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

import com.twitter.dal.personal\_data.thriftjava.PersonalDataType

import com.twitter.ml.api.Feature

import com.twitter.ml.api.DataType

import com.twitter.ml.api.thriftscala.GeneralTensor

import com.twitter.ml.api.thriftscala.StringTensor

import com.twitter.ml.api.util.ScalaToJavaDataRecordConversions

import com.twitter.ml.api.{GeneralTensor => JGeneralTensor}

import com.twitter.ml.api.{RawTypedTensor => JRawTypedTensor}

import com.twitter.ml.api.{Feature => MlFeature}

import java.nio.ByteBuffer

import java.nio.ByteOrder

import java.util.{Map => JMap}

import java.util.{Set => JSet}

import java.lang.{Long => JLong}

import java.lang.{Boolean => JBoolean}

import java.lang.{Double => JDouble}

import scala.collection.JavaConverters.\_

/\*\*

\* Defines a conversion function for customers to mix-in when constructing a DataRecord supported

\* feature. We do this because the ML Feature representation is written in Java and uses Java types.

\* Furthermore, allowing customers to construct their own ML Feature directly can leave room

\* for mistyping errors, such as using a Double ML Feature on a String Product Mixer feature.

\* This mix in enforces that the customer only uses the right types, while making it easier

\* to setup a DataRecord Feature with nothing but a feature name and personal data types.

\* @tparam FeatureValueType The type of the underlying Product Mixer feature value.

\*/

sealed trait DataRecordCompatible[FeatureValueType] {

// The feature value type in ProMix.

final type FeatureType = FeatureValueType

// The underlying DataRecord value type, sometimes this differs from the Feature Store and ProMix type.

type DataRecordType

def featureName: String

def personalDataTypes: Set[PersonalDataType]

private[product\_mixer] def mlFeature: MlFeature[DataRecordType]

/\*\*

\* To & from Data Record value converters. In most cases, this is one to one when the types match

\* but in some cases, certain features are modeled as different types in Data Record. For example,

\* some features that are Long (e.g, such as TweepCred) are sometimes stored as Doubles.

\*/

private[product\_mixer] def toDataRecordFeatureValue(featureValue: FeatureType): DataRecordType

private[product\_mixer] def fromDataRecordFeatureValue(featureValue: DataRecordType): FeatureType

}

/\*\*

\* Converter for going from String feature value to String ML Feature.

\*/

trait StringDataRecordCompatible extends DataRecordCompatible[String] {

override type DataRecordType = String

final override lazy val mlFeature: MlFeature[String] =

new MlFeature.Text(featureName, personalDataTypes.asJava)

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: String

): String = featureValue

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: String

): String = featureValue

}

/\*\*

\* Converter for going from Long feature value to Discrete/Long ML Feature.

\*/

trait LongDiscreteDataRecordCompatible extends DataRecordCompatible[Long] {

override type DataRecordType = JLong

final override lazy val mlFeature: MlFeature[JLong] =

new Feature.Discrete(featureName, personalDataTypes.asJava)

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JLong

): Long = featureValue

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: Long

): JLong = featureValue

}

/\*\*

\* Converter for going from Long feature value to Continuous/Double ML Feature.

\*/

trait LongContinuousDataRecordCompatible extends DataRecordCompatible[Long] {

override type DataRecordType = JDouble

final override lazy val mlFeature: MlFeature[JDouble] =

new Feature.Continuous(featureName, personalDataTypes.asJava)

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: FeatureType

): JDouble = featureValue.toDouble

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JDouble

): Long = featureValue.longValue()

}

/\*\*

\* Converter for going from an Integer feature value to Long/Discrete ML Feature.

\*/

trait IntDiscreteDataRecordCompatible extends DataRecordCompatible[Int] {

override type DataRecordType = JLong

final override lazy val mlFeature: MlFeature[JLong] =

new MlFeature.Discrete(featureName, personalDataTypes.asJava)

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JLong

): Int = featureValue.toInt

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: Int

): JLong = featureValue.toLong

}

/\*\*

\* Converter for going from Integer feature value to Continuous/Double ML Feature.

\*/

trait IntContinuousDataRecordCompatible extends DataRecordCompatible[Int] {

override type DataRecordType = JDouble

final override lazy val mlFeature: MlFeature[JDouble] =

new Feature.Continuous(featureName, personalDataTypes.asJava)

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: Int

): JDouble = featureValue.toDouble

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JDouble

): Int = featureValue.toInt

}

/\*\*

\* Converter for going from Double feature value to Continuous/Double ML Feature.

\*/

trait DoubleDataRecordCompatible extends DataRecordCompatible[Double] {

override type DataRecordType = JDouble

final override lazy val mlFeature: MlFeature[JDouble] =

new MlFeature.Continuous(featureName, personalDataTypes.asJava)

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JDouble

): Double = featureValue

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: Double

): JDouble = featureValue

}

/\*\*

\* Converter for going from Boolean feature value to Boolean ML Feature.

