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

import com.spotify.scio.ScioMetrics

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

object EdgeFeatureCombiner {

def apply(srcId: Long, destId: Long): EdgeFeatureCombiner = new EdgeFeatureCombiner(

instanceEdge = Edge(srcId, destId),

featureMap = Map(

FeatureName.NumRetweets -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumFavorites -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumMentions -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumTweetClicks -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumLinkClicks -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumProfileViews -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumFollows -> new BooleanOrEdgeCombiner,

FeatureName.NumUnfollows -> new BooleanOrEdgeCombiner,

FeatureName.NumMutualFollows -> new BooleanOrEdgeCombiner,

FeatureName.NumBlocks -> new BooleanOrEdgeCombiner,

FeatureName.NumMutes -> new BooleanOrEdgeCombiner,

FeatureName.NumReportAsAbuses -> new BooleanOrEdgeCombiner,

FeatureName.NumReportAsSpams -> new BooleanOrEdgeCombiner,

FeatureName.NumTweetQuotes -> new WeightedAdditiveEdgeCombiner,

FeatureName.AddressBookEmail -> new BooleanOrEdgeCombiner,

FeatureName.AddressBookPhone -> new BooleanOrEdgeCombiner,

FeatureName.AddressBookInBoth -> new BooleanOrEdgeCombiner,

FeatureName.AddressBookMutualEdgeEmail -> new BooleanOrEdgeCombiner,

FeatureName.AddressBookMutualEdgePhone -> new BooleanOrEdgeCombiner,

FeatureName.AddressBookMutualEdgeInBoth -> new BooleanOrEdgeCombiner,

FeatureName.TotalDwellTime -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumInspectedStatuses -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumPhotoTags -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumPushOpens -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumNtabClicks -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumRtMentions -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumRtReplies -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumRtRetweets -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumRtFavories -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumRtLinkClicks -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumRtTweetClicks -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumRtTweetQuotes -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumShares -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumEmailOpen -> new WeightedAdditiveEdgeCombiner,

FeatureName.NumEmailClick -> new WeightedAdditiveEdgeCombiner,

)

)

}

/\*\*

\* This class can take in a number of input Edge thrift objects, (all of which are assumed to

\* contain information about a single edge) and builds a combined Edge protobuf object, which has

\* the union of all the input.

\* <p>

\* There are two modes of aggregation: one of them just adds the values in assuming that these are

\* from the same day, and the other adds them in a time-decayed manner using the passed in weights.

\* <p>

\* The input objects features must be disjoint. Also, remember that the edge is directed!

\*/

class EdgeFeatureCombiner(instanceEdge: Edge, featureMap: Map[FeatureName, EFeatureCombiner]) {

/\*\*

\* Adds features without any decay. To be used for the same day.

\*

\* @param edge edge to be added into the combiner

\*/

def addFeature(edge: Edge): EdgeFeatureCombiner = {

val newEdge =

if (edge.weight.isDefined) instanceEdge.copy(weight = edge.weight) else instanceEdge

val newFeatures = featureMap.map {

case (featureName, combiner) =>

edge.features.find(\_.name.equals(featureName)) match {

case Some(feature) =>

val updatedCombiner =

if (combiner.isSet) combiner.updateFeature(feature) else combiner.setFeature(feature)

(featureName, updatedCombiner)

case \_ => (featureName, combiner)

}

}

new EdgeFeatureCombiner(newEdge, newFeatures)

}

/\*\*

\* Adds features with decays. Used for combining multiple days.

\*

\* @param edge edge to be added into the combiner

\* @param alpha parameters for the decay calculation

\* @param day number of days from today

\*/

def addFeature(edge: Edge, alpha: Double, day: Int): EdgeFeatureCombiner = {

val newEdge = if (edge.weight.isDefined) edge.copy(weight = edge.weight) else edge

val newFeatures = featureMap.map {

case (featureName, combiner) =>

edge.features.find(\_.name.equals(featureName)) match {

case Some(feature) =>

val updatedCombiner =

if (combiner.isSet) combiner.updateFeature(feature, alpha, day)

else combiner.setFeature(feature, alpha, day)

ScioMetrics.counter("EdgeFeatureCombiner.addFeature", feature.name.name).inc()

