package com.twitter.frigate.pushservice.model.candidate

import com.twitter.frigate.common.base.FeatureMap

import com.twitter.frigate.common.rec\_types.RecTypes

import com.twitter.frigate.pushservice.model.PushTypes.PushCandidate

import com.twitter.frigate.pushservice.ml.HydrationContextBuilder

import com.twitter.frigate.pushservice.ml.PushMLModelScorer

import com.twitter.frigate.pushservice.params.PushFeatureSwitchParams

import com.twitter.frigate.pushservice.params.PushMLModel

import com.twitter.frigate.pushservice.params.WeightedOpenOrNtabClickModel

import com.twitter.nrel.hydration.push.HydrationContext

import com.twitter.timelines.configapi.FSParam

import com.twitter.util.Future

import java.util.concurrent.ConcurrentHashMap

import scala.collection.concurrent.{Map => CMap}

import scala.collection.convert.decorateAsScala.\_

trait MLScores {

self: PushCandidate =>

lazy val candidateHydrationContext: Future[HydrationContext] = HydrationContextBuilder.build(self)

def weightedOpenOrNtabClickModelScorer: PushMLModelScorer

// Used to store the scores and avoid duplicate prediction

private val qualityModelScores: CMap[

(PushMLModel.Value, WeightedOpenOrNtabClickModel.ModelNameType),

Future[Option[Double]]

] =

new ConcurrentHashMap[(PushMLModel.Value, WeightedOpenOrNtabClickModel.ModelNameType), Future[

Option[Double]

]]().asScala

def populateQualityModelScore(

pushMLModel: PushMLModel.Value,

modelVersion: WeightedOpenOrNtabClickModel.ModelNameType,

prob: Future[Option[Double]]

) = {

val modelAndVersion = (pushMLModel, modelVersion)

if (!qualityModelScores.contains(modelAndVersion)) {

qualityModelScores += modelAndVersion -> prob

}

}

// The ML scores that also depend on other candidates and are only available after all candidates are processed

// For example, the likelihood info for Importance Sampling

private lazy val crossCandidateMlScores: CMap[String, Double] =

new ConcurrentHashMap[String, Double]().asScala

def populateCrossCandidateMlScores(scoreName: String, score: Double): Unit = {

if (crossCandidateMlScores.contains(scoreName)) {

throw new Exception(

s"$scoreName has been populated in the CrossCandidateMlScores!\n" +

s"Existing crossCandidateMlScores are ${crossCandidateMlScores}\n"

)

}

crossCandidateMlScores += scoreName -> score

}

def getMLModelScore(

pushMLModel: PushMLModel.Value,

modelVersion: WeightedOpenOrNtabClickModel.ModelNameType

): Future[Option[Double]] = {

qualityModelScores.getOrElseUpdate(

(pushMLModel, modelVersion),

weightedOpenOrNtabClickModelScorer

.singlePredicationForModelVersion(modelVersion, self, Some(pushMLModel))

)

}

def getMLModelScoreWithoutUpdate(

pushMLModel: PushMLModel.Value,

modelVersion: WeightedOpenOrNtabClickModel.ModelNameType

): Future[Option[Double]] = {

qualityModelScores.getOrElse(

(pushMLModel, modelVersion),

Future.None

)

}

def getWeightedOpenOrNtabClickModelScore(

weightedOONCModelParam: FSParam[WeightedOpenOrNtabClickModel.ModelNameType]

): Future[Option[Double]] = {

getMLModelScore(

PushMLModel.WeightedOpenOrNtabClickProbability,

target.params(weightedOONCModelParam)

)

}

/\* After we unify the ranking and filtering models, we follow the iteration process below

When improving the WeightedOONC model,

1) Run experiment which only replace the ranking model

2) Make decisions according to the experiment results

3) Use the ranking model for filtering

4) Adjust percentile thresholds if necessary

\*/

lazy val mrWeightedOpenOrNtabClickRankingProbability: Future[Option[Double]] =

target.rankingModelParam.flatMap { modelParam =>

getWeightedOpenOrNtabClickModelScore(modelParam)

