package com.twitter.simclusters\_v2.scalding.common

import com.fasterxml.jackson.core.JsonGenerator

import com.fasterxml.jackson.databind.ObjectMapper

import com.fasterxml.jackson.databind.ObjectWriter

import com.fasterxml.jackson.module.scala.DefaultScalaModule

import com.fasterxml.jackson.module.scala.ScalaObjectMapper

import com.twitter.algebird.Aggregator

import com.twitter.algebird.Moments

import com.twitter.algebird.MultiAggregator

import com.twitter.algebird.SetSizeAggregator

import com.twitter.algebird.SketchMap

import com.twitter.algebird.SketchMapParams

import com.twitter.algebird.mutable.PriorityQueueMonoid

import com.twitter.bijection.Injection

import com.twitter.hashing.KeyHasher

import com.twitter.scalding.Execution

import com.twitter.scalding.Stat

import com.twitter.scalding.TypedPipe

import com.twitter.scalding.UniqueID

import java.io.File

import java.io.PrintWriter

import scala.sys.process.\_

object Util {

private val formatter = java.text.NumberFormat.getNumberInstance

private val jsonMapper = {

val mapper = new ObjectMapper() with ScalaObjectMapper

mapper.registerModule(DefaultScalaModule)

mapper.configure(JsonGenerator.Feature.WRITE\_NUMBERS\_AS\_STRINGS, true)

mapper

}

val prettyJsonMapper: ObjectWriter = jsonMapper.writerWithDefaultPrettyPrinter()

def getCustomCounters[T](exec: Execution[T]): Execution[Map[String, Long]] = {

exec.getCounters.map {

case (\_, counters) =>

counters.toMap.collect {

case (key, value) if key.group == "Scalding Custom" =>

key.counter -> value

}

}

}

def getCustomCountersString[T](exec: Execution[T]): Execution[String] = {

getCustomCounters(exec).map { map =>

val customCounterStrings = map.toList.map {

case (key, value) =>

s"$key:${formatter.format(value)}"

}

if (customCounterStrings.nonEmpty) {

"Printing all custom counters:\n" + customCounterStrings.mkString("\n")

} else {

"No custom counters to print"

}

}

}

// Note ideally this should not allow T that is itself Execution[U] i.e. don't accept

// nested executions

def printCounters[T](exec: Execution[T]): Execution[Unit] = {

getCustomCountersString(exec).map { s => println(s) }

}

/\*\*

\* Print some basic stats of a numeric column.

\*/

def printSummaryOfNumericColumn[V](

input: TypedPipe[V],

columnName: Option[String] = None

)(

implicit num: Numeric[V]

): Execution[String] = {

lazy val randomSampler = Aggregator.reservoirSample[V](100)

lazy val percentiles = QTreeMultiAggregator(Seq(0.05, 0.25, 0.50, 0.75, 0.95))

lazy val moments = Moments.numericAggregator

val multiAggregator = MultiAggregator(

Aggregator.size,

percentiles,

Aggregator.max,

Aggregator.min,

Aggregator.numericSum,

moments,

randomSampler

).andThenPresent {

case (size\_, percentiles\_, max\_, min\_, sum\_, moments\_, samples\_) =>

percentiles\_.mapValues(\_.toString) ++ Map(

"size" -> size\_.toString,

"max" -> max\_.toString,

"min" -> min\_.toString,

"sum" -> sum\_.toString,

"avg" -> moments\_.mean.toString,

"stddev" -> moments\_.stddev.toString,

"skewness" -> moments\_.skewness.toString,

"samples" -> samples\_.mkString(",")

)

}

input

.aggregate(multiAggregator)

.toIterableExecution

.map { m =>

val summary =

s"Column Name: $columnName\nSummary:\n${Util.prettyJsonMapper.writeValueAsString(m)}"

println(summary)

summary

}

}

/\*\*

\* Output some basic stats of a categorical column.

\*

\* Note that HeavyHitters only work when the distribution is skewed.

\*/

def printSummaryOfCategoricalColumn[V](

input: TypedPipe[V],

columnName: Option[String] = None

)(

implicit injection: Injection[V, Array[Byte]]

): Execution[String] = {

lazy val randomSampler = Aggregator.reservoirSample[V](100)

lazy val uniqueCounter = new SetSizeAggregator[V](hllBits = 13, maxSetSize = 1000)(injection)

lazy val sketchMapParams =

SketchMapParams[V](seed = 1618, eps = 0.001, delta = 0.05, heavyHittersCount = 20)(injection)

lazy val heavyHitter =

SketchMap.aggregator[V, Long](sketchMapParams).composePrepare[V](v => v -> 1L)

val multiAggregator = MultiAggregator(

Aggregator.size,

uniqueCounter,

heavyHitter,

randomSampler

).andThenPresent {

case (size\_, uniqueSize\_, heavyHitter\_, sampler\_) =>

Map(

"size" -> size\_.toString,

"unique" -> uniqueSize\_.toString,

"samples" -> sampler\_.mkString(","),

"heavyHitter" -> heavyHitter\_.heavyHitterKeys

.map { key =>

val freq = sketchMapParams.frequency(key, heavyHitter\_.valuesTable)

key -> freq

}

.sortBy(-\_.\_2).mkString(",")

