package com.twitter.interaction\_graph.scio.common

import com.spotify.scio.ScioMetrics

import com.spotify.scio.values.SCollection

import com.twitter.interaction\_graph.scio.common.FeatureGroups.DWELL\_TIME\_FEATURE\_LIST

import com.twitter.interaction\_graph.scio.common.FeatureGroups.STATUS\_FEATURE\_LIST

import com.twitter.interaction\_graph.scio.common.UserUtil.DUMMY\_USER\_ID

import com.twitter.interaction\_graph.thriftscala.Edge

import com.twitter.interaction\_graph.thriftscala.EdgeFeature

import com.twitter.interaction\_graph.thriftscala.FeatureName

import com.twitter.interaction\_graph.thriftscala.TimeSeriesStatistics

import com.twitter.interaction\_graph.thriftscala.Vertex

import com.twitter.interaction\_graph.thriftscala.VertexFeature

object FeatureGeneratorUtil {

// Initialize a TimeSeriesStatistics object by (value, age) pair

def initializeTSS(featureValue: Double, age: Int = 1): TimeSeriesStatistics =

TimeSeriesStatistics(

mean = featureValue,

m2ForVariance = 0.0,

ewma = featureValue,

numElapsedDays = age,

numNonZeroDays = age,

numDaysSinceLast = Some(age)

)

/\*\*

\* Create vertex feature from InteractionGraphRawInput graph (src, dst, feature name, age, featureValue)

\* We will represent non-directional features (eg num\_create\_tweets) as "outgoing" values.

\* @return

\*/

def getVertexFeature(

input: SCollection[InteractionGraphRawInput]

): SCollection[Vertex] = {

// For vertex features we need to calculate both in and out featureValue

val vertexAggregatedFeatureValues = input

.flatMap { input =>

if (input.dst != DUMMY\_USER\_ID) {

Seq(

((input.src, input.name.value), (input.featureValue, 0.0)),

((input.dst, input.name.value), (0.0, input.featureValue))

)

} else {

// we put the non-directional features as "outgoing" values

Seq(((input.src, input.name.value), (input.featureValue, 0.0)))

}

}

.sumByKey

.map {

case ((userId, nameId), (outEdges, inEdges)) =>

(userId, (FeatureName(nameId), outEdges, inEdges))

}.groupByKey

vertexAggregatedFeatureValues.map {

case (userId, records) =>

// sort features by FeatureName for deterministic order (esp during testing)

val features = records.toSeq.sortBy(\_.\_1.value).flatMap {

case (name, outEdges, inEdges) =>

// create out vertex features

val outFeatures = if (outEdges > 0) {

val outTss = initializeTSS(outEdges)

List(

VertexFeature(

name = name,

outgoing = true,

tss = outTss

))

} else Nil

// create in vertex features

val inFeatures = if (inEdges > 0) {

val inTss = initializeTSS(inEdges)

List(

VertexFeature(

name = name,

outgoing = false,

tss = inTss

))

} else Nil

outFeatures ++ inFeatures

}

Vertex(userId = userId, features = features)

}

}

/\*\*

\* Create edge feature from InteractionGraphRawInput graph (src, dst, feature name, age, featureValue)

\* We will exclude all non-directional features (eg num\_create\_tweets) from all edge aggregates

\*/

def getEdgeFeature(

input: SCollection[InteractionGraphRawInput]

): SCollection[Edge] = {

input

.withName("filter non-directional features")

.flatMap { input =>

if (input.dst != DUMMY\_USER\_ID) {

ScioMetrics.counter("getEdgeFeature", s"directional feature ${input.name.name}").inc()

Some(((input.src, input.dst), (input.name, input.age, input.featureValue)))

} else {

ScioMetrics.counter("getEdgeFeature", s"non-directional feature ${input.name.name}").inc()

None

}

}

.withName("group features by pairs")

.groupByKey

.map {

case ((src, dst), records) =>

// sort features by FeatureName for deterministic order (esp during testing)

val features = records.toSeq.sortBy(\_.\_1.value).map {

case (name, age, featureValue) =>

val tss = initializeTSS(featureValue, age)

EdgeFeature(

name = name,

tss = tss

)

