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

import com.twitter.ml.api.\_

import com.twitter.ml.api.FeatureContext

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

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

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

import java.lang.{Boolean => JBoolean}

import java.lang.{Double => JDouble}

case class CtrDescriptor(

engagementFeature: Feature[JDouble],

impressionFeature: Feature[JDouble],

outputFeature: Feature[JDouble])

object PickTopCtrBuilderHelper {

def createCtrDescriptors(

aggregatePrefix: String,

engagementLabels: Set[Feature[JBoolean]],

aggregatesToCompute: Set[TypedAggregateGroup[\_]],

outputSuffix: String

): Set[CtrDescriptor] = {

val aggregateFeatures = aggregatesToCompute

.filter(\_.aggregatePrefix == aggregatePrefix)

val impressionFeature = aggregateFeatures

.flatMap { group =>

group.individualAggregateDescriptors

.filter(\_.query.feature == None)

.filter(\_.query.label == None)

.flatMap(\_.outputFeatures)

}

.head

.asInstanceOf[Feature[JDouble]]

val aggregateEngagementFeatures =

aggregateFeatures

.flatMap { group =>

group.individualAggregateDescriptors

.filter(\_.query.feature == None)

.filter { descriptor =>

//TODO: we should remove the need to pass around engagementLabels and just use all the labels available.

descriptor.query.label.exists(engagementLabels.contains(\_))

}

.flatMap(\_.outputFeatures)

}

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

aggregateEngagementFeatures

.map { aggregateEngagementFeature =>

CtrDescriptor(

engagementFeature = aggregateEngagementFeature,

impressionFeature = impressionFeature,

outputFeature = new Feature.Continuous(

aggregateEngagementFeature.getDenseFeatureName + "." + outputSuffix,

AggregationMetricCommon.derivePersonalDataTypes(

Some(aggregateEngagementFeature),

Some(impressionFeature)

)

)

)

}

}

}

object PickTopCtrPolicy {

def build(

aggregatePrefix: String,

engagementLabels: Set[Feature[JBoolean]],

aggregatesToCompute: Set[TypedAggregateGroup[\_]],

smoothing: Double = 1.0,

outputSuffix: String = "ratio"

): PickTopCtrPolicy = {

val ctrDescriptors = PickTopCtrBuilderHelper.createCtrDescriptors(

aggregatePrefix = aggregatePrefix,

engagementLabels = engagementLabels,

aggregatesToCompute = aggregatesToCompute,

outputSuffix = outputSuffix

)

PickTopCtrPolicy(

ctrDescriptors = ctrDescriptors,

smoothing = smoothing

)

}

}

object CombinedTopNCtrsByWilsonConfidenceIntervalPolicy {

def build(

aggregatePrefix: String,

engagementLabels: Set[Feature[JBoolean]],

aggregatesToCompute: Set[TypedAggregateGroup[\_]],

outputSuffix: String = "ratioWithWCI",

z: Double = 1.96,

topN: Int = 1

): CombinedTopNCtrsByWilsonConfidenceIntervalPolicy = {

val ctrDescriptors = PickTopCtrBuilderHelper.createCtrDescriptors(

aggregatePrefix = aggregatePrefix,

engagementLabels = engagementLabels,

aggregatesToCompute = aggregatesToCompute,

outputSuffix = outputSuffix

)

CombinedTopNCtrsByWilsonConfidenceIntervalPolicy(

ctrDescriptors = ctrDescriptors,

z = z,

topN = topN

)

}

}

/\*

\* A merge policy that picks the aggregate features corresponding to

\* the sparse key value with the highest engagement rate (defined

\* as the ratio of two specified features, representing engagements

\* and impressions). Also outputs the engagement rate to the specified

\* outputFeature.

\*

\* This is an abstract class. We can make variants of this policy by overriding

\* the calculateCtr method.