)

}

input

.aggregate(multiAggregator)

.toIterableExecution

.map { m =>

val summary =

s"Column Name: $columnName\nSummary:\n${Util.prettyJsonMapper.writeValueAsString(m)}"

println(summary)

summary

}

}

val edgeOrdering: Ordering[(Long, Long)] = Ordering.by {

case (fromNodeId, toNodeId) => hashToLong(fromNodeId, toNodeId)

}

def reservoirSamplerMonoidForPairs[K, V](

sampleSize: Int

)(

implicit ord: Ordering[K]

): PriorityQueueMonoid[(K, V)] = {

implicit val fullOrdering: Ordering[(K, V)] = Ordering.by(\_.\_1)

new PriorityQueueMonoid[(K, V)](sampleSize)

}

def reservoirSamplerMonoid[T, U](

sampleSize: Int,

convert: T => U

)(

implicit ord: Ordering[U]

): PriorityQueueMonoid[T] = {

new PriorityQueueMonoid[T](sampleSize)(Ordering.by(convert))

}

def hashToLong(a: Long, b: Long): Long = {

val bb = java.nio.ByteBuffer.allocate(16)

bb.putLong(a)

bb.putLong(b)

KeyHasher.KETAMA.hashKey(bb.array())

}

def hashToLong(a: Long): Long = {

val bb = java.nio.ByteBuffer.allocate(8)

bb.putLong(a)

KeyHasher.KETAMA.hashKey(bb.array())

}

// https://en.wikipedia.org/wiki/Pearson\_correlation\_coefficient

def computeCorrelation(pairedIter: Iterator[(Double, Double)]): Double = {

val (len, xSum, ySum, x2Sum, y2Sum, xySum) =

pairedIter.foldLeft((0.0, 0.0, 0.0, 0.0, 0.0, 0.0)) {

case ((l, xs, ys, x2s, y2s, xys), (x, y)) =>

(l + 1, xs + x, ys + y, x2s + x \* x, y2s + y \* y, xys + x \* y)

}

val den = math.sqrt(len \* x2Sum - xSum \* xSum) \* math.sqrt(len \* y2Sum - ySum \* ySum)

if (den > 0) {

(len \* xySum - xSum \* ySum) / den

} else 0.0

}

// https://en.wikipedia.org/wiki/Cosine\_similarity

def cosineSimilarity(pairedIter: Iterator[(Double, Double)]): Double = {

val (xySum, x2Sum, y2Sum) = pairedIter.foldLeft(0.0, 0.0, 0.0) {

case ((xy, x2, y2), (x, y)) =>

(xy + x \* y, x2 + x \* x, y2 + y \* y)

}

val den = math.sqrt(x2Sum) \* math.sqrt(y2Sum)

if (den > 0) {

xySum / den

} else 0.0

}

case class Distribution(

avg: Double,

stdDev: Double,

p1: Double,

p10: Double,

p50: Double,

p90: Double,

p99: Double)

val emptyDist: Distribution = Distribution(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

def distributionFromArray(l: Array[Double]): Distribution = {

val s = l.sorted

val len = l.length

if (len < 1) {

emptyDist

} else {

def pctToIndex(p: Double): Int = {

val idx = math.round(l.length \* p).toInt

if (idx < 0) {

0

} else if (idx >= len) {

len - 1

} else {

idx

}

}

val (sum, sumSquared) = l.foldLeft((0.0, 0.0)) {

case ((curSum, curSumSquared), x) =>

(curSum + x, curSumSquared + x \* x)

}

val avg = sum / len

val stdDev = math.sqrt(sumSquared / len - avg \* avg)

Distribution(

avg,

stdDev,

p1 = s(pctToIndex(0.01)),

p10 = s(pctToIndex(0.1)),

p50 = s(pctToIndex(0.5)),

p90 = s(pctToIndex(0.9)),

p99 = s(pctToIndex(0.99)))

}

}

// Calculate cumulative frequency using Scalding Custom Counters.

// Increment all buckets by 1 where value <= bucket\_threshold.

case class CumulativeStat(

key: String,

buckets: Seq[Double]

)(

implicit uniqueID: UniqueID) {

val counters = buckets.map { bucket =>

bucket -> Stat(key + "\_<=" + bucket.toString)

}

def incForValue(value: Double): Unit = {

counters.foreach {

case (bucket, stat) =>

if (value <= bucket) stat.inc()

}

}

}

def sendEmail(text: String, subject: String, toAddress: String): String = {

val file = File.createTempFile("somePrefix\_", "\_someSuffix")

println(s"Email body is at ${file.getPath}")

val writer = new PrintWriter(file)

writer.write(text)

writer.close()

val mailCmd = s"cat ${file.getPath}" #| Seq("mail", "-s", subject, toAddress)

mailCmd.!!

}

}