\*/

trait BoolDataRecordCompatible extends DataRecordCompatible[Boolean] {

override type DataRecordType = JBoolean

final override lazy val mlFeature: MlFeature[JBoolean] =

new MlFeature.Binary(featureName, personalDataTypes.asJava)

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JBoolean

): Boolean = featureValue

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: Boolean

): JBoolean = featureValue

}

/\*\*

\* Converter for going from a ByteBuffer feature value to ByteBuffer ML Feature.

\*/

trait BlobDataRecordCompatible extends DataRecordCompatible[ByteBuffer] {

override type DataRecordType = ByteBuffer

final override lazy val mlFeature: MlFeature[ByteBuffer] =

new Feature.Blob(featureName, personalDataTypes.asJava)

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: ByteBuffer

): ByteBuffer = featureValue

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: ByteBuffer

): ByteBuffer = featureValue

}

/\*\*

\* Converter for going from a Map[String, Double] feature value to Sparse Double/Continious ML Feature.

\*/

trait SparseContinuousDataRecordCompatible extends DataRecordCompatible[Map[String, Double]] {

override type DataRecordType = JMap[String, JDouble]

final override lazy val mlFeature: MlFeature[JMap[String, JDouble]] =

new Feature.SparseContinuous(featureName, personalDataTypes.asJava)

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: Map[String, Double]

): JMap[String, JDouble] =

featureValue.mapValues(\_.asInstanceOf[JDouble]).asJava

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JMap[String, JDouble]

) = featureValue.asScala.toMap.mapValues(\_.doubleValue())

}

/\*\*

\* Converter for going from a Set[String] feature value to SparseBinary/String Set ML Feature.

\*/

trait SparseBinaryDataRecordCompatible extends DataRecordCompatible[Set[String]] {

override type DataRecordType = JSet[String]

final override lazy val mlFeature: MlFeature[JSet[String]] =

new Feature.SparseBinary(featureName, personalDataTypes.asJava)

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JSet[String]

) = featureValue.asScala.toSet

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: FeatureType

): JSet[String] = featureValue.asJava

}

/\*\*

\* Marker trait for any feature value to Tensor ML Feature. Not directly usable.

\*/

sealed trait TensorDataRecordCompatible[FeatureV] extends DataRecordCompatible[FeatureV] {

override type DataRecordType = JGeneralTensor

override def mlFeature: MlFeature[JGeneralTensor]

}

/\*\*

\* Converter for a double to a Tensor feature encoded as float encoded RawTypedTensor

\*/

trait RawTensorFloatDoubleDataRecordCompatible extends TensorDataRecordCompatible[Double] {

final override lazy val mlFeature: MlFeature[JGeneralTensor] =

new Feature.Tensor(

featureName,

DataType.FLOAT,

List.empty[JLong].asJava,

personalDataTypes.asJava)

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: FeatureType

) = {

val byteBuffer: ByteBuffer =

ByteBuffer

.allocate(4).order(ByteOrder.LITTLE\_ENDIAN).putFloat(featureValue.toFloat)

byteBuffer.flip()

val tensor = new JGeneralTensor()

tensor.setRawTypedTensor(new JRawTypedTensor(DataType.FLOAT, byteBuffer))

tensor

}

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JGeneralTensor

) = {

val tensor = Option(featureValue.getRawTypedTensor)

.getOrElse(throw new UnexpectedTensorException(featureValue))

tensor.content.order(ByteOrder.LITTLE\_ENDIAN).getFloat().toDouble

}

}

/\*\*

\* Converter for a scala general tensor to java general tensor ML feature.

\*/

trait GeneralTensorDataRecordCompatible extends TensorDataRecordCompatible[GeneralTensor] {

def dataType: DataType

final override lazy val mlFeature: MlFeature[JGeneralTensor] =

new Feature.Tensor(featureName, dataType, List.empty[JLong].asJava, personalDataTypes.asJava)

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: FeatureType

) = ScalaToJavaDataRecordConversions.scalaTensor2Java(featureValue)

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JGeneralTensor

) = ScalaToJavaDataRecordConversions.javaTensor2Scala(featureValue)

}

/\*\*

\* Converter for a scala string tensor to java general tensor ML feature.

\*/

trait StringTensorDataRecordCompatible extends TensorDataRecordCompatible[StringTensor] {

final override lazy val mlFeature: MlFeature[JGeneralTensor] =

new Feature.Tensor(

featureName,

DataType.STRING,

List.empty[JLong].asJava,

personalDataTypes.asJava)

override private[product\_mixer] def fromDataRecordFeatureValue(

featureValue: JGeneralTensor

) = {

ScalaToJavaDataRecordConversions.javaTensor2Scala(featureValue) match {

case GeneralTensor.StringTensor(stringTensor) => stringTensor

case \_ => throw new UnexpectedTensorException(featureValue)

}

}

override private[product\_mixer] def toDataRecordFeatureValue(

featureValue: FeatureType

) = ScalaToJavaDataRecordConversions.scalaTensor2Java(GeneralTensor.StringTensor(featureValue))

}

class UnexpectedTensorException(tensor: JGeneralTensor)

extends Exception(s"Unexpected Tensor: $tensor")