package com.twitter.search.common.util.ml.prediction\_engine;

import java.util.Map;

import java.util.stream.Collectors;

import com.google.common.collect.HashMultimap;

import com.google.common.collect.Maps;

import com.google.common.collect.Multimap;

import com.twitter.ml.api.FeatureParser;

import com.twitter.ml.api.transform.DiscretizerTransform;

import com.twitter.search.common.features.thrift.ThriftSearchFeatureSchema;

import com.twitter.search.common.features.thrift.ThriftSearchFeatureSchemaEntry;

/\*\*

\* Builds a model with schema-based features, here all features are tracked by Id.

\* This class is very similar to LegacyModelBuilder, which will eventually be deprecated.

\*/

public class SchemaBasedModelBuilder extends BaseModelBuilder {

private final Map<String, FeatureData> featuresByName;

private final Map<Integer, Double> binaryFeatures;

private final Map<Integer, Double> continuousFeatures;

private final Multimap<Integer, DiscretizedFeatureRange> discretizedFeatureRanges;

/\*\*

\* a class to hold feature information

\*/

static class FeatureData {

private final ThriftSearchFeatureSchemaEntry entry;

private final int id;

public FeatureData(ThriftSearchFeatureSchemaEntry entry, int id) {

this.entry = entry;

this.id = id;

}

}

SchemaBasedModelBuilder(String modelName, ThriftSearchFeatureSchema featureSchema) {

super(modelName);

featuresByName = getFeatureDataMap(featureSchema);

binaryFeatures = Maps.newHashMap();

continuousFeatures = Maps.newHashMap();

discretizedFeatureRanges = HashMultimap.create();

}

/\*\*

\* Creates a map from feature name to thrift entries

\*/

private static Map<String, FeatureData> getFeatureDataMap(

ThriftSearchFeatureSchema schema) {

return schema.getEntries().entrySet().stream()

.collect(Collectors.toMap(

e -> e.getValue().getFeatureName(),

e -> new FeatureData(e.getValue(), e.getKey())

));

}

@Override

protected void addFeature(String baseName, double weight, FeatureParser parser) {

FeatureData feature = featuresByName.get(baseName);

if (feature != null) {

switch (feature.entry.getFeatureType()) {

case BOOLEAN\_VALUE:

binaryFeatures.put(feature.id, weight);

break;

case INT32\_VALUE:

case LONG\_VALUE:

case DOUBLE\_VALUE:

continuousFeatures.put(feature.id, weight);

break;

default:

// other values are not supported yet

throw new IllegalArgumentException(

String.format("Unsupported feature type: %s", feature));

}

} else if (baseName.endsWith(DISCRETIZER\_NAME\_SUFFIX)

&& parser.getExtension().containsKey(DiscretizerTransform.DEFAULT\_RANGE\_EXT)) {

String featureName =

baseName.substring(0, baseName.length() - DISCRETIZER\_NAME\_SUFFIX.length());

feature = featuresByName.get(featureName);

if (feature == null) {

return;

}

String rangeSpec = parser.getExtension().get(DiscretizerTransform.DEFAULT\_RANGE\_EXT);

discretizedFeatureRanges.put(feature.id, new DiscretizedFeatureRange(weight, rangeSpec));

}

}

@Override

public LightweightLinearModel build() {

Map<Integer, DiscretizedFeature> discretizedFeatures = Maps.newHashMap();

for (Integer feature : discretizedFeatureRanges.keySet()) {

DiscretizedFeature discretizedFeature =

BaseModelBuilder.buildFeature(discretizedFeatureRanges.get(feature));

if (!discretizedFeature.allValuesBelowThreshold(MIN\_WEIGHT)) {

discretizedFeatures.put(feature, discretizedFeature);

}

}

return LightweightLinearModel.createForSchemaBased(

modelName, bias, binaryFeatures, continuousFeatures, discretizedFeatures);

}

}