package com.twitter.home\_mixer.product.scored\_tweets.scorer

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.home\_mixer.model.HomeFeatures.ScoreFeature

import com.twitter.home\_mixer.model.HomeFeatures.WeightedModelScoreFeature

import com.twitter.home\_mixer.product.scored\_tweets.model.ScoredTweetsQuery

import com.twitter.home\_mixer.product.scored\_tweets.scorer.PredictedScoreFeature.PredictedScoreFeatures

import com.twitter.ml.api.DataRecord

import com.twitter.product\_mixer.component\_library.model.candidate.TweetCandidate

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

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

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

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

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

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.functional\_component.scorer.Scorer

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

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.product\_mixer.core.util.OffloadFuturePools

import com.twitter.stitch.Stitch

import com.twitter.timelines.clients.predictionservice.PredictionGRPCService

import com.twitter.timelines.clients.predictionservice.PredictionServiceGRPCClient

import com.twitter.util.Future

import com.twitter.util.Return

import javax.inject.Inject

import javax.inject.Singleton

object CommonFeaturesDataRecordFeature

extends DataRecordInAFeature[PipelineQuery]

with FeatureWithDefaultOnFailure[PipelineQuery, DataRecord] {

override def defaultValue: DataRecord = new DataRecord()

}

object CandidateFeaturesDataRecordFeature

extends DataRecordInAFeature[TweetCandidate]

with FeatureWithDefaultOnFailure[TweetCandidate, DataRecord] {

override def defaultValue: DataRecord = new DataRecord()

}

@Singleton

case class NaviModelScorer @Inject() (

predictionGRPCService: PredictionGRPCService,

statsReceiver: StatsReceiver)

extends Scorer[ScoredTweetsQuery, TweetCandidate] {

override val identifier: ScorerIdentifier = ScorerIdentifier("NaviModel")

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

CommonFeaturesDataRecordFeature,

CandidateFeaturesDataRecordFeature,

WeightedModelScoreFeature,

ScoreFeature

) ++ PredictedScoreFeatures.asInstanceOf[Set[Feature[\_, \_]]]

private val queryDataRecordAdapter = new DataRecordConverter(AllFeatures())

private val candidatesDataRecordAdapter = new DataRecordConverter(AllFeatures())

private val resultDataRecordExtractor = new DataRecordExtractor(PredictedScoreFeatures)

private val scopedStatsReceiver = statsReceiver.scope(getClass.getSimpleName)

private val failuresStat = scopedStatsReceiver.stat("failures")

private val responsesStat = scopedStatsReceiver.stat("responses")

private val invalidResponsesCounter = scopedStatsReceiver.counter("invalidResponses")

private val candidatesDataRecordAdapterLatencyStat =

scopedStatsReceiver.scope("candidatesDataRecordAdapter").stat("latency\_ms")

private val StatsReadabilityMultiplier = 1000

private val Epsilon = 0.001

private val PredictedScoreStatName = f"predictedScore${StatsReadabilityMultiplier}x"

private val MissingScoreStatName = "missingScore"

private val scoreStat = scopedStatsReceiver.stat(f"score${StatsReadabilityMultiplier}x")

private val RequestBatchSize = 64

private val DataRecordConstructionParallelism = 32

private val ModelId = "Home"

private val modelClient = new PredictionServiceGRPCClient(

service = predictionGRPCService,

statsReceiver = statsReceiver,

requestBatchSize = RequestBatchSize,

useCompact = false

)

override def apply(

query: ScoredTweetsQuery,

candidates: Seq[CandidateWithFeatures[TweetCandidate]]

): Stitch[Seq[FeatureMap]] = {

val commonRecord = query.features.map(queryDataRecordAdapter.toDataRecord)

val candidateRecords: Future[Seq[DataRecord]] =

Stat.time(candidatesDataRecordAdapterLatencyStat) {

OffloadFuturePools.parallelize[FeatureMap, DataRecord](

inputSeq = candidates.map(\_.features),

transformer = candidatesDataRecordAdapter.toDataRecord(\_),

parallelism = DataRecordConstructionParallelism,

default = new DataRecord

)

}

val scoreFeatureMaps = candidateRecords.flatMap { records =>

val predictionResponses =

modelClient.getPredictions(records, commonRecord, modelId = Some(ModelId))

predictionResponses.map { responses =>

failuresStat.add(responses.count(\_.isThrow))

responsesStat.add(responses.size)

if (responses.size == candidates.size) {

val predictedScoreFeatureMaps = responses.map {

case Return(dataRecord) => resultDataRecordExtractor.fromDataRecord(dataRecord)

case \_ => resultDataRecordExtractor.fromDataRecord(new DataRecord())

}

// Add Data Record to candidate Feature Map for logging in later stages

predictedScoreFeatureMaps.zip(records).map {

case (predictedScoreFeatureMap, candidateRecord) =>

val weightedModelScore = computeWeightedModelScore(query, predictedScoreFeatureMap)

scoreStat.add((weightedModelScore \* StatsReadabilityMultiplier).toFloat)

predictedScoreFeatureMap +

(CandidateFeaturesDataRecordFeature, candidateRecord) +

(CommonFeaturesDataRecordFeature, commonRecord.getOrElse(new DataRecord())) +

(ScoreFeature, Some(weightedModelScore)) +

(WeightedModelScoreFeature, Some(weightedModelScore))

}

} else {

invalidResponsesCounter.incr()

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

}

}

}

Stitch.callFuture(scoreFeatureMaps)

}

/\*\*

\* Compute the weighted sum of predicted scores of all engagements

\* Convert negative score to positive, if needed

\*/

private def computeWeightedModelScore(

query: PipelineQuery,

features: FeatureMap

): Double = {

val weightedScoreAndModelWeightSeq = PredictedScoreFeatures.toSeq.map { predictedScoreFeature =>

val predictedScoreOpt = predictedScoreFeature.extractScore(features)

predictedScoreOpt match {

case Some(predictedScore) =>

scopedStatsReceiver

.stat(predictedScoreFeature.statName, PredictedScoreStatName)

.add((predictedScore \* StatsReadabilityMultiplier).toFloat)

case None =>

scopedStatsReceiver.counter(predictedScoreFeature.statName, MissingScoreStatName).incr()

}

val weight = query.params(predictedScoreFeature.modelWeightParam)

val weightedScore = predictedScoreOpt.getOrElse(0.0) \* weight

(weightedScore, weight)

}

val (weightedScores, modelWeights) = weightedScoreAndModelWeightSeq.unzip

val combinedScoreSum = weightedScores.sum

val positiveModelWeightsSum = modelWeights.filter(\_ > 0.0).sum

val negativeModelWeightsSum = modelWeights.filter(\_ < 0).sum.abs

val modelWeightsSum = positiveModelWeightsSum + negativeModelWeightsSum

val weightedScoresSum =

if (modelWeightsSum == 0) combinedScoreSum.max(0.0)

else if (combinedScoreSum < 0)

(combinedScoreSum + negativeModelWeightsSum) / modelWeightsSum \* Epsilon

else combinedScoreSum + Epsilon

weightedScoresSum

}

}