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.timelines.data\_processing.ml\_util.aggregation\_framework.TypedAggregateGroup

import scala.collection.JavaConverters.\_

/\*\*

\* When using the aggregates framework to group by sparse binary keys,

\* we generate different aggregate feature values for each possible

\* value of the sparse key. Hence, when joining back the aggregate

\* features with a training data set, each individual training record

\* has multiple aggregate features to choose from, for each value taken

\* by the sparse key(s) in the training record. The merge policy trait

\* below specifies how to condense/combine this variable number of

\* aggregate features into a constant number of features for training.

\* Some simple policies might be: pick the first feature set (randomly),

\* pick the top sorted by some attribute, or take some average.

\*

\* Example: suppose we group by (ADVERTISER\_ID, INTEREST\_ID) where INTEREST\_ID

\* is the sparse key, and compute a "CTR" aggregate feature for each such

\* pair measuring the click through rate on ads with (ADVERTISER\_ID, INTEREST\_ID).

\* Say we have the following aggregate records:

\*

\* (ADVERTISER\_ID = 1, INTEREST\_ID = 1, CTR = 5%)

\* (ADVERTISER\_ID = 1, INTEREST\_ID = 2, CTR = 15%)

\* (ADVERTISER\_ID = 2, INTEREST\_ID = 1, CTR = 1%)

\* (ADVERTISER\_ID = 2, INTEREST\_ID = 2, CTR = 10%)

\* ...

\* At training time, each training record has one value for ADVERTISER\_ID, but it

\* has multiple values for INTEREST\_ID e.g.

\*

\* (ADVERTISER\_ID = 1, INTEREST\_IDS = (1,2))

\*

\* There are multiple potential CTRs we can get when joining in the aggregate features:

\* in this case 2 values (5% and 15%) but in general it could be many depending on how

\* many interests the user has. When joining back the CTR features, the merge policy says how to

\* combine all these CTRs to engineer features.

\*

\* "Pick first" would say - pick some random CTR (whatever is first in the list, maybe 5%)

\* for training (probably not a good policy). "Sort by CTR" could be a policy

\* that just picks the top CTR and uses it as a feature (here 15%). Similarly, you could

\* imagine "Top K sorted by CTR" (use both 5 and 15%) or "Avg CTR" (10%) or other policies,

\* all of which are defined as objects/case classes that override this trait.

\*/

trait SparseBinaryMergePolicy {

/\*\*

\* @param mutableInputRecord Input record to add aggregates to

\* @param aggregateRecords Aggregate feature records

\* @param aggregateContext Context for aggregate records

\*/

def mergeRecord(

mutableInputRecord: DataRecord,

aggregateRecords: List[DataRecord],

aggregateContext: FeatureContext

): Unit

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

/\*\*

\* @param inputContext Context for input record

\* @param aggregateContext Context for aggregate records

\* @return Context for record returned by mergeRecord()

\*/

def mergeContext(

inputContext: FeatureContext,

aggregateContext: FeatureContext

): FeatureContext = new FeatureContext(

(inputContext.getAllFeatures.asScala.toSet ++ aggregateFeaturesPostMerge(

aggregateContext)).toSeq.asJava

)

def allOutputFeaturesPostMergePolicy[T](config: TypedAggregateGroup[T]): Set[Feature[\_]] = {

val containsSparseBinary = config.keysToAggregate

.exists(\_.getFeatureType == FeatureType.SPARSE\_BINARY)

if (!containsSparseBinary) config.allOutputFeatures

else aggregateFeaturesPostMerge(new FeatureContext(config.allOutputFeatures.toSeq.asJava))

}

}