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

import com.google.common.annotations.VisibleForTesting

import com.google.inject.Inject

import com.google.inject.Singleton

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.follow\_recommendations.common.base.Ranker

import com.twitter.follow\_recommendations.common.base.StatsUtil

import com.twitter.follow\_recommendations.common.base.StatsUtil.profileSeqResults

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

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

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

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

import com.twitter.follow\_recommendations.common.rankers.common.RankerId

import com.twitter.follow\_recommendations.common.rankers.common.RankerId.RankerId

import com.twitter.follow\_recommendations.common.rankers.utils.Utils

import com.twitter.follow\_recommendations.common.rankers.ml\_ranker.scoring.AdhocScorer

import com.twitter.follow\_recommendations.common.rankers.ml\_ranker.scoring.Scorer

import com.twitter.follow\_recommendations.common.rankers.ml\_ranker.scoring.ScorerFactory

import com.twitter.follow\_recommendations.common.utils.CollectionUtil

import com.twitter.ml.api.DataRecord

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

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

import com.twitter.timelines.configapi.Params

import com.twitter.util.logging.Logging

/\*\*

\* This class has a rank function that will perform 4 steps:

\* - choose which scorer to use for each candidate

\* - score candidates given their respective features

\* - add scoring information to the candidate

\* - sort candidates by their respective scores

\* The feature source and scorer will depend on the request's params

\*/

@Singleton

class MlRanker[

Target <: HasClientContext with HasParams with HasDisplayLocation with HasDebugOptions] @Inject() (

scorerFactory: ScorerFactory,

statsReceiver: StatsReceiver)

extends Ranker[Target, CandidateUser]

with Logging {

private val stats: StatsReceiver = statsReceiver.scope("ml\_ranker")

private val inputStat = stats.scope("1\_input")

private val selectScorerStat = stats.scope("2\_select\_scorer")

private val scoreStat = stats.scope("3\_score")

override def rank(

target: Target,

candidates: Seq[CandidateUser]

): Stitch[Seq[CandidateUser]] = {

profileSeqResults(candidates, inputStat)

val requestRankerId = target.params(MlRankerParams.RequestScorerIdParam)

val rankerIds = chooseRankerByCandidate(candidates, requestRankerId)

val scoreStitch = score(candidates, rankerIds, requestRankerId).map { scoredCandidates =>

{

// sort the candidates by score

val sortedCandidates = sort(target, scoredCandidates)

// add scribe field to candidates (if applicable) and return candidates

scribeCandidates(target, sortedCandidates)

}

}

StatsUtil.profileStitch(scoreStitch, stats.scope("rank"))

}

/\*\*

\* @param target: The WTF request for a given consumer.

\* @param candidates A list of candidates considered for recommendation.

\* @return A map from each candidate to a tuple that includes:

\* (1) The selected scorer that should be used to rank this candidate

\* (2) a flag determining whether the candidate is in a producer-side experiment.

\*/

private[ranking] def chooseRankerByCandidate(

candidates: Seq[CandidateUser],

requestRankerId: RankerId

): Map[CandidateUser, RankerId] = {

candidates.map { candidate =>

val selectedCandidateRankerId =

if (candidate.params == Params.Invalid || candidate.params == Params.Empty) {

selectScorerStat.counter("candidate\_params\_empty").incr()

requestRankerId

} else {

val candidateRankerId = candidate.params(MlRankerParams.CandidateScorerIdParam)

if (candidateRankerId == RankerId.None) {

// This candidate is a not part of any producer-side experiment.

selectScorerStat.counter("default\_to\_request\_ranker").incr()

requestRankerId

} else {

// This candidate is in a treatment bucket of a producer-side experiment.

selectScorerStat.counter("use\_candidate\_ranker").incr()

candidateRankerId

}

}

selectScorerStat.scope("selected").counter(selectedCandidateRankerId.toString).incr()

candidate -> selectedCandidateRankerId

}.toMap

}

@VisibleForTesting

private[ranking] def score(

candidates: Seq[CandidateUser],

rankerIds: Map[CandidateUser, RankerId],

requestRankerId: RankerId

): Stitch[Seq[CandidateUser]] = {

val features = candidates.map(\_.dataRecord.flatMap(\_.dataRecord))

require(features.forall(\_.nonEmpty), "features are not hydrated for all the candidates")

val scorers = scorerFactory.getScorers(rankerIds.values.toSeq.sorted.distinct)