}

Edge(

sourceId = src,

destinationId = dst,

weight = Some(0.0),

features = features.toSeq

)

}

}

// For same user id, combine different vertex feature records into one record

// The input will assume for each (userId, featureName, direction), there will be only one record

def combineVertexFeatures(

vertex: SCollection[Vertex],

): SCollection[Vertex] = {

vertex

.groupBy { v: Vertex =>

v.userId

}

.map {

case (userId, vertexes) =>

val combiner = vertexes.foldLeft(VertexFeatureCombiner(userId)) {

case (combiner, vertex) =>

combiner.addFeature(vertex)

}

combiner.getCombinedVertex(0)

}

}

def combineEdgeFeatures(

edge: SCollection[Edge]

): SCollection[Edge] = {

edge

.groupBy { e =>

(e.sourceId, e.destinationId)

}

.withName("combining edge features for each (src, dst)")

.map {

case ((src, dst), edges) =>

val combiner = edges.foldLeft(EdgeFeatureCombiner(src, dst)) {

case (combiner, edge) =>

combiner.addFeature(edge)

}

combiner.getCombinedEdge(0)

}

}

def combineVertexFeaturesWithDecay(

history: SCollection[Vertex],

daily: SCollection[Vertex],

historyWeight: Double,

dailyWeight: Double

): SCollection[Vertex] = {

history

.keyBy(\_.userId)

.cogroup(daily.keyBy(\_.userId)).map {

case (userId, (h, d)) =>

// Adding history iterators

val historyCombiner = h.toList.foldLeft(VertexFeatureCombiner(userId)) {

case (combiner, vertex) =>

combiner.addFeature(vertex, historyWeight, 0)

}

// Adding daily iterators

val finalCombiner = d.toList.foldLeft(historyCombiner) {

case (combiner, vertex) =>

combiner.addFeature(vertex, dailyWeight, 1)

}

finalCombiner.getCombinedVertex(

2

) // 2 means totally we have 2 days(yesterday and today) data to combine together

}

}

def combineEdgeFeaturesWithDecay(

history: SCollection[Edge],

daily: SCollection[Edge],

historyWeight: Double,

dailyWeight: Double

): SCollection[Edge] = {

history

.keyBy { e =>

(e.sourceId, e.destinationId)

}

.withName("combine history and daily edges with decay")

.cogroup(daily.keyBy { e =>

(e.sourceId, e.destinationId)

}).map {

case ((src, dst), (h, d)) =>

//val combiner = EdgeFeatureCombiner(src, dst)

// Adding history iterators

val historyCombiner = h.toList.foldLeft(EdgeFeatureCombiner(src, dst)) {

case (combiner, edge) =>

combiner.addFeature(edge, historyWeight, 0)

}

val finalCombiner = d.toList.foldLeft(historyCombiner) {

case (combiner, edge) =>

combiner.addFeature(edge, dailyWeight, 1)

}

finalCombiner.getCombinedEdge(

2

) // 2 means totally we have 2 days(yesterday and today) data to combine together

}

}

/\*\*

\* Create features from following graph (src, dst, age, featureValue)

\* Note that we will filter out vertex features represented as edges from the edge output.

\*/

def getFeatures(

input: SCollection[InteractionGraphRawInput]

): (SCollection[Vertex], SCollection[Edge]) = {

(getVertexFeature(input), getEdgeFeature(input))

}

// remove the edge features that from flock, address book or sms as we will refresh them on a daily basis

def removeStatusFeatures(e: Edge): Seq[Edge] = {

val updatedFeatureList = e.features.filter { e =>

!STATUS\_FEATURE\_LIST.contains(e.name)

}

if (updatedFeatureList.size > 0) {

val edge = Edge(

sourceId = e.sourceId,

destinationId = e.destinationId,

weight = e.weight,

features = updatedFeatureList

)

Seq(edge)

} else

Nil

}

// check if the edge feature has features other than dwell time feature

def edgeWithFeatureOtherThanDwellTime(e: Edge): Boolean = {

e.features.exists { f =>

!DWELL\_TIME\_FEATURE\_LIST.contains(f.name)

}

}

}