package com.twitter.cr\_mixer.similarity\_engine

import com.twitter.cr\_mixer.model.SimilarityEngineInfo

import com.twitter.cr\_mixer.model.SourceInfo

import com.twitter.cr\_mixer.model.TweetWithScore

import com.twitter.cr\_mixer.param.ConsumerBasedWalsParams

import com.twitter.cr\_mixer.similarity\_engine.ConsumerBasedWalsSimilarityEngine.Query

import com.twitter.cr\_mixer.thriftscala.SimilarityEngineType

import com.twitter.cr\_mixer.thriftscala.SourceType

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.simclusters\_v2.thriftscala.InternalId

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi

import com.twitter.util.Future

import io.grpc.ManagedChannel

import tensorflow.serving.Predict.PredictRequest

import tensorflow.serving.Predict.PredictResponse

import tensorflow.serving.PredictionServiceGrpc

import org.tensorflow.example.Feature

import org.tensorflow.example.Int64List

import org.tensorflow.example.FloatList

import org.tensorflow.example.Features

import org.tensorflow.example.Example

import tensorflow.serving.Model

import org.tensorflow.framework.TensorProto

import org.tensorflow.framework.DataType

import org.tensorflow.framework.TensorShapeProto

import com.twitter.finagle.grpc.FutureConverters

import java.util.ArrayList

import java.lang

import com.twitter.util.Return

import com.twitter.util.Throw

import java.util.concurrent.ConcurrentHashMap

import scala.jdk.CollectionConverters.\_

// Stats object maintain a set of stats that are specific to the Wals Engine.

case class WalsStats(scope: String, scopedStats: StatsReceiver) {

val requestStat = scopedStats.scope(scope)

val inputSignalSize = requestStat.stat("input\_signal\_size")

val latency = requestStat.stat("latency\_ms")

val latencyOnError = requestStat.stat("error\_latency\_ms")

val latencyOnSuccess = requestStat.stat("success\_latency\_ms")

val requests = requestStat.counter("requests")

val success = requestStat.counter("success")

val failures = requestStat.scope("failures")

def onFailure(t: Throwable, startTimeMs: Long) {

val duration = System.currentTimeMillis() - startTimeMs

latency.add(duration)

latencyOnError.add(duration)

failures.counter(t.getClass.getName).incr()

}

def onSuccess(startTimeMs: Long) {

val duration = System.currentTimeMillis() - startTimeMs

latency.add(duration)

latencyOnSuccess.add(duration)

success.incr()

}

}

// StatsMap maintains a mapping from Model's input signature to a stats receiver

// The Wals model suports multiple input signature which can run different graphs internally and

// can have a different performance profile.

// Invoking StatsReceiver.stat() on each request can create a new stat object and can be expensive

// in performance critical paths.

object WalsStatsMap {

val mapping = new ConcurrentHashMap[String, WalsStats]()

def get(scope: String, scopedStats: StatsReceiver): WalsStats = {

mapping.computeIfAbsent(scope, (scope) => WalsStats(scope, scopedStats))

}

}

case class ConsumerBasedWalsSimilarityEngine(

homeNaviGRPCClient: ManagedChannel,

adsFavedNaviGRPCClient: ManagedChannel,

adsMonetizableNaviGRPCClient: ManagedChannel,

statsReceiver: StatsReceiver)

extends ReadableStore[

Query,

Seq[TweetWithScore]

] {

override def get(

query: ConsumerBasedWalsSimilarityEngine.Query

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

val startTimeMs = System.currentTimeMillis()

val stats =

WalsStatsMap.get(

query.wilyNsName + "/" + query.modelSignatureName,

statsReceiver.scope("NaviPredictionService")

)

stats.requests.incr()

stats.inputSignalSize.add(query.sourceIds.size)

try {

// avoid inference calls is source signals are empty

if (query.sourceIds.isEmpty) {

Future.value(Some(Seq.empty))

} else {

val grpcClient = query.wilyNsName match {

case "navi-wals-recommended-tweets-home-client" => homeNaviGRPCClient

case "navi-wals-ads-faved-tweets" => adsFavedNaviGRPCClient

case "navi-wals-ads-monetizable-tweets" => adsFavedNaviGRPCClient

// default to homeNaviGRPCClient

case \_ => homeNaviGRPCClient

}

val stub = PredictionServiceGrpc.newFutureStub(grpcClient)

val inferRequest = getModelInput(query)

