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

import com.twitter.algebird.Monoid

import com.twitter.ml.api.\_

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

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

import scala.collection.mutable

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

/\*\*

\* Monoid to aggregate over DataRecord objects.

\*

\* @param aggregates Set of ''TypedAggregateGroup'' case classes\*

\* to compute using this monoid (see TypedAggregateGroup.scala)

\*/

trait DataRecordMonoid extends Monoid[DataRecord] {

val aggregates: Set[TypedAggregateGroup[\_]]

def zero(): DataRecord = new DataRecord

/\*

\* Add two datarecords using this monoid.

\*

\* @param left Left datarecord to add

\* @param right Right datarecord to add

\* @return Sum of the two datarecords as a DataRecord

\*/

def plus(left: DataRecord, right: DataRecord): DataRecord = {

val result = zero()

aggregates.foreach(\_.mutatePlus(result, left, right))

val leftTimestamp = getTimestamp(left)

val rightTimestamp = getTimestamp(right)

SRichDataRecord(result).setFeatureValue(

SharedFeatures.TIMESTAMP,

leftTimestamp.max(rightTimestamp)

)

result

}

}

case class DataRecordAggregationMonoid(aggregates: Set[TypedAggregateGroup[\_]])

extends DataRecordMonoid {

private def sumBuffer(buffer: mutable.ArrayBuffer[DataRecord]): Unit = {

val bufferSum = zero()

buffer.toIterator.foreach { value =>

val leftTimestamp = getTimestamp(bufferSum)

val rightTimestamp = getTimestamp(value)

aggregates.foreach(\_.mutatePlus(bufferSum, bufferSum, value))

SRichDataRecord(bufferSum).setFeatureValue(

SharedFeatures.TIMESTAMP,

leftTimestamp.max(rightTimestamp)

)

}

buffer.clear()

buffer += bufferSum

}

/\*

\* Efficient batched aggregation of datarecords using

\* this monoid + a buffer, for performance.

\*

\* @param dataRecordIter An iterator of datarecords to sum

\* @return A datarecord option containing the sum

\*/

override def sumOption(dataRecordIter: TraversableOnce[DataRecord]): Option[DataRecord] = {

if (dataRecordIter.isEmpty) {

None

} else {

var buffer = mutable.ArrayBuffer[DataRecord]()

val BatchSize = 1000

dataRecordIter.foreach { u =>

if (buffer.size > BatchSize) sumBuffer(buffer)

buffer += u

}

if (buffer.size > 1) sumBuffer(buffer)

Some(buffer(0))

}

}

}

/\*

\* This class is used when there is no need to use sumBuffer functionality, as in the case of

\* online aggregation of datarecords where using a buffer on a small number of datarecords

\* would add some performance overhead.

\*/

case class DataRecordAggregationMonoidNoBuffer(aggregates: Set[TypedAggregateGroup[\_]])

extends DataRecordMonoid {}