package com.twitter.product\_mixer.core.feature.featuremap.datarecord

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.FeatureContext

import com.twitter.ml.api.util.SRichDataRecord

import scala.collection.JavaConverters.\_

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

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

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

import com.twitter.product\_mixer.core.feature.featurestorev1.FeatureStoreV1Feature

import com.twitter.product\_mixer.core.feature.featurestorev1.featurevalue.FeatureStoreV1ResponseFeature

/\*\*

\* FeaturesScope for defining what features should be included in a DataRecord from a FeatureMap.

\* Where possible, prefer [[SpecificFeatures]]. It fails loudly on missing features which can help

\* identify programmer error, but can be complex to manage for multi-phase hydrators.

\*/

sealed trait FeaturesScope[+DRFeature <: BaseDataRecordFeature[\_, \_]] {

def getNonFeatureStoreDataRecordFeatures(featureMap: FeatureMap): Seq[DRFeature]

/\*\*

\* Because Feature Store features aren't direct features in the FeatureMap and instead live

\* aggregated in a DataRecord in our Feature Map, we need to interface with the underlying Data

\* Record instead. e.g. for the `AllFeatures` case, we won't know what all FStore ProMix Features

\* we have in a FeatureMap just by looping through features & need to just return the DataRecord.

\*/

def getFeatureStoreFeaturesDataRecord(featureMap: FeatureMap): SRichDataRecord

}

/\*\*

\* Use all DataRecord features on a FeatureMap to output a DataRecord.

\*/

case class AllFeatures[-Entity]() extends FeaturesScope[BaseDataRecordFeature[Entity, \_]] {

override def getNonFeatureStoreDataRecordFeatures(

featureMap: FeatureMap

): Seq[BaseDataRecordFeature[Entity, \_]] = {

/\*\*

\* See [[com.twitter.product\_mixer.core.benchmark.FeatureMapBenchmark]]

\*

\* `toSeq`` is a no-op, `view`` makes later compositions lazy. Currently we only perform a `forEach`

\* on the result but `view` here has no performance impact but protects us if we accidentally add

\* more compositions in the middle.

\*

\* Feature Store features aren't in the FeatureMap so this will only ever return the non-FStore Features.

\*/

featureMap.getFeatures.toSeq.view.collect {

case feature: BaseDataRecordFeature[Entity, \_] => feature

}

}

// Get the entire underlying DataRecord if available.

override def getFeatureStoreFeaturesDataRecord(

featureMap: FeatureMap

): SRichDataRecord = if (featureMap.getFeatures.contains(FeatureStoreV1ResponseFeature)) {

// Note, we do not copy over the feature context because JRichDataRecord will enforce that

// all features are in the FeatureContext which we do not know at init time, and it's pricey

// to compute at run time.

SRichDataRecord(featureMap.get(FeatureStoreV1ResponseFeature).richDataRecord.getRecord)

} else {

SRichDataRecord(new DataRecord())

}

}

/\*\*

\* Build a DataRecord with only the given features from the FeatureMap used. Missing features

\* will fail loudly.

\* @param features the specific features to include in the DataRecord.

\*/

case class SpecificFeatures[DRFeature <: BaseDataRecordFeature[\_, \_]](

features: Set[DRFeature])

extends FeaturesScope[DRFeature] {

private val featuresForContext = features.collect {

case featureStoreFeatures: FeatureStoreV1Feature[\_, \_, \_, \_] =>

featureStoreFeatures.boundFeature.mlApiFeature

case dataRecordCompatible: DataRecordCompatible[\_] => dataRecordCompatible.mlFeature

}.asJava

private val featureContext = new FeatureContext(featuresForContext)

private val fsFeatures = features

.collect {

case featureStoreV1Feature: FeatureStoreV1Feature[\_, \_, \_, \_] =>

featureStoreV1Feature

}

// Since it's possible a customer will pass feature store features in the DR Feature list, let's

// partition them out to only return non-FS ones in getFeatures. See [[FeaturesScope]] comment.

private val nonFsFeatures: Seq[DRFeature] = features.flatMap {

case \_: FeatureStoreV1Feature[\_, \_, \_, \_] =>

None

case otherFeature => Some(otherFeature)

}.toSeq

override def getNonFeatureStoreDataRecordFeatures(

featureMap: FeatureMap

): Seq[DRFeature] = nonFsFeatures

override def getFeatureStoreFeaturesDataRecord(

featureMap: FeatureMap

): SRichDataRecord =

if (fsFeatures.nonEmpty && featureMap.getFeatures.contains(FeatureStoreV1ResponseFeature)) {

// Return a DataRecord only with the explicitly requested features set.

val richDataRecord = SRichDataRecord(new DataRecord(), featureContext)

val existingDataRecord = featureMap.get(FeatureStoreV1ResponseFeature).richDataRecord

fsFeatures.foreach { feature =>

richDataRecord.setFeatureValue(

feature.boundFeature.mlApiFeature,

existingDataRecord.getFeatureValue(feature.boundFeature.mlApiFeature))

}

richDataRecord

} else {

SRichDataRecord(new DataRecord())

}

}

/\*\*

\* Build a DataRecord with every feature available in a FeatureMap except for the ones provided.

\* @param featuresToExclude the features to be excluded in the DataRecord.

\*/

case class AllExceptFeatures(

featuresToExclude: Set[BaseDataRecordFeature[\_, \_]])

extends FeaturesScope[BaseDataRecordFeature[\_, \_]] {

private val fsFeatures = featuresToExclude

.collect {

case featureStoreV1Feature: FeatureStoreV1Feature[\_, \_, \_, \_] =>

featureStoreV1Feature

}

override def getNonFeatureStoreDataRecordFeatures(

featureMap: FeatureMap

): Seq[BaseDataRecordFeature[\_, \_]] =

featureMap.getFeatures

.collect {

case feature: BaseDataRecordFeature[\_, \_] => feature

}.filterNot(featuresToExclude.contains).toSeq

override def getFeatureStoreFeaturesDataRecord(

featureMap: FeatureMap

): SRichDataRecord = if (featureMap.getFeatures.contains(FeatureStoreV1ResponseFeature)) {

// Return a data record only with the explicitly requested features set. Do this by copying

// the existing one and removing the features in the denylist.

// Note, we do not copy over the feature context because JRichDataRecord will enforce that

// all features are in the FeatureContext which we do not know at init time, and it's pricey

// to compute at run time.

val richDataRecord = SRichDataRecord(

featureMap.get(FeatureStoreV1ResponseFeature).richDataRecord.getRecord.deepCopy())

fsFeatures.foreach { feature =>

richDataRecord.clearFeature(feature.boundFeature.mlApiFeature)

}

richDataRecord

} else {

SRichDataRecord(new DataRecord())

}

}