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

import com.twitter.algebird.ScMapMonoid

import com.twitter.bijection.Injection

import com.twitter.bijection.thrift.CompactThriftCodec

import com.twitter.ml.api.util.CompactDataRecordConverter

import com.twitter.ml.api.CompactDataRecord

import com.twitter.ml.api.DataRecord

import com.twitter.scalding.commons.source.VersionedKeyValSource

import com.twitter.scalding.Args

import com.twitter.scalding.Days

import com.twitter.scalding.Duration

import com.twitter.scalding.RichDate

import com.twitter.scalding.TypedPipe

import com.twitter.scalding.TypedTsv

import com.twitter.scalding\_internal.job.HasDateRange

import com.twitter.scalding\_internal.job.analytics\_batch.AnalyticsBatchJob

import com.twitter.summingbird.batch.BatchID

import com.twitter.summingbird\_internal.bijection.BatchPairImplicits

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

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

import java.lang.{Double => JDouble}

import java.lang.{Long => JLong}

import scala.collection.JavaConverters.\_

/\*\*

\* The job takes four inputs:

\* - The path to a AggregateStore using the DataRecord format.

\* - The path to a AggregateStore using the CompactDataRecord format.

\* - A version that must be present in both sources.

\* - A sink to write the comparison statistics.

\*

\* The job reads in the two stores, converts the second one to DataRecords and

\* then compared each key to see if the two stores have identical DataRecords,

\* modulo the loss in precision on converting the Double to Float.

\*/

class AggregatesStoreComparisonJob(args: Args)

extends AnalyticsBatchJob(args)

with BatchPairImplicits

with HasDateRange {

import AggregatesStoreComparisonJob.\_

override def batchIncrement: Duration = Days(1)

override def firstTime: RichDate = RichDate(args("firstTime"))

private val dataRecordSourcePath = args("dataRecordSource")

private val compactDataRecordSourcePath = args("compactDataRecordSource")

private val version = args.long("version")

private val statsSink = args("sink")

require(dataRecordSourcePath != compactDataRecordSourcePath)

private val dataRecordSource =

VersionedKeyValSource[AggregationKey, (BatchID, DataRecord)](

path = dataRecordSourcePath,

sourceVersion = Some(version)

)

private val compactDataRecordSource =

VersionedKeyValSource[AggregationKey, (BatchID, CompactDataRecord)](

path = compactDataRecordSourcePath,

sourceVersion = Some(version)

)

private val dataRecordPipe: TypedPipe[((AggregationKey, BatchID), DataRecord)] = TypedPipe

.from(dataRecordSource)

.map { case (key, (batchId, record)) => ((key, batchId), record) }

private val compactDataRecordPipe: TypedPipe[((AggregationKey, BatchID), DataRecord)] = TypedPipe

.from(compactDataRecordSource)

.map {

case (key, (batchId, compactRecord)) =>

val record = compactConverter.compactDataRecordToDataRecord(compactRecord)

((key, batchId), record)

}

dataRecordPipe

.outerJoin(compactDataRecordPipe)

.mapValues { case (leftOpt, rightOpt) => compareDataRecords(leftOpt, rightOpt) }

.values

.sum(mapMonoid)

.flatMap(\_.toList)

.write(TypedTsv(statsSink))

}

object AggregatesStoreComparisonJob {

val mapMonoid: ScMapMonoid[String, Long] = new ScMapMonoid[String, Long]()

implicit private val aggregationKeyInjection: Injection[AggregationKey, Array[Byte]] =

AggregationKeyInjection

implicit private val aggregationKeyOrdering: Ordering[AggregationKey] = AggregationKeyOrdering

implicit private val dataRecordCodec: Injection[DataRecord, Array[Byte]] =

CompactThriftCodec[DataRecord]

implicit private val compactDataRecordCodec: Injection[CompactDataRecord, Array[Byte]] =

