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

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.Feature

import com.twitter.ml.api.Feature.Continuous

import com.twitter.ml.api.FeatureContext

import com.twitter.ml.api.IRecordOneToOneAdapter

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

import com.twitter.ml.featurestore.catalog.features.customer\_journey.PostNuxAlgorithmFeatures

import com.twitter.ml.featurestore.catalog.features.customer\_journey.PostNuxAlgorithmIdAggregateFeatureGroup

import com.twitter.ml.featurestore.catalog.features.customer\_journey.PostNuxAlgorithmTypeAggregateFeatureGroup

import scala.collection.JavaConverters.\_

object PostNuxAlgorithmIdAdapter extends PostNuxAlgorithmAdapter {

override val PostNuxAlgorithmFeatureGroup: PostNuxAlgorithmFeatures =

PostNuxAlgorithmIdAggregateFeatureGroup

// To keep the length of feature names reasonable, we remove the prefix added by FeatureStore.

override val FeatureStorePrefix: String =

"wtf\_algorithm\_id.customer\_journey.post\_nux\_algorithm\_id\_aggregate\_feature\_group."

}

object PostNuxAlgorithmTypeAdapter extends PostNuxAlgorithmAdapter {

override val PostNuxAlgorithmFeatureGroup: PostNuxAlgorithmFeatures =

PostNuxAlgorithmTypeAggregateFeatureGroup

// To keep the length of feature names reasonable, we remove the prefix added by FeatureStore.

override val FeatureStorePrefix: String =

"wtf\_algorithm\_type.customer\_journey.post\_nux\_algorithm\_type\_aggregate\_feature\_group."

}

trait PostNuxAlgorithmAdapter extends IRecordOneToOneAdapter[DataRecord] {

val PostNuxAlgorithmFeatureGroup: PostNuxAlgorithmFeatures

// The string that is attached to the feature name when it is fetched from feature store.

val FeatureStorePrefix: String

/\*\*

\*

\* This stores transformed aggregate features for PostNux algorithm aggregate features. The

\* transformation here is log-ratio, where ratio is the raw value divided by # of impressions.

\*/

case class TransformedAlgorithmFeatures(

ratioLog: Continuous) {

def getFeatures: Seq[Continuous] = Seq(ratioLog)

}

private def applyFeatureStorePrefix(feature: Continuous) = new Continuous(

s"$FeatureStorePrefix${feature.getFeatureName}")

// The list of input features WITH the prefix assigned to them by FeatureStore.

lazy val allInputFeatures: Seq[Seq[Continuous]] = Seq(

PostNuxAlgorithmFeatureGroup.Aggregate7DayFeatures.map(applyFeatureStorePrefix),

PostNuxAlgorithmFeatureGroup.Aggregate30DayFeatures.map(applyFeatureStorePrefix)

)

// This is a list of the features WITHOUT the prefix assigned to them by FeatureStore.

lazy val outputBaseFeatureNames: Seq[Seq[Continuous]] = Seq(

PostNuxAlgorithmFeatureGroup.Aggregate7DayFeatures,

PostNuxAlgorithmFeatureGroup.Aggregate30DayFeatures

)

// We use backend impression to calculate ratio values.

lazy val ratioDenominators: Seq[Continuous] = Seq(

applyFeatureStorePrefix(PostNuxAlgorithmFeatureGroup.BackendImpressions7Days),

applyFeatureStorePrefix(PostNuxAlgorithmFeatureGroup.BackendImpressions30Days)

)

/\*\*

\* A mapping from an original feature's ID to the corresponding set of transformed features.

\* This is used to compute the transformed features for each of the original ones.

\*/

private lazy val TransformedFeaturesMap: Map[Continuous, TransformedAlgorithmFeatures] =

outputBaseFeatureNames.flatten.map { feature =>

(

// The input feature would have the FeatureStore prefix attached to it.

new Continuous(s"$FeatureStorePrefix${feature.getFeatureName}"),

// We don't keep the FeatureStore prefix to keep the length of feature names reasonable.

TransformedAlgorithmFeatures(

new Continuous(s"${feature.getFeatureName}-ratio-log")

))

}.toMap

/\*\*

\* Given a denominator, number of impressions, this function returns another function that adds

\* transformed features (log1p and ratio) of an input feature to a DataRecord.

\*/

private def addTransformedFeaturesToDataRecordFunc(

originalDr: DataRecord,

numImpressions: Double,

): (DataRecord, Continuous) => DataRecord = { (record: DataRecord, feature: Continuous) =>

{

Option(originalDr.getFeatureValue(feature)) foreach { featureValue =>

TransformedFeaturesMap.get(feature).foreach { transformedFeatures =>

record.setFeatureValue(

transformedFeatures.ratioLog,

// We don't use log1p here since the values are ratios and adding 1 to the \_ratio\_ would

// lead to logarithm of values between 1 and 2, essentially making all values the same.

math.log((featureValue + 1) / numImpressions)

)

}

}

record

}

}

/\*\*

\* @param record: The input record whose PostNuxAlgorithm aggregates are to be transformed.

\* @return the input [[DataRecord]] with transformed aggregates added.

\*/

override def adaptToDataRecord(record: DataRecord): DataRecord = {

if (record.continuousFeatures == null) {

// There are no base features available, and hence no transformations.

record

} else {

/\*\*

\* The `foldLeft` below goes through pairs of (1) Feature groups, such as those calculated over

\* 7 days or 30 days, and (2) the number of impressions for each of these groups, which is the

\* denominator when ratio is calculated.

\*/

ratioDenominators

.zip(allInputFeatures).foldLeft( /\* initial empty DataRecord \*/ record)(

(

/\* DataRecord with transformed features up to here \*/ transformedRecord,

/\* A tuple with the denominator (#impressions) and features to be transformed \*/ numImpressionsAndFeatures

) => {

val (numImpressionsFeature, features) = numImpressionsAndFeatures

Option(record.getFeatureValue(numImpressionsFeature)) match {

case Some(numImpressions) if numImpressions > 0.0 =>

/\*\*

\* With the number of impressions fixed, we generate a function that adds log-ratio

\* for each feature in the current [[DataRecord]]. The `foldLeft` goes through all

\* such features and applies that function while updating the kept DataRecord.

\*/

features.foldLeft(transformedRecord)(

addTransformedFeaturesToDataRecordFunc(record, numImpressions))

case \_ =>

transformedRecord

}

})

}

}

def getFeatures: Seq[Feature[\_]] = TransformedFeaturesMap.values.flatMap(\_.getFeatures).toSeq

override def getFeatureContext: FeatureContext =

new FeatureContext()

.addFeatures(this.getFeatures.asJava)

}