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

import com.twitter.ml.api.\_

import com.twitter.ml.api.constant.SharedFeatures

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

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

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

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

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

import com.twitter.timelines.data\_processing.ml\_util.transforms.OneToSomeTransform

import com.twitter.util.Duration

import com.twitter.util.Try

import java.lang.{Boolean => JBoolean}

import java.lang.{Double => JDouble}

import java.lang.{Long => JLong}

import java.util.{Set => JSet}

import scala.annotation.tailrec

import scala.language.existentials

import scala.collection.JavaConverters.\_

import scala.util.matching.Regex

/\*\*

\* A case class contained precomputed data useful to quickly

\* process operations over an aggregate.

\*

\* @param query The underlying feature being aggregated

\* @param metric The aggregation metric

\* @param outputFeatures The output features that aggregation will produce

\* @param outputFeatureIds The precomputed hashes of the above outputFeatures

\*/

case class PrecomputedAggregateDescriptor[T](

query: AggregateFeature[T],

metric: AggregationMetric[T, \_],

outputFeatures: List[Feature[\_]],

outputFeatureIds: List[JLong])

object TypedAggregateGroup {

/\*\*

\* Recursive function that generates all combinations of value

\* assignments for a collection of sparse binary features.

\*

\* @param sparseBinaryIdValues list of sparse binary feature ids and possible values they can take

\* @return A set of maps, where each map represents one possible assignment of values to ids

\*/

def sparseBinaryPermutations(

sparseBinaryIdValues: List[(Long, Set[String])]

): Set[Map[Long, String]] = sparseBinaryIdValues match {

case (id, values) +: rest =>

tailRecSparseBinaryPermutations(

existingPermutations = values.map(value => Map(id -> value)),

remainingIdValues = rest

)

case Nil => Set.empty

}

@tailrec private[this] def tailRecSparseBinaryPermutations(

existingPermutations: Set[Map[Long, String]],

remainingIdValues: List[(Long, Set[String])]

): Set[Map[Long, String]] = remainingIdValues match {

case Nil => existingPermutations

case (id, values) +: rest =>

tailRecSparseBinaryPermutations(

existingPermutations.flatMap { existingIdValueMap =>

values.map(value => existingIdValueMap ++ Map(id -> value))

},

rest

)

}

val SparseFeatureSuffix = ".member"

def sparseFeature(sparseBinaryFeature: Feature[\_]): Feature[String] =

new Feature.Text(

sparseBinaryFeature.getDenseFeatureName + SparseFeatureSuffix,

AggregationMetricCommon.derivePersonalDataTypes(Some(sparseBinaryFeature)))

/\* Throws exception if obj not an instance of U \*/

private[this] def validate[U](obj: Any): U = {

require(obj.isInstanceOf[U])

obj.asInstanceOf[U]

}

private[this] def getFeatureOpt[U](dataRecord: DataRecord, feature: Feature[U]): Option[U] =

Option(SRichDataRecord(dataRecord).getFeatureValue(feature)).map(validate[U](\_))

/\*\*

\* Get a mapping from feature ids

\* (including individual sparse elements of a sparse feature) to values

\* from the given data record, for a given feature type.

\*

\* @param dataRecord Data record to get features from

\* @param keysToAggregate key features to get id-value mappings for

\* @param featureType Feature type to get id-value maps for

\*/

def getKeyFeatureIdValues[U](

dataRecord: DataRecord,

keysToAggregate: Set[Feature[\_]],

featureType: FeatureType

): Set[(Long, Option[U])] = {

val featuresOfThisType: Set[Feature[U]] = keysToAggregate

.filter(\_.getFeatureType == featureType)

.map(validate[Feature[U]])

featuresOfThisType

.map { feature: Feature[U] =>

val featureId: Long = getDenseFeatureId(feature)

val featureOpt: Option[U] = getFeatureOpt(dataRecord, feature)

(featureId, featureOpt)

}

}

// TypedAggregateGroup may transform the aggregate keys for internal use. This method generates

// denseFeatureIds for the transformed feature.

def getDenseFeatureId(feature: Feature[\_]): Long =

if (feature.getFeatureType != FeatureType.SPARSE\_BINARY) {

feature.getDenseFeatureId

} else {

sparseFeature(feature).getDenseFeatureId

}

/\*\*

\* Return denseFeatureIds for the input features after applying the custom transformation that

\* TypedAggregateGroup applies to its keysToAggregate.

