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

import com.google.common.annotations.VisibleForTesting

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

import com.twitter.ml.api.FeatureContext

import com.twitter.ml.api.\_

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

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

import java.lang.{Double => JDouble}

import scala.collection.JavaConverters.\_

case class CombinedFeatures(

sum: Feature[JDouble],

nonzero: Feature[JDouble],

mean: Feature[JDouble],

topK: Seq[Feature[JDouble]])

trait CombineCountsBase {

val SparseSum = "sparse\_sum"

val SparseNonzero = "sparse\_nonzero"

val SparseMean = "sparse\_mean"

val SparseTop = "sparse\_top"

def topK: Int

def hardLimit: Option[Int]

def precomputedCountFeatures: Seq[Feature[\_]]

lazy val precomputedFeaturesMap: Map[Feature[\_], CombinedFeatures] =

precomputedCountFeatures.map { countFeature =>

val derivedPersonalDataTypes =

AggregationMetricCommon.derivePersonalDataTypes(Some(countFeature))

val sum = new Feature.Continuous(

countFeature.getDenseFeatureName + "." + SparseSum,

derivedPersonalDataTypes)

val nonzero = new Feature.Continuous(

countFeature.getDenseFeatureName + "." + SparseNonzero,

derivedPersonalDataTypes)

val mean = new Feature.Continuous(

countFeature.getDenseFeatureName + "." + SparseMean,

derivedPersonalDataTypes)

val topKFeatures = (1 to topK).map { k =>

new Feature.Continuous(

countFeature.getDenseFeatureName + "." + SparseTop + k,

derivedPersonalDataTypes)

}

(countFeature, CombinedFeatures(sum, nonzero, mean, topKFeatures))

}.toMap

lazy val outputFeaturesPostMerge: Set[Feature[JDouble]] =

precomputedFeaturesMap.values.flatMap { combinedFeatures: CombinedFeatures =>

Seq(

combinedFeatures.sum,

combinedFeatures.nonzero,

combinedFeatures.mean

) ++ combinedFeatures.topK

}.toSet

private case class ComputedStats(sum: Double, nonzero: Double, mean: Double)

private def preComputeStats(featureValues: Seq[Double]): ComputedStats = {

val (sum, nonzero) = featureValues.foldLeft((0.0, 0.0)) {

case ((accSum, accNonzero), value) =>

(accSum + value, if (value > 0.0) accNonzero + 1.0 else accNonzero)

}

ComputedStats(sum, nonzero, if (nonzero > 0.0) sum / nonzero else 0.0)

}

private def computeSortedFeatureValues(featureValues: List[Double]): List[Double] =

featureValues.sortBy(-\_)

private def extractKth(sortedFeatureValues: Seq[Double], k: Int): Double =

sortedFeatureValues

.lift(k - 1)

.getOrElse(0.0)

private def setContinuousFeatureIfNonZero(

record: SRichDataRecord,

feature: Feature[JDouble],

value: Double

): Unit =

if (value != 0.0) {

record.setFeatureValue(feature, value)

}

def hydrateCountFeatures(

richRecord: SRichDataRecord,

features: Seq[Feature[\_]],

featureValuesMap: Map[Feature[\_], List[Double]]

): Unit =

for {

feature <- features

featureValues <- featureValuesMap.get(feature)

} {

mergeRecordFromCountFeature(

countFeature = feature,

featureValues = featureValues,

richInputRecord = richRecord

)

}

def mergeRecordFromCountFeature(

richInputRecord: SRichDataRecord,

countFeature: Feature[\_],

featureValues: List[Double]

): Unit = {

// In majority of calls to this method from timeline scorer

// the featureValues list is empty.

