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

import java.io.IOException;

import java.util.List;

import java.util.Map;

import com.google.common.base.Optional;

import com.google.common.base.Supplier;

import com.google.common.base.Suppliers;

import com.google.common.collect.Lists;

import com.google.common.collect.Maps;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.search.common.file.AbstractFile;

import com.twitter.search.common.file.FileUtils;

import com.twitter.search.common.metrics.SearchCounter;

import com.twitter.search.common.metrics.SearchLongGauge;

import com.twitter.search.common.metrics.SearchStatsReceiver;

/\*\*

\* Loads LightweightLinearModel objects from a directory and provides an interface for reloading

\* them periodically.

\*

\* All the models must support the same features (defined by a FeatureContext) and they are

\* identified by the name of the subdirectory. This is the required directory structure:

\*

\* /path/to/base-directory

\* one-model/model.tsv

\* another-model/model.tsv

\* experimental-model/model.tsv

\*

\* Each subdirectory must contain a file named 'model.tsv' in the format required by

\* LightweightLinearModel.

\*/

public class ModelLoader implements Runnable {

private static final Logger LOG = LoggerFactory.getLogger(ModelLoader.class);

private static final String MODEL\_FILE\_NAME = "model.tsv";

private final CompositeFeatureContext featureContext;

private final Supplier<AbstractFile> directorySupplier;

private final Map<String, LightweightLinearModel> models;

private final Map<String, Long> lastModifiedMsByModel;

private final SearchLongGauge lastModelLoadedAtMs;

private final SearchLongGauge numModels;

private final SearchCounter numLoads;

private final SearchCounter numErrors;

/\*\*

\* Creates a new instance for a feature context and a base directory.

\*

\* It exports 4 counters:

\*

\* ${counterPrefix}\_last\_loaded:

\* Timestamp (in ms) when the last model was loaded.

\* ${counterPrefix}\_num\_models:

\* Number of models currently loaded.

\* ${counterPrefix}\_num\_loads:

\* Number of succesful model loads.

\* ${counterPrefix}\_num\_errors:

\* Number of errors occurred while loading the models.

\*/

protected ModelLoader(

CompositeFeatureContext featureContext,

Supplier<AbstractFile> directorySupplier,

String counterPrefix,

SearchStatsReceiver statsReceiver) {

this.featureContext = featureContext;

// This function returns the base directory every time we call 'run'. We use a function instead

// of using directly an AbstractFile instance, in case that we can't obtain an instance at

// initialization time (e.g. if there's an issue with HDFS).

this.directorySupplier = directorySupplier;

this.models = Maps.newConcurrentMap();

this.lastModifiedMsByModel = Maps.newConcurrentMap();

this.lastModelLoadedAtMs = statsReceiver.getLongGauge(counterPrefix + "last\_loaded");

this.numModels = statsReceiver.getLongGauge(counterPrefