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

import com.twitter.bijection.Bufferable

import com.twitter.bijection.Injection

import scala.util.Try

/\*\*

\* Case class that represents the "grouping" key for any aggregate feature.

\* Used by Summingbird to output aggregates to the key-value "store" using sumByKey()

\*

\* @discreteFeaturesById All discrete featureids (+ values) that are part of this key

\* @textFeaturesById All string featureids (+ values) that are part of this key

\*

\* Example 1: the user aggregate features in aggregatesv1 all group by USER\_ID,

\* which is a discrete feature. When storing these features, the key would be:

\*

\* discreteFeaturesById = Map(hash(USER\_ID) -> <the actual user id>), textFeaturesById = Map()

\*

\* Ex 2: If aggregating grouped by USER\_ID, AUTHOR\_ID, tweet link url, the key would be:

\*

\* discreteFeaturesById = Map(hash(USER\_ID) -> <actual user id>, hash(AUTHOR\_ID) -> <actual author id>),

\* textFeaturesById = Map(hash(URL\_FEATURE) -> <the link url>)

\*

\* I could have just used a DataRecord for the key, but I wanted to make it strongly typed

\* and only support grouping by discrete and string features, so using a case class instead.

\*

\* Re: efficiency, storing the hash of the feature in addition to just the feature value

\* is somewhat more inefficient than only storing the feature value in the key, but it

\* adds flexibility to group multiple types of aggregates in the same output store. If we

\* decide this isn't a good tradeoff to make later, we can reverse/refactor this decision.

\*/

case class AggregationKey(

discreteFeaturesById: Map[Long, Long],

textFeaturesById: Map[Long, String])

/\*\*

\* A custom injection for the above case class,

\* so that Summingbird knows how to store it in Manhattan.

\*/

object AggregationKeyInjection extends Injection[AggregationKey, Array[Byte]] {

/\* Injection from tuple representation of AggregationKey to Array[Byte] \*/

val featureMapsInjection: Injection[(Map[Long, Long], Map[Long, String]), Array[Byte]] =

Bufferable.injectionOf[(Map[Long, Long], Map[Long, String])]

def apply(aggregationKey: AggregationKey): Array[Byte] =

featureMapsInjection(AggregationKey.unapply(aggregationKey).get)

def invert(ab: Array[Byte]): Try[AggregationKey] =

featureMapsInjection.invert(ab).map(AggregationKey.tupled(\_))

}