// While with empty list each operation will be not that expensive, these

// small things do add up. By adding early stop here we can avoid sorting

// empty list, allocating several options and making multiple function

// calls. In addition to that, we won't iterate over [1, topK].

if (featureValues.nonEmpty) {

val sortedFeatureValues = hardLimit

.map { limit =>

computeSortedFeatureValues(featureValues).take(limit)

}.getOrElse(computeSortedFeatureValues(featureValues)).toIndexedSeq

val computed = preComputeStats(sortedFeatureValues)

val combinedFeatures = precomputedFeaturesMap(countFeature)

setContinuousFeatureIfNonZero(

richInputRecord,

combinedFeatures.sum,

computed.sum

)

setContinuousFeatureIfNonZero(

richInputRecord,

combinedFeatures.nonzero,

computed.nonzero

)

setContinuousFeatureIfNonZero(

richInputRecord,

combinedFeatures.mean,

computed.mean

)

(1 to topK).foreach { k =>

setContinuousFeatureIfNonZero(

richInputRecord,

combinedFeatures.topK(k - 1),

extractKth(sortedFeatureValues, k)

)

}

}

}

}

object CombineCountsPolicy {

def getCountFeatures(aggregateContext: FeatureContext): Seq[Feature[\_]] =

aggregateContext.getAllFeatures.asScala.toSeq

.filter { feature =>

feature.getFeatureType == FeatureType.CONTINUOUS &&

feature.getDenseFeatureName.endsWith(TypedCountMetric[JDouble]().operatorName)

}

@VisibleForTesting

private[conversion] def getFeatureValues(

dataRecordsWithCounts: List[DataRecord],

countFeature: Feature[\_]

): List[Double] =

dataRecordsWithCounts.map(new SRichDataRecord(\_)).flatMap { record =>

Option(record.getFeatureValue(countFeature)).map(\_.asInstanceOf[JDouble].toDouble)

}

}

/\*\*

\* A merge policy that works whenever all aggregate features are

\* counts (computed using CountMetric), and typically represent

\* either impressions or engagements. For each such input count

\* feature, the policy outputs the following (3+k) derived features

\* into the output data record:

\*

\* Sum of the feature's value across all aggregate records

\* Number of aggregate records that have the feature set to non-zero

\* Mean of the feature's value across all aggregate records

\* topK values of the feature across all aggregate records

\*

\* @param topK topK values to compute

\* @param hardLimit when set, records are sorted and only the top values will be used for aggregation if

\* the number of records are higher than this hard limit.

\*/

case class CombineCountsPolicy(

override val topK: Int,

aggregateContextToPrecompute: FeatureContext,

override val hardLimit: Option[Int] = None)

extends SparseBinaryMergePolicy

with CombineCountsBase {

import CombineCountsPolicy.\_

override val precomputedCountFeatures: Seq[Feature[\_]] = getCountFeatures(

aggregateContextToPrecompute)

override def mergeRecord(

mutableInputRecord: DataRecord,

aggregateRecords: List[DataRecord],

aggregateContext: FeatureContext

): Unit = {

// Assumes aggregateContext === aggregateContextToPrecompute

mergeRecordFromCountFeatures(mutableInputRecord, aggregateRecords, precomputedCountFeatures)

}

def defaultMergeRecord(

mutableInputRecord: DataRecord,

aggregateRecords: List[DataRecord]

): Unit = {

mergeRecordFromCountFeatures(mutableInputRecord, aggregateRecords, precomputedCountFeatures)

}

def mergeRecordFromCountFeatures(

mutableInputRecord: DataRecord,

aggregateRecords: List[DataRecord],

countFeatures: Seq[Feature[\_]]

): Unit = {

val richInputRecord = new SRichDataRecord(mutableInputRecord)

countFeatures.foreach { countFeature =>

mergeRecordFromCountFeature(

richInputRecord = richInputRecord,

countFeature = countFeature,

featureValues = getFeatureValues(aggregateRecords, countFeature)

)

}

}

override def aggregateFeaturesPostMerge(aggregateContext: FeatureContext): Set[Feature[\_]] =

outputFeaturesPostMerge.map(\_.asInstanceOf[Feature[\_]])

}