CompactThriftCodec[CompactDataRecord]

private val compactConverter = new CompactDataRecordConverter

val missingRecordFromLeft = "missingRecordFromLeft"

val missingRecordFromRight = "missingRecordFromRight"

val nonContinuousFeaturesDidNotMatch = "nonContinuousFeaturesDidNotMatch"

val missingFeaturesFromLeft = "missingFeaturesFromLeft"

val missingFeaturesFromRight = "missingFeaturesFromRight"

val recordsWithUnmatchedKeys = "recordsWithUnmatchedKeys"

val featureValuesMatched = "featureValuesMatched"

val featureValuesThatDidNotMatch = "featureValuesThatDidNotMatch"

val equalRecords = "equalRecords"

val keyCount = "keyCount"

def compareDataRecords(

leftOpt: Option[DataRecord],

rightOpt: Option[DataRecord]

): collection.Map[String, Long] = {

val stats = collection.Map((keyCount, 1L))

(leftOpt, rightOpt) match {

case (Some(left), Some(right)) =>

if (isIdenticalNonContinuousFeatureSet(left, right)) {

getContinuousFeaturesStats(left, right).foldLeft(stats)(mapMonoid.add)

} else {

mapMonoid.add(stats, (nonContinuousFeaturesDidNotMatch, 1L))

}

case (Some(\_), None) => mapMonoid.add(stats, (missingRecordFromRight, 1L))

case (None, Some(\_)) => mapMonoid.add(stats, (missingRecordFromLeft, 1L))

case (None, None) => throw new IllegalArgumentException("Should never be possible")

}

}

/\*\*

\* For Continuous features.

\*/

private def getContinuousFeaturesStats(

left: DataRecord,

right: DataRecord

): Seq[(String, Long)] = {

val leftFeatures = Option(left.getContinuousFeatures)

.map(\_.asScala.toMap)

.getOrElse(Map.empty[JLong, JDouble])

val rightFeatures = Option(right.getContinuousFeatures)

.map(\_.asScala.toMap)

.getOrElse(Map.empty[JLong, JDouble])

val numMissingFeaturesLeft = (rightFeatures.keySet diff leftFeatures.keySet).size

val numMissingFeaturesRight = (leftFeatures.keySet diff rightFeatures.keySet).size

if (numMissingFeaturesLeft == 0 && numMissingFeaturesRight == 0) {

val Epsilon = 1e-5

val numUnmatchedValues = leftFeatures.map {

case (id, lValue) =>

val rValue = rightFeatures(id)

// The approximate match is to account for the precision loss due to

// the Double -> Float -> Double conversion.

if (math.abs(lValue - rValue) <= Epsilon) 0L else 1L

}.sum

if (numUnmatchedValues == 0) {

Seq(

(equalRecords, 1L),

(featureValuesMatched, leftFeatures.size.toLong)

)

} else {

Seq(

(featureValuesThatDidNotMatch, numUnmatchedValues),

(

featureValuesMatched,

math.max(leftFeatures.size, rightFeatures.size) - numUnmatchedValues)

)

}

} else {

Seq(

(recordsWithUnmatchedKeys, 1L),

(missingFeaturesFromLeft, numMissingFeaturesLeft.toLong),

(missingFeaturesFromRight, numMissingFeaturesRight.toLong)

)

}

}

/\*\*

\* For feature types that are not Feature.Continuous. We expect these to match exactly in the two stores.

\* Mutable change

\*/

private def isIdenticalNonContinuousFeatureSet(left: DataRecord, right: DataRecord): Boolean = {

val booleanMatched = safeEquals(left.binaryFeatures, right.binaryFeatures)

val discreteMatched = safeEquals(left.discreteFeatures, right.discreteFeatures)

val stringMatched = safeEquals(left.stringFeatures, right.stringFeatures)

val sparseBinaryMatched = safeEquals(left.sparseBinaryFeatures, right.sparseBinaryFeatures)

val sparseContinuousMatched =

safeEquals(left.sparseContinuousFeatures, right.sparseContinuousFeatures)

val blobMatched = safeEquals(left.blobFeatures, right.blobFeatures)

val tensorsMatched = safeEquals(left.tensors, right.tensors)

val sparseTensorsMatched = safeEquals(left.sparseTensors, right.sparseTensors)

booleanMatched && discreteMatched && stringMatched && sparseBinaryMatched &&

sparseContinuousMatched && blobMatched && tensorsMatched && sparseTensorsMatched

}

def safeEquals[T](l: T, r: T): Boolean = Option(l).equals(Option(r))

}