// Scorers are split into ML-based and Adhoc (defined as a scorer that does not need to call an

// ML prediction service and scores candidates using locally-available data).

val (adhocScorers, mlScorers) = scorers.partition {

case \_: AdhocScorer => true

case \_ => false

}

// score candidates

val scoresStitch = score(features.map(\_.get), mlScorers)

val candidatesWithMlScoresStitch = scoresStitch.map { scoresSeq =>

candidates

.zip(scoresSeq).map { // copy datarecord and score into candidate object

case (candidate, scores) =>

val selectedRankerId = rankerIds(candidate)

val useRequestRanker =

candidate.params == Params.Invalid ||

candidate.params == Params.Empty ||

candidate.params(MlRankerParams.CandidateScorerIdParam) == RankerId.None

candidate.copy(

score = scores.scores.find(\_.rankerId.contains(requestRankerId)).map(\_.value),

scores = if (scores.scores.nonEmpty) {

Some(

scores.copy(

scores = scores.scores,

selectedRankerId = Some(selectedRankerId),

isInProducerScoringExperiment = !useRequestRanker

))

} else None

)

}

}

candidatesWithMlScoresStitch.map { candidates =>

// The basis for adhoc scores are the "request-level" ML ranker. We add the base score here

// while adhoc scorers are applied in [[AdhocRanker]].

addMlBaseScoresForAdhocScorers(candidates, requestRankerId, adhocScorers)

}

}

@VisibleForTesting

private[ranking] def addMlBaseScoresForAdhocScorers(

candidates: Seq[CandidateUser],

requestRankerId: RankerId,

adhocScorers: Seq[Scorer]

): Seq[CandidateUser] = {

candidates.map { candidate =>

candidate.scores match {

case Some(oldScores) =>

// 1. We fetch the ML score that is the basis of adhoc scores:

val baseMlScoreOpt = Utils.getCandidateScoreByRankerId(candidate, requestRankerId)

// 2. For each adhoc scorer, we copy the ML score object, changing only the ID and type.

val newScores = adhocScorers flatMap { adhocScorer =>

baseMlScoreOpt.map(

\_.copy(rankerId = Some(adhocScorer.id), scoreType = adhocScorer.scoreType))

}

// 3. We add the new adhoc score entries to the candidate.

candidate.copy(scores = Some(oldScores.copy(scores = oldScores.scores ++ newScores)))

case \_ =>

// Since there is no base ML score, there should be no adhoc score modification as well.

candidate

}

}

}

private[this] def score(

dataRecords: Seq[DataRecord],

scorers: Seq[Scorer]

): Stitch[Seq[Scores]] = {

val scoredResponse = scorers.map { scorer =>

StatsUtil.profileStitch(scorer.score(dataRecords), scoreStat.scope(scorer.id.toString))

}

// If we could score a candidate with too many rankers, it is likely to blow up the whole system.

// and fail back to default production model

StatsUtil.profileStitch(Stitch.collect(scoredResponse), scoreStat).map { scoresByScorerId =>

CollectionUtil.transposeLazy(scoresByScorerId).map { scoresPerCandidate =>

Scores(scoresPerCandidate)

}

}

}

// sort candidates using score in descending order

private[this] def sort(

target: Target,

candidates: Seq[CandidateUser]

): Seq[CandidateUser] = {

candidates.sortBy(c => -c.score.getOrElse(MlRanker.DefaultScore))

}

private[this] def scribeCandidates(

target: Target,

candidates: Seq[CandidateUser]

): Seq[CandidateUser] = {

val scribeRankingInfo: Boolean = target.params(MlRankerParams.ScribeRankingInfoInMlRanker)

scribeRankingInfo match {

case true => Utils.addRankingInfo(candidates, "MlRanker")

case false => candidates

}

}

}

object MlRanker {

// this is to ensure candidates with absent scores are ranked the last

val DefaultScore: Double = Double.MinValue

}