\*/

abstract class PickTopCtrPolicyBase(ctrDescriptors: Set[CtrDescriptor])

extends SparseBinaryMergePolicy {

private def getContinuousFeature(

aggregateRecord: DataRecord,

feature: Feature[JDouble]

): Double = {

Option(SRichDataRecord(aggregateRecord).getFeatureValue(feature))

.map(\_.asInstanceOf[JDouble].toDouble)

.getOrElse(0.0)

}

/\*\*

\* For every provided descriptor, compute the corresponding CTR feature

\* and only hydrate this result to the provided input record.

\*/

override def mergeRecord(

mutableInputRecord: DataRecord,

aggregateRecords: List[DataRecord],

aggregateContext: FeatureContext

): Unit = {

ctrDescriptors

.foreach {

case CtrDescriptor(engagementFeature, impressionFeature, outputFeature) =>

val sortedCtrs =

aggregateRecords

.map { aggregateRecord =>

val impressions = getContinuousFeature(aggregateRecord, impressionFeature)

val engagements = getContinuousFeature(aggregateRecord, engagementFeature)

calculateCtr(impressions, engagements)

}

.sortBy { ctr => -ctr }

combineTopNCtrsToSingleScore(sortedCtrs)

.foreach { score =>

SRichDataRecord(mutableInputRecord).setFeatureValue(outputFeature, score)

}

}

}

protected def calculateCtr(impressions: Double, engagements: Double): Double

protected def combineTopNCtrsToSingleScore(sortedCtrs: Seq[Double]): Option[Double]

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

ctrDescriptors

.map(\_.outputFeature)

.toSet

}

case class PickTopCtrPolicy(ctrDescriptors: Set[CtrDescriptor], smoothing: Double = 1.0)

extends PickTopCtrPolicyBase(ctrDescriptors) {

require(smoothing > 0.0)

override def calculateCtr(impressions: Double, engagements: Double): Double =

(1.0 \* engagements) / (smoothing + impressions)

override def combineTopNCtrsToSingleScore(sortedCtrs: Seq[Double]): Option[Double] =

sortedCtrs.headOption

}

case class CombinedTopNCtrsByWilsonConfidenceIntervalPolicy(

ctrDescriptors: Set[CtrDescriptor],

z: Double = 1.96,

topN: Int = 1)

extends PickTopCtrPolicyBase(ctrDescriptors) {

private val zSquared = z \* z

private val zSquaredDiv2 = zSquared / 2.0

private val zSquaredDiv4 = zSquared / 4.0

/\*\*

\* calculates the lower bound of wilson score interval. which roughly says "the actual engagement

\* rate is at least this value" with confidence designated by the z-score:

\* https://en.wikipedia.org/wiki/Binomial\_proportion\_confidence\_interval#Wilson\_score\_interval

\*/

override def calculateCtr(rawImpressions: Double, engagements: Double): Double = {

// just in case engagements happens to be more than impressions...

val impressions = Math.max(rawImpressions, engagements)

if (impressions > 0.0) {

val p = engagements / impressions

(p

+ zSquaredDiv2 / impressions

- z \* Math.sqrt(

(p \* (1.0 - p) + zSquaredDiv4 / impressions) / impressions)) / (1.0 + zSquared / impressions)

} else 0.0

}

/\*\*

\* takes the topN engagement rates, and returns the joint probability as {1.0 - Π(1.0 - p)}

\*

\* e.g. let's say you have 0.6 chance of clicking on a tweet shared by the user A.

\* you also have 0.3 chance of clicking on a tweet shared by the user B.

\* seeing a tweet shared by both A and B will not lead to 0.9 chance of you clicking on it.

\* but you could say that you have 0.4\*0.7 chance of NOT clicking on that tweet.

\*/

override def combineTopNCtrsToSingleScore(sortedCtrs: Seq[Double]): Option[Double] =

if (sortedCtrs.nonEmpty) {

val inverseLogP = sortedCtrs

.take(topN).map { p => Math.log(1.0 - p) }.sum

Some(1.0 - Math.exp(inverseLogP))

} else None

}