package com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.conversion

import com.twitter.algebird.DecayedValue

import com.twitter.algebird.DecayedValueMonoid

import com.twitter.algebird.Monoid

import com.twitter.ml.api.\_

import com.twitter.ml.api.constant.SharedFeatures

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

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

import com.twitter.summingbird.batch.BatchID

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.AggregationKey

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.TypedAggregateGroup

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.metrics.AggregateFeature

import com.twitter.util.Duration

import java.lang.{Double => JDouble}

import java.lang.{Long => JLong}

import scala.collection.JavaConverters.\_

import scala.collection.mutable

import java.{util => ju}

object AggregatesV2Adapter {

type AggregatesV2Tuple = (AggregationKey, (BatchID, DataRecord))

val Epsilon: Double = 1e-6

val decayedValueMonoid: Monoid[DecayedValue] = DecayedValueMonoid(Epsilon)

/\*

\* Decays the storedValue from timestamp -> sourceVersion

\*

\* @param storedValue value read from the aggregates v2 output store

\* @param timestamp timestamp corresponding to store value

\* @param sourceVersion timestamp of version to decay all values to uniformly

\* @param halfLife Half life duration to use for applying decay

\*

\* By applying this function, the feature values for all users are decayed

\* to sourceVersion. This is important to ensure that a user whose aggregates

\* were updated long in the past does not have an artifically inflated count

\* compared to one whose aggregates were updated (and hence decayed) more recently.

\*/

def decayValueToSourceVersion(

storedValue: Double,

timestamp: Long,

sourceVersion: Long,

halfLife: Duration

): Double =

if (timestamp > sourceVersion) {

storedValue

} else {

decayedValueMonoid

.plus(

DecayedValue.build(storedValue, timestamp, halfLife.inMilliseconds),

DecayedValue.build(0, sourceVersion, halfLife.inMilliseconds)

)

.value

}

/\*

\* Decays all the aggregate features occurring in the ''inputRecord''

\* to a given timestamp, and mutates the ''outputRecord'' accordingly.

\* Note that inputRecord and outputRecord can be the same if you want

\* to mutate the input in place, the function does this correctly.

\*

\* @param inputRecord Input record to get features from

\* @param aggregates Aggregates to decay

\* @param decayTo Timestamp to decay to

\* @param trimThreshold Drop features below this trim threshold

\* @param outputRecord Output record to mutate

\* @return the mutated outputRecord

\*/

def mutateDecay(

inputRecord: DataRecord,

aggregateFeaturesAndHalfLives: List[(Feature[\_], Duration)],

decayTo: Long,

trimThreshold: Double,

outputRecord: DataRecord

): DataRecord = {

val timestamp = inputRecord.getFeatureValue(SharedFeatures.TIMESTAMP).toLong

aggregateFeaturesAndHalfLives.foreach {

case (aggregateFeature: Feature[\_], halfLife: Duration) =>

if (aggregateFeature.getFeatureType() == FeatureType.CONTINUOUS) {

val continuousFeature = aggregateFeature.asInstanceOf[Feature[JDouble]]

if (inputRecord.hasFeature(continuousFeature)) {

val storedValue = inputRecord.getFeatureValue(continuousFeature).toDouble

val decayedValue = decayValueToSourceVersion(storedValue, timestamp, decayTo, halfLife)

if (math.abs(decayedValue) > trimThreshold) {

outputRecord.setFeatureValue(continuousFeature, decayedValue)

}

}

}

}

/\* Update timestamp to version (now that we've decayed all aggregates) \*/

outputRecord.setFeatureValue(SharedFeatures.TIMESTAMP, decayTo)

outputRecord

}

}

class AggregatesV2Adapter(

aggregates: Set[TypedAggregateGroup[\_]],

sourceVersion: Long,

trimThreshold: Double)

extends IRecordOneToManyAdapter[AggregatesV2Adapter.AggregatesV2Tuple] {

import AggregatesV2Adapter.\_

val keyFeatures: List[Feature[\_]] = aggregates.flatMap(\_.allOutputKeys).toList

val aggregateFeatures: List[Feature[\_]] = aggregates.flatMap(\_.allOutputFeatures).toList

val timestampFeatures: List[Feature[JLong]] = List(SharedFeatures.TIMESTAMP)

val allFeatures: List[Feature[\_]] = keyFeatures ++ aggregateFeatures ++ timestampFeatures

val featureContext: FeatureContext = new FeatureContext(allFeatures.asJava)

override def getFeatureContext: FeatureContext = featureContext

val aggregateFeaturesAndHalfLives: List[(Feature[\_$3], Duration) forSome { type \_$3 }] =

aggregateFeatures.map { aggregateFeature: Feature[\_] =>

val halfLife = AggregateFeature.parseHalfLife(aggregateFeature)

(aggregateFeature, halfLife)

}

override def adaptToDataRecords(tuple: AggregatesV2Tuple): ju.List[DataRecord] = tuple match {

case (key: AggregationKey, (batchId: BatchID, record: DataRecord)) => {

val resultRecord = new SRichDataRecord(new DataRecord, featureContext)

val itr = resultRecord.continuousFeaturesIterator()

val featuresToClear = mutable.Set[Feature[JDouble]]()

while (itr.moveNext()) {

val nextFeature = itr.getFeature

if (!aggregateFeatures.contains(nextFeature)) {

featuresToClear += nextFeature

}

}

featuresToClear.foreach(resultRecord.clearFeature)

keyFeatures.foreach { keyFeature: Feature[\_] =>

if (keyFeature.getFeatureType == FeatureType.DISCRETE) {

resultRecord.setFeatureValue(

keyFeature.asInstanceOf[Feature[JLong]],

key.discreteFeaturesById(keyFeature.getDenseFeatureId)

)

} else if (keyFeature.getFeatureType == FeatureType.STRING) {

resultRecord.setFeatureValue(

keyFeature.asInstanceOf[Feature[String]],

key.textFeaturesById(keyFeature.getDenseFeatureId)

)

}

}

if (record.hasFeature(SharedFeatures.TIMESTAMP)) {

mutateDecay(

record,

aggregateFeaturesAndHalfLives,

sourceVersion,

trimThreshold,

resultRecord)

List(resultRecord.getRecord).asJava

} else {

List.empty[DataRecord].asJava

}

}

}

}