FutureConverters

.RichListenableFuture(stub.predict(inferRequest)).toTwitter

.transform {

case Return(resp) =>

stats.onSuccess(startTimeMs)

Future.value(Some(getModelOutput(query, resp)))

case Throw(e) =>

stats.onFailure(e, startTimeMs)

Future.exception(e)

}

}

} catch {

case e: Throwable => Future.exception(e)

}

}

def getFeaturesForRecommendations(query: ConsumerBasedWalsSimilarityEngine.Query): Example = {

val tweetIds = new ArrayList[lang.Long]()

val tweetFaveWeight = new ArrayList[lang.Float]()

query.sourceIds.foreach { sourceInfo =>

val weight = sourceInfo.sourceType match {

case SourceType.TweetFavorite | SourceType.Retweet => 1.0f

// currently no-op - as we do not get negative signals

case SourceType.TweetDontLike | SourceType.TweetReport | SourceType.AccountMute |

SourceType.AccountBlock =>

0.0f

case \_ => 0.0f

}

sourceInfo.internalId match {

case InternalId.TweetId(tweetId) =>

tweetIds.add(tweetId)

tweetFaveWeight.add(weight)

case \_ =>

throw new IllegalArgumentException(

s"Invalid InternalID - does not contain TweetId for Source Signal: ${sourceInfo}")

}

}

val tweetIdsFeature =

Feature

.newBuilder().setInt64List(

Int64List

.newBuilder().addAllValue(tweetIds).build()

).build()

val tweetWeightsFeature = Feature

.newBuilder().setFloatList(

FloatList.newBuilder().addAllValue(tweetFaveWeight).build()).build()

val features = Features

.newBuilder()

.putFeature("tweet\_ids", tweetIdsFeature)

.putFeature("tweet\_weights", tweetWeightsFeature)

.build()

Example.newBuilder().setFeatures(features).build()

}

def getModelInput(query: ConsumerBasedWalsSimilarityEngine.Query): PredictRequest = {

val tfExample = getFeaturesForRecommendations(query)

val inferenceRequest = PredictRequest

.newBuilder()

.setModelSpec(

Model.ModelSpec

.newBuilder()

.setName(query.modelName)

.setSignatureName(query.modelSignatureName))

.putInputs(

query.modelInputName,

TensorProto

.newBuilder()

.setDtype(DataType.DT\_STRING)

.setTensorShape(TensorShapeProto

.newBuilder()

.addDim(TensorShapeProto.Dim.newBuilder().setSize(1)))

.addStringVal(tfExample.toByteString)

.build()

).build()

inferenceRequest

}

def getModelOutput(query: Query, response: PredictResponse): Seq[TweetWithScore] = {

val outputName = query.modelOutputName

if (response.containsOutputs(outputName)) {

val tweetList = response.getOutputsMap

.get(outputName)

.getInt64ValList.asScala

tweetList.zip(tweetList.size to 1 by -1).map { (tweetWithScore) =>

TweetWithScore(tweetWithScore.\_1, tweetWithScore.\_2.toLong)

}

} else {

Seq.empty

}

}

}

object ConsumerBasedWalsSimilarityEngine {

case class Query(

sourceIds: Seq[SourceInfo],

modelName: String,

modelInputName: String,

modelOutputName: String,

modelSignatureName: String,

wilyNsName: String,

)

def fromParams(

sourceIds: Seq[SourceInfo],

params: configapi.Params,

): EngineQuery[Query] = {

EngineQuery(

Query(

sourceIds,

params(ConsumerBasedWalsParams.ModelNameParam),

params(ConsumerBasedWalsParams.ModelInputNameParam),

params(ConsumerBasedWalsParams.ModelOutputNameParam),

params(ConsumerBasedWalsParams.ModelSignatureNameParam),

params(ConsumerBasedWalsParams.WilyNsNameParam),

),

params

)

}

def toSimilarityEngineInfo(

score: Double

): SimilarityEngineInfo = {

SimilarityEngineInfo(

similarityEngineType = SimilarityEngineType.ConsumerBasedWalsANN,

modelId = None,

score = Some(score))

}

}