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

import com.twitter.algebird.ScMapMonoid

import com.twitter.algebird.Semigroup

import com.twitter.ml.api.\_

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

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.Feature

import com.twitter.ml.api.FeatureType

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

import java.lang.{Long => JLong}

import scala.collection.{Map => ScMap}

object Utils {

val dataRecordMerger: DataRecordMerger = new DataRecordMerger

def EmptyDataRecord: DataRecord = new DataRecord()

private val random = scala.util.Random

private val keyedDataRecordMapMonoid = {

val dataRecordMergerSg = new Semigroup[DataRecord] {

override def plus(x: DataRecord, y: DataRecord): DataRecord = {

dataRecordMerger.merge(x, y)

x

}

}

new ScMapMonoid[Long, DataRecord]()(dataRecordMergerSg)

}

def keyFromLong(record: DataRecord, feature: Feature[JLong]): Long =

SRichDataRecord(record).getFeatureValue(feature).longValue

def keyFromString(record: DataRecord, feature: Feature[String]): Long =

try {

SRichDataRecord(record).getFeatureValue(feature).toLong

} catch {

case \_: NumberFormatException => 0L

}

def keyFromHash(record: DataRecord, feature: Feature[String]): Long =

SRichDataRecord(record).getFeatureValue(feature).hashCode.toLong

def extractSecondary[T](

record: DataRecord,

secondaryKey: Feature[T],

shouldHash: Boolean = false

): Long = secondaryKey.getFeatureType match {

case FeatureType.STRING =>

if (shouldHash) keyFromHash(record, secondaryKey.asInstanceOf[Feature[String]])

else keyFromString(record, secondaryKey.asInstanceOf[Feature[String]])

case FeatureType.DISCRETE => keyFromLong(record, secondaryKey.asInstanceOf[Feature[JLong]])

case f => throw new IllegalArgumentException(s"Feature type $f is not supported.")

}

def mergeKeyedRecordOpts(args: Option[KeyedRecord]\*): Option[KeyedRecord] = {

val keyedRecords = args.flatten

if (keyedRecords.isEmpty) {

None

} else {

val keys = keyedRecords.map(\_.aggregateType)

require(keys.toSet.size == 1, "All merged records must have the same aggregate key.")

val mergedRecord = mergeRecords(keyedRecords.map(\_.record): \_\*)

Some(KeyedRecord(keys.head, mergedRecord))

}

}

private def mergeRecords(args: DataRecord\*): DataRecord =

if (args.isEmpty) EmptyDataRecord

else {

// can just do foldLeft(new DataRecord) for both cases, but try reusing the EmptyDataRecord singleton as much as possible

args.tail.foldLeft(args.head) { (merged, record) =>

dataRecordMerger.merge(merged, record)

merged

}

}

def mergeKeyedRecordMapOpts(

opt1: Option[KeyedRecordMap],

opt2: Option[KeyedRecordMap],

maxSize: Int = Int.MaxValue

): Option[KeyedRecordMap] = {

if (opt1.isEmpty && opt2.isEmpty) {

None

} else {

val keys = Seq(opt1, opt2).flatten.map(\_.aggregateType)

require(keys.toSet.size == 1, "All merged records must have the same aggregate key.")

val mergedRecordMap = mergeMapOpts(opt1.map(\_.recordMap), opt2.map(\_.recordMap), maxSize)

Some(KeyedRecordMap(keys.head, mergedRecordMap))

}

}

private def mergeMapOpts(

opt1: Option[ScMap[Long, DataRecord]],

opt2: Option[ScMap[Long, DataRecord]],

maxSize: Int = Int.MaxValue

): ScMap[Long, DataRecord] = {

require(maxSize >= 0)

val keySet = opt1.map(\_.keySet).getOrElse(Set.empty) ++ opt2.map(\_.keySet).getOrElse(Set.empty)

val totalSize = keySet.size

val rate = if (totalSize <= maxSize) 1.0 else maxSize.toDouble / totalSize

val prunedOpt1 = opt1.map(downsample(\_, rate))

val prunedOpt2 = opt2.map(downsample(\_, rate))

Seq(prunedOpt1, prunedOpt2).flatten

.foldLeft(keyedDataRecordMapMonoid.zero)(keyedDataRecordMapMonoid.plus)

}

def downsample[K, T](m: ScMap[K, T], samplingRate: Double): ScMap[K, T] = {

if (samplingRate >= 1.0) {

m

} else if (samplingRate <= 0) {

Map.empty

} else {

m.filter {

case (key, \_) =>

// It is important that the same user with the same sampling rate be deterministically

// selected or rejected. Otherwise, mergeMapOpts will choose different keys for the

// two input maps and their union will be larger than the limit we want.

random.setSeed((key.hashCode, samplingRate.hashCode).hashCode)

random.nextDouble < samplingRate

}

}

}

}