}

def getBigFilteringScore(

pushMLModel: PushMLModel.Value,

modelVersion: WeightedOpenOrNtabClickModel.ModelNameType

): Future[Option[Double]] = {

mrWeightedOpenOrNtabClickRankingProbability.flatMap {

case Some(rankingScore) =>

// Adds ranking score to feature map (we must ensure the feature key is also in the feature context)

mergeFeatures(

FeatureMap(

numericFeatures = Map("scribe.WeightedOpenOrNtabClickProbability" -> rankingScore)

)

)

getMLModelScore(pushMLModel, modelVersion)

case \_ => Future.None

}

}

def getWeightedOpenOrNtabClickScoreForScribing(): Seq[Future[Map[String, Double]]] = {

Seq(

mrWeightedOpenOrNtabClickRankingProbability.map {

case Some(score) => Map(PushMLModel.WeightedOpenOrNtabClickProbability.toString -> score)

case \_ => Map.empty[String, Double]

},

Future

.join(

target.rankingModelParam,

mrWeightedOpenOrNtabClickRankingProbability

).map {

case (rankingModelParam, Some(score)) =>

Map(target.params(rankingModelParam).toString -> score)

case \_ => Map.empty[String, Double]

}

)

}

def getNsfwScoreForScribing(): Seq[Future[Map[String, Double]]] = {

val nsfwScoreFut = getMLModelScoreWithoutUpdate(

PushMLModel.HealthNsfwProbability,

target.params(PushFeatureSwitchParams.BqmlHealthModelTypeParam))

Seq(nsfwScoreFut.map { nsfwScoreOpt =>

nsfwScoreOpt

.map(nsfwScore => Map(PushMLModel.HealthNsfwProbability.toString -> nsfwScore)).getOrElse(

Map.empty[String, Double])

})

}

def getBigFilteringSupervisedScoresForScribing(): Seq[Future[Map[String, Double]]] = {

if (target.params(

PushFeatureSwitchParams.EnableMrRequestScribingBigFilteringSupervisedScores)) {

Seq(

mrBigFilteringSupervisedSendingScore.map {

case Some(score) =>

Map(PushMLModel.BigFilteringSupervisedSendingModel.toString -> score)

case \_ => Map.empty[String, Double]

},

mrBigFilteringSupervisedWithoutSendingScore.map {

case Some(score) =>

Map(PushMLModel.BigFilteringSupervisedWithoutSendingModel.toString -> score)

case \_ => Map.empty[String, Double]

}

)

} else Seq.empty[Future[Map[String, Double]]]

}

def getBigFilteringRLScoresForScribing(): Seq[Future[Map[String, Double]]] = {

if (target.params(PushFeatureSwitchParams.EnableMrRequestScribingBigFilteringRLScores)) {

Seq(

mrBigFilteringRLSendingScore.map {

case Some(score) => Map(PushMLModel.BigFilteringRLSendingModel.toString -> score)

case \_ => Map.empty[String, Double]

},

mrBigFilteringRLWithoutSendingScore.map {

case Some(score) => Map(PushMLModel.BigFilteringRLWithoutSendingModel.toString -> score)

case \_ => Map.empty[String, Double]

}

)

} else Seq.empty[Future[Map[String, Double]]]

}

def buildModelScoresSeqForScribing(): Seq[Future[Map[String, Double]]] = {

getWeightedOpenOrNtabClickScoreForScribing() ++

getBigFilteringSupervisedScoresForScribing() ++

getBigFilteringRLScoresForScribing() ++

getNsfwScoreForScribing()

}

lazy val mrBigFilteringSupervisedSendingScore: Future[Option[Double]] =

getBigFilteringScore(

PushMLModel.BigFilteringSupervisedSendingModel,

target.params(PushFeatureSwitchParams.BigFilteringSupervisedSendingModelParam)

)

lazy val mrBigFilteringSupervisedWithoutSendingScore: Future[Option[Double]] =

getBigFilteringScore(

PushMLModel.BigFilteringSupervisedWithoutSendingModel,

target.params(PushFeatureSwitchParams.BigFilteringSupervisedWithoutSendingModelParam)