\*

\* @param keysToAggregate key features to get id for

\*/

def getKeyFeatureIds(keysToAggregate: Set[Feature[\_]]): Set[Long] =

keysToAggregate.map(getDenseFeatureId)

def checkIfAllKeysExist[U](featureIdValueMap: Map[Long, Option[U]]): Boolean =

featureIdValueMap.forall { case (\_, valueOpt) => valueOpt.isDefined }

def liftOptions[U](featureIdValueMap: Map[Long, Option[U]]): Map[Long, U] =

featureIdValueMap

.flatMap {

case (id, valueOpt) =>

valueOpt.map { value => (id, value) }

}

val timestampFeature: Feature[JLong] = SharedFeatures.TIMESTAMP

/\*\*

\* Builds all valid aggregation keys (for the output store) from

\* a datarecord and a spec listing the keys to aggregate. There

\* can be multiple aggregation keys generated from a single data

\* record when grouping by sparse binary features, for which multiple

\* values can be set within the data record.

\*

\* @param dataRecord Data record to read values for key features from

\* @return A set of AggregationKeys encoding the values of all keys

\*/

def buildAggregationKeys(

dataRecord: DataRecord,

keysToAggregate: Set[Feature[\_]]

): Set[AggregationKey] = {

val discreteAggregationKeys = getKeyFeatureIdValues[Long](

dataRecord,

keysToAggregate,

FeatureType.DISCRETE

).toMap

val textAggregationKeys = getKeyFeatureIdValues[String](

dataRecord,

keysToAggregate,

FeatureType.STRING

).toMap

val sparseBinaryIdValues = getKeyFeatureIdValues[JSet[String]](

dataRecord,

keysToAggregate,

FeatureType.SPARSE\_BINARY

).map {

case (id, values) =>

(

id,

values

.map(\_.asScala.toSet)

.getOrElse(Set.empty[String])

)

}.toList

if (checkIfAllKeysExist(discreteAggregationKeys) &&

checkIfAllKeysExist(textAggregationKeys)) {

if (sparseBinaryIdValues.nonEmpty) {

sparseBinaryPermutations(sparseBinaryIdValues).map { sparseBinaryTextKeys =>

AggregationKey(

discreteFeaturesById = liftOptions(discreteAggregationKeys),

textFeaturesById = liftOptions(textAggregationKeys) ++ sparseBinaryTextKeys

)

}

} else {

Set(

AggregationKey(

discreteFeaturesById = liftOptions(discreteAggregationKeys),

textFeaturesById = liftOptions(textAggregationKeys)

)

)

}

} else Set.empty[AggregationKey]

}

}

/\*\*

\* Specifies one or more related aggregate(s) to compute in the summingbird job.

\*

\* @param inputSource Source to compute this aggregate over

\* @param preTransforms Sequence of [[com.twitter.ml.api.RichITransform]] that transform

\* data records pre-aggregation (e.g. discretization, renaming)

\* @param samplingTransformOpt Optional [[OneToSomeTransform]] that transform data

\* record to optional data record (e.g. for sampling) before aggregation

\* @param aggregatePrefix Prefix to use for naming resultant aggregate features

\* @param keysToAggregate Features to group by when computing the aggregates

\* (e.g. USER\_ID, AUTHOR\_ID)

\* @param featuresToAggregate Features to aggregate (e.g. blender\_score or is\_photo)

\* @param labels Labels to cross the features with to make pair features, if any.

\* use Label.All if you don't want to cross with a label.

\* @param metrics Aggregation metrics to compute (e.g. count, mean)

\* @param halfLives Half lives to use for the aggregations, to be crossed with the above.

\* use Duration.Top for "forever" aggregations over an infinite time window (no decay).

\* @param outputStore Store to output this aggregate to

\* @param includeAnyFeature Aggregate label counts for any feature value

\* @param includeAnyLabel Aggregate feature counts for any label value (e.g. all impressions)

\*

\* The overall config for the summingbird job consists of a list of "AggregateGroup"

\* case class objects, which get translated into strongly typed "TypedAggregateGroup"

\* case class objects. A single TypedAggregateGroup always groups input data records from

\* ''inputSource'' by a single set of aggregation keys (''featuresToAggregate'').

\* Within these groups, we perform a comprehensive cross of:

\*

\* ''featuresToAggregate'' x ''labels'' x ''metrics'' x ''halfLives''

\*

\* All the resultant aggregate features are assigned a human-readable feature name

\* beginning with ''aggregatePrefix'', and are written to DataRecords that get

\* aggregated and written to the store specified by ''outputStore''.