(featureName, updatedCombiner)

case \_ => (featureName, combiner)

}

}

new EdgeFeatureCombiner(newEdge, newFeatures)

}

/\*\*

\* Generate the final combined Edge instance

\* We return a deterministically sorted list of edge features

\*

\* @param totalDays total number of days to be combined together

\*/

def getCombinedEdge(totalDays: Int): Edge = {

val moreFeatures = featureMap.values

.flatMap { combiner =>

combiner.getFinalFeature(totalDays)

}.toList.sortBy(\_.name.value)

instanceEdge.copy(

features = moreFeatures

)

}

}

/\*\*

\* This portion contains the actual combination logic. For now, we only implement a simple

\* additive combiner, but in future we'd like to have things like time-weighted (exponential

\* decay, maybe) values.

\*/

trait EFeatureCombiner {

val edgeFeature: Option[EdgeFeature]

val startingDay: Int

val endingDay: Int

val timeSeriesStatistics: Option[TimeSeriesStatistics]

def updateTSS(feature: EdgeFeature, alpha: Double): Option[TimeSeriesStatistics]

def addToTSS(feature: EdgeFeature): Option[TimeSeriesStatistics]

def updateFeature(feature: EdgeFeature): EFeatureCombiner

def updateFeature(feature: EdgeFeature, alpha: Double, day: Int): EFeatureCombiner

def isSet: Boolean

def dropFeature: Boolean

def setFeature(feature: EdgeFeature, alpha: Double, day: Int): EFeatureCombiner

def setFeature(feature: EdgeFeature): EFeatureCombiner

def getFinalFeature(totalDays: Int): Option[EdgeFeature]

}

case class WeightedAdditiveEdgeCombiner(

override val edgeFeature: Option[EdgeFeature] = None,

override val startingDay: Int = Integer.MAX\_VALUE,

override val endingDay: Int = Integer.MIN\_VALUE,

override val timeSeriesStatistics: Option[TimeSeriesStatistics] = None)

extends EFeatureCombiner {

override def updateTSS(

feature: EdgeFeature,

alpha: Double

): Option[TimeSeriesStatistics] = {

timeSeriesStatistics.map(tss =>

InteractionGraphUtils.updateTimeSeriesStatistics(tss, feature.tss.mean, alpha))

}

override def addToTSS(feature: EdgeFeature): Option[TimeSeriesStatistics] = {

timeSeriesStatistics.map(tss =>

InteractionGraphUtils.addToTimeSeriesStatistics(tss, feature.tss.mean))

}

override def updateFeature(feature: EdgeFeature): WeightedAdditiveEdgeCombiner = {

WeightedAdditiveEdgeCombiner(

edgeFeature,

startingDay,

endingDay,

addToTSS(feature)

)

}

def setFeature(feature: EdgeFeature, alpha: Double, day: Int): WeightedAdditiveEdgeCombiner = {

val newStartingDay = Math.min(startingDay, day)

val newEndingDay = Math.max(endingDay, day)

val numDaysSinceLast =

if (feature.tss.numDaysSinceLast.exists(\_ > 0))

feature.tss.numDaysSinceLast

else Some(feature.tss.numElapsedDays - feature.tss.numNonZeroDays + 1)

val tss = feature.tss.copy(

numDaysSinceLast = numDaysSinceLast,

ewma = alpha \* feature.tss.ewma

)

val newFeature = EdgeFeature(

name = feature.name,

tss = tss

)

WeightedAdditiveEdgeCombiner(

Some(newFeature),

newStartingDay,

newEndingDay,

Some(tss)

)

}

def getFinalFeature(totalDays: Int): Option[EdgeFeature] = {

if (edgeFeature.isEmpty || dropFeature) return None

val newTss = if (totalDays > 0) {

val elapsed =

timeSeriesStatistics.map(tss => tss.numElapsedDays + totalDays - 1 - startingDay)

val latest =

if (endingDay > 0) Some(totalDays - endingDay)

else

timeSeriesStatistics.flatMap(tss =>

tss.numDaysSinceLast.map(numDaysSinceLast => numDaysSinceLast + totalDays - 1))

timeSeriesStatistics.map(tss =>

tss.copy(

numElapsedDays = elapsed.get,

numDaysSinceLast = latest

))