)

lazy val mrBigFilteringRLSendingScore: Future[Option[Double]] =

getBigFilteringScore(

PushMLModel.BigFilteringRLSendingModel,

target.params(PushFeatureSwitchParams.BigFilteringRLSendingModelParam)

)

lazy val mrBigFilteringRLWithoutSendingScore: Future[Option[Double]] =

getBigFilteringScore(

PushMLModel.BigFilteringRLWithoutSendingModel,

target.params(PushFeatureSwitchParams.BigFilteringRLWithoutSendingModelParam)

)

lazy val mrWeightedOpenOrNtabClickFilteringProbability: Future[Option[Double]] =

getWeightedOpenOrNtabClickModelScore(

target.filteringModelParam

)

lazy val mrQualityUprankingProbability: Future[Option[Double]] =

getMLModelScore(

PushMLModel.FilteringProbability,

target.params(PushFeatureSwitchParams.QualityUprankingModelTypeParam)

)

lazy val mrNsfwScore: Future[Option[Double]] =

getMLModelScoreWithoutUpdate(

PushMLModel.HealthNsfwProbability,

target.params(PushFeatureSwitchParams.BqmlHealthModelTypeParam)

)

// MR quality upranking param

private val qualityUprankingBoost: String = "QualityUprankingBoost"

private val producerQualityUprankingBoost: String = "ProducerQualityUprankingBoost"

private val qualityUprankingInfo: CMap[String, Double] =

new ConcurrentHashMap[String, Double]().asScala

lazy val mrQualityUprankingBoost: Option[Double] =

qualityUprankingInfo.get(qualityUprankingBoost)

lazy val mrProducerQualityUprankingBoost: Option[Double] =

qualityUprankingInfo.get(producerQualityUprankingBoost)

def setQualityUprankingBoost(boost: Double) =

if (qualityUprankingInfo.contains(qualityUprankingBoost)) {

qualityUprankingInfo(qualityUprankingBoost) = boost

} else {

qualityUprankingInfo += qualityUprankingBoost -> boost

}

def setProducerQualityUprankingBoost(boost: Double) =

if (qualityUprankingInfo.contains(producerQualityUprankingBoost)) {

qualityUprankingInfo(producerQualityUprankingBoost) = boost

} else {

qualityUprankingInfo += producerQualityUprankingBoost -> boost

}

private lazy val mrModelScoresFut: Future[Map[String, Double]] = {

if (self.target.isLoggedOutUser) {

Future.value(Map.empty[String, Double])

} else {

Future

.collectToTry {

buildModelScoresSeqForScribing()

}.map { scoreTrySeq =>

scoreTrySeq

.collect {

case result if result.isReturn => result.get()

}.reduce(\_ ++ \_)

}

}

}

// Internal model scores (scores that are independent of other candidates) for scribing

lazy val modelScores: Future[Map[String, Double]] =

target.dauProbability.flatMap { dauProbabilityOpt =>

val dauProbScoreMap = dauProbabilityOpt

.map(\_.probability).map { dauProb =>

PushMLModel.DauProbability.toString -> dauProb

}.toMap

// Avoid unnecessary MR model scribing

if (target.isDarkWrite) {

mrModelScoresFut.map(dauProbScoreMap ++ \_)

} else if (RecTypes.isSendHandlerType(commonRecType) && !RecTypes

.sendHandlerTypesUsingMrModel(commonRecType)) {

Future.value(dauProbScoreMap)

} else {

mrModelScoresFut.map(dauProbScoreMap ++ \_)

}

}

// We will scribe both internal ML scores and cross-Candidate scores

def getModelScoresforScribing(): Future[Map[String, Double]] = {

if (RecTypes.notEligibleForModelScoreTracking(commonRecType) || self.target.isLoggedOutUser) {

Future.value(Map.empty[String, Double])

} else {

modelScores.map { internalScores =>

if (internalScores.keySet.intersect(crossCandidateMlScores.keySet).nonEmpty) {

throw new Exception(

"crossCandidateMlScores overlap internalModelScores\n" +

s"internalScores keySet: ${internalScores.keySet}\n" +

s"crossCandidateScores keySet: ${crossCandidateMlScores.keySet}\n"

)

}

internalScores ++ crossCandidateMlScores

}

}

}

}