\*

\* Illustrative example. Suppose we define our spec as follows:

\*

\* TypedAggregateGroup(

\* inputSource = "timelines\_recap\_daily",

\* aggregatePrefix = "user\_author\_aggregate",

\* keysToAggregate = Set(USER\_ID, AUTHOR\_ID),

\* featuresToAggregate = Set(RecapFeatures.TEXT\_SCORE, RecapFeatures.BLENDER\_SCORE),

\* labels = Set(RecapFeatures.IS\_FAVORITED, RecapFeatures.IS\_REPLIED),

\* metrics = Set(CountMetric, MeanMetric),

\* halfLives = Set(7.Days, 30.Days),

\* outputStore = "user\_author\_aggregate\_store"

\* )

\*

\* This will process data records from the source named "timelines\_recap\_daily"

\* (see AggregateSource.scala for more details on how to add your own source)

\* It will produce a total of 2x2x2x2 = 16 aggregation features, named like:

\*

\* user\_author\_aggregate.pair.recap.engagement.is\_favorited.recap.searchfeature.blender\_score.count.7days

\* user\_author\_aggregate.pair.recap.engagement.is\_favorited.recap.searchfeature.blender\_score.count.30days

\* user\_author\_aggregate.pair.recap.engagement.is\_favorited.recap.searchfeature.blender\_score.mean.7days

\*

\* ... (and so on)

\*

\* and all the result features will be stored in DataRecords, summed up, and written

\* to the output store defined by the name "user\_author\_aggregate\_store".

\* (see AggregateStore.scala for details on how to add your own store).

\*

\* If you do not want a full cross, split up your config into multiple TypedAggregateGroup

\* objects. Splitting is strongly advised to avoid blowing up and creating invalid

\* or unnecessary combinations of aggregate features (note that some combinations

\* are useless or invalid e.g. computing the mean of a binary feature). Splitting

\* also does not cost anything in terms of real-time performance, because all

\* Aggregate objects in the master spec that share the same ''keysToAggregate'', the

\* same ''inputSource'' and the same ''outputStore'' are grouped by the summingbird

\* job logic and stored into a single DataRecord in the output store. Overlapping

\* aggregates will also automatically be deduplicated so don't worry about overlaps.

\*/

case class TypedAggregateGroup[T](

inputSource: AggregateSource,

aggregatePrefix: String,

keysToAggregate: Set[Feature[\_]],

featuresToAggregate: Set[Feature[T]],

labels: Set[\_ <: Feature[JBoolean]],

metrics: Set[AggregationMetric[T, \_]],

halfLives: Set[Duration],

outputStore: AggregateStore,

preTransforms: Seq[OneToSomeTransform] = Seq.empty,

includeAnyFeature: Boolean = true,

includeAnyLabel: Boolean = true,

aggExclusionRegex: Seq[String] = Seq.empty) {

import TypedAggregateGroup.\_

val compiledRegexes = aggExclusionRegex.map(new Regex(\_))

// true if should drop, false if should keep

def filterOutAggregateFeature(

feature: PrecomputedAggregateDescriptor[\_],

regexes: Seq[Regex]

): Boolean = {

if (regexes.nonEmpty)

feature.outputFeatures.exists { feature =>

regexes.exists { re => re.findFirstMatchIn(feature.getDenseFeatureName).nonEmpty }

}

else false

}

def buildAggregationKeys(

dataRecord: DataRecord

): Set[AggregationKey] = {

TypedAggregateGroup.buildAggregationKeys(dataRecord, keysToAggregate)

}

/\*\*

\* This val precomputes descriptors for all individual aggregates in this group

\* (of type ''AggregateFeature''). Also precompute hashes of all aggregation

\* "output" features generated by these operators for faster

\* run-time performance (this turns out to be a primary CPU bottleneck).

\* Ex: for the mean operator, "sum" and "count" are output features

\*/

val individualAggregateDescriptors: Set[PrecomputedAggregateDescriptor[T]] = {

/\*

\* By default, in additional to all feature-label crosses, also

\* compute in aggregates over each feature and label without crossing

\*/

val labelOptions = labels.map(Option(\_)) ++

(if (includeAnyLabel) Set(None) else Set.empty)

val featureOptions = featuresToAggregate.map(Option(\_)) ++

(if (includeAnyFeature) Set(None) else Set.empty)

for {

feature <- featureOptions

label <- labelOptions

metric <- metrics

halfLife <- halfLives

} yield {

val query = AggregateFeature[T](aggregatePrefix, feature, label, halfLife)

val aggregateOutputFeatures = metric.getOutputFeatures(query)

val aggregateOutputFeatureIds = metric.getOutputFeatureIds(query)

PrecomputedAggregateDescriptor(

query,

metric,

aggregateOutputFeatures,

aggregateOutputFeatureIds

)

}

}.filterNot(filterOutAggregateFeature(\_, compiledRegexes))

/\* Precomputes a map from all generated aggregate feature ids to their half lives. \*/

val continuousFeatureIdsToHalfLives: Map[Long, Duration] =

individualAggregateDescriptors.flatMap { descriptor =>

descriptor.outputFeatures

.flatMap { feature =>

if (feature.getFeatureType() == FeatureType.CONTINUOUS) {

Try(feature.asInstanceOf[Feature[JDouble]]).toOption

.map(feature => (feature.getFeatureId(), descriptor.query.halfLife))

} else None

}

}.toMap

/\*

\* Sparse binary keys become individual string keys in the output.

