatureWithExtractor[

-Query <: PipelineQuery,

-Candidate <: UniversalNoun[Any],

ResultType

](

feature: Feature[Candidate, Option[ResultType]],

featureExtractor: ModelFeatureExtractor[Query, ResultType])

class UnexpectedFeatureTypeException(feature: Feature[\_, \_])

extends UnsupportedOperationException(s"Unsupported Feature type passed in $feature")