} else timeSeriesStatistics

edgeFeature.map(ef => ef.copy(tss = newTss.get))

}

override def updateFeature(

feature: EdgeFeature,

alpha: Double,

day: Int

): WeightedAdditiveEdgeCombiner = copy(

endingDay = Math.max(endingDay, day),

timeSeriesStatistics = updateTSS(feature, alpha)

)

override def dropFeature: Boolean = timeSeriesStatistics.exists(tss =>

tss.numDaysSinceLast.exists(\_ > InteractionGraphUtils.MAX\_DAYS\_RETENTION) ||

tss.ewma < InteractionGraphUtils.MIN\_FEATURE\_VALUE)

override def isSet = edgeFeature.isDefined

override def setFeature(feature: EdgeFeature): WeightedAdditiveEdgeCombiner =

setFeature(feature, 1.0, 0)

}

/\*\*

\* This combiner resets the value to 0 if the latest event being combined = 0. Ignores time decays.

\*/

case class BooleanOrEdgeCombiner(

override val edgeFeature: Option[EdgeFeature] = None,

override val startingDay: Int = Integer.MAX\_VALUE,

override val endingDay: Int = Integer.MIN\_VALUE,

override val timeSeriesStatistics: Option[TimeSeriesStatistics] = None)

extends EFeatureCombiner {

override def updateTSS(

feature: EdgeFeature,

alpha: Double

): Option[TimeSeriesStatistics] = {

val value = timeSeriesStatistics.map(tss => Math.floor(tss.ewma))

val newValue = if (value.exists(\_ == 1.0) || feature.tss.mean > 0.0) 1.0 else 0.0

timeSeriesStatistics.map(tss =>

tss.copy(

mean = newValue,

ewma = newValue,

numNonZeroDays = tss.numNonZeroDays + 1

))

}

override def addToTSS(feature: EdgeFeature): Option[TimeSeriesStatistics] = {

val value = timeSeriesStatistics.map(tss => Math.floor(tss.ewma))

val newValue = if (value.exists(\_ == 1.0) || feature.tss.mean > 0.0) 1.0 else 0.0

timeSeriesStatistics.map(tss => tss.copy(mean = newValue, ewma = newValue))

}

override def updateFeature(feature: EdgeFeature): BooleanOrEdgeCombiner = BooleanOrEdgeCombiner(

edgeFeature,

startingDay,

endingDay,

addToTSS(feature)

)

def setFeature(feature: EdgeFeature, alpha: Double, day: Int): BooleanOrEdgeCombiner = {

val newStartingDay = Math.min(startingDay, day)

val newEndingDay = Math.max(endingDay, day)

val numDaysSinceLast =

if (feature.tss.numDaysSinceLast.exists(\_ > 0))

feature.tss.numDaysSinceLast.get

else feature.tss.numElapsedDays - feature.tss.numNonZeroDays + 1

val tss = feature.tss.copy(

numDaysSinceLast = Some(numDaysSinceLast),

ewma = alpha \* feature.tss.ewma

)

val newFeature = EdgeFeature(

name = feature.name,

tss = tss

)

BooleanOrEdgeCombiner(

Some(newFeature),

newStartingDay,

newEndingDay,

Some(tss)

)

}

override def getFinalFeature(totalDays: Int): Option[EdgeFeature] =

if (timeSeriesStatistics.exists(tss => tss.ewma < 1.0)) None

else {

if (edgeFeature.isEmpty || dropFeature) return None

edgeFeature.map(ef =>

ef.copy(

tss = timeSeriesStatistics.get

))

}

override def updateFeature(

feature: EdgeFeature,

alpha: Double,

day: Int

): BooleanOrEdgeCombiner = copy(

endingDay = Math.max(endingDay, day),

timeSeriesStatistics = updateTSS(feature, alpha)

)

override def dropFeature: Boolean = false // we will keep rolling up status-based features

override def isSet = edgeFeature.isDefined

override def setFeature(feature: EdgeFeature): BooleanOrEdgeCombiner = setFeature(feature, 1.0, 0)

}