\* e.g. group by "words.in.tweet", output key: "words.in.tweet.member"

\*/

val allOutputKeys: Set[Feature[\_]] = keysToAggregate.map { key =>

if (key.getFeatureType == FeatureType.SPARSE\_BINARY) sparseFeature(key)

else key

}

val allOutputFeatures: Set[Feature[\_]] = individualAggregateDescriptors.flatMap {

case PrecomputedAggregateDescriptor(

query,

metric,

outputFeatures,

outputFeatureIds

) =>

outputFeatures

}

val aggregateContext: FeatureContext = new FeatureContext(allOutputFeatures.toList.asJava)

/\*\*

\* Adds all aggregates in this group found in the two input data records

\* into a result, mutating the result. Uses a while loop for an

\* approximately 10% gain in speed over a for comprehension.

\*

\* WARNING: mutates ''result''

\*

\* @param result The output data record to mutate

\* @param left The left data record to add

\* @param right The right data record to add

\*/

def mutatePlus(result: DataRecord, left: DataRecord, right: DataRecord): Unit = {

val featureIterator = individualAggregateDescriptors.iterator

while (featureIterator.hasNext) {

val descriptor = featureIterator.next

descriptor.metric.mutatePlus(

result,

left,

right,

descriptor.query,

Some(descriptor.outputFeatureIds)

)

}

}

/\*\*

\* Apply preTransforms sequentially. If any transform results in a dropped (None)

\* DataRecord, then entire tranform sequence will result in a dropped DataRecord.

\* Note that preTransforms are order-dependent.

\*/

private[this] def sequentiallyTransform(dataRecord: DataRecord): Option[DataRecord] = {

val recordOpt = Option(new DataRecord(dataRecord))

preTransforms.foldLeft(recordOpt) {

case (Some(previousRecord), preTransform) =>

preTransform(previousRecord)

case \_ => Option.empty[DataRecord]

}

}

/\*\*

\* Given a data record, apply transforms and fetch the incremental contributions to

\* each configured aggregate from this data record, and store these in an output data record.

\*

\* @param dataRecord Input data record to aggregate.

\* @return A set of tuples (AggregationKey, DataRecord) whose first entry is an

\* AggregationKey indicating what keys we're grouping by, and whose second entry

\* is an output data record with incremental contributions to the aggregate value(s)

\*/

def computeAggregateKVPairs(dataRecord: DataRecord): Set[(AggregationKey, DataRecord)] = {

sequentiallyTransform(dataRecord)

.flatMap { dataRecord =>

val aggregationKeys = buildAggregationKeys(dataRecord)

val increment = new DataRecord

val isNonEmptyIncrement = individualAggregateDescriptors

.map { descriptor =>

descriptor.metric.setIncrement(

output = increment,

input = dataRecord,

query = descriptor.query,

timestampFeature = inputSource.timestampFeature,

aggregateOutputs = Some(descriptor.outputFeatureIds)

)

}

.exists(identity)

if (isNonEmptyIncrement) {

SRichDataRecord(increment).setFeatureValue(

timestampFeature,

getTimestamp(dataRecord, inputSource.timestampFeature)

)

Some(aggregationKeys.map(key => (key, increment)))

} else {

None

}

}

.getOrElse(Set.empty[(AggregationKey, DataRecord)])

}

def outputFeaturesToRenamedOutputFeatures(prefix: String): Map[Feature[\_], Feature[\_]] = {

require(prefix.nonEmpty)

allOutputFeatures.map { feature =>

if (feature.isSetFeatureName) {

val renamedFeatureName = prefix + feature.getDenseFeatureName

val personalDataTypes =

if (feature.getPersonalDataTypes.isPresent) feature.getPersonalDataTypes.get()

else null

val renamedFeature = feature.getFeatureType match {

case FeatureType.BINARY =>

new Feature.Binary(renamedFeatureName, personalDataTypes)

case FeatureType.DISCRETE =>

new Feature.Discrete(renamedFeatureName, personalDataTypes)

case FeatureType.STRING =>

new Feature.Text(renamedFeatureName, personalDataTypes)

case FeatureType.CONTINUOUS =>

new Feature.Continuous(renamedFeatureName, personalDataTypes)

case FeatureType.SPARSE\_BINARY =>

new Feature.SparseBinary(renamedFeatureName, personalDataTypes)

case FeatureType.SPARSE\_CONTINUOUS =>

new Feature.SparseContinuous(renamedFeatureName, personalDataTypes)

}

feature -> renamedFeature

} else {

feature -> feature

}

}.toMap

}

}