package com.twitter.follow\_recommendations.common.feature\_hydration.adapters

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

import com.twitter.hermit.constants.AlgorithmFeedbackTokens.AlgorithmToFeedbackTokenMap

import com.twitter.hermit.model.Algorithm

import com.twitter.hermit.model.Algorithm.Algorithm

import com.twitter.hermit.model.Algorithm.UttProducerOfflineMbcgV1

import com.twitter.hermit.model.Algorithm.UttProducerOnlineMbcgV1

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.Feature.SparseBinary

import com.twitter.ml.api.Feature.SparseContinuous

import com.twitter.ml.api.FeatureContext

import com.twitter.ml.api.IRecordOneToOneAdapter

import com.twitter.ml.api.util.FDsl.\_

object CandidateAlgorithmAdapter

extends IRecordOneToOneAdapter[Option[UserCandidateSourceDetails]] {

val CANDIDATE\_ALGORITHMS: SparseBinary = new SparseBinary("candidate.source.algorithm\_ids")

val CANDIDATE\_SOURCE\_SCORES: SparseContinuous =

new SparseContinuous("candidate.source.scores")

val CANDIDATE\_SOURCE\_RANKS: SparseContinuous =

new SparseContinuous("candidate.source.ranks")

override val getFeatureContext: FeatureContext = new FeatureContext(

CANDIDATE\_ALGORITHMS,

CANDIDATE\_SOURCE\_SCORES,

CANDIDATE\_SOURCE\_RANKS

)

/\*\* list of candidate source remaps to avoid creating different features for experimental sources.

\* the LHS should contain the experimental source, and the RHS should contain the prod source.

\*/

def remapCandidateSource(a: Algorithm): Algorithm = a match {

case UttProducerOnlineMbcgV1 => UttProducerOfflineMbcgV1

case \_ => a

}

// add the list of algorithm feedback tokens (integers) as a sparse binary feature

override def adaptToDataRecord(

userCandidateSourceDetailsOpt: Option[UserCandidateSourceDetails]

): DataRecord = {

val dr = new DataRecord()

userCandidateSourceDetailsOpt.foreach { userCandidateSourceDetails =>

val scoreMap = for {

(source, scoreOpt) <- userCandidateSourceDetails.candidateSourceScores

score <- scoreOpt

algo <- Algorithm.withNameOpt(source.name)

algoId <- AlgorithmToFeedbackTokenMap.get(remapCandidateSource(algo))

} yield algoId.toString -> score

val rankMap = for {

(source, rank) <- userCandidateSourceDetails.candidateSourceRanks

algo <- Algorithm.withNameOpt(source.name)

algoId <- AlgorithmToFeedbackTokenMap.get(remapCandidateSource(algo))

} yield algoId.toString -> rank.toDouble

val algoIds = scoreMap.keys.toSet ++ rankMap.keys.toSet

// hydrate if not empty

if (rankMap.nonEmpty) {

dr.setFeatureValue(CANDIDATE\_SOURCE\_RANKS, rankMap)

}

if (scoreMap.nonEmpty) {

dr.setFeatureValue(CANDIDATE\_SOURCE\_SCORES, scoreMap)

}

if (algoIds.nonEmpty) {

dr.setFeatureValue(CANDIDATE\_ALGORITHMS, algoIds)

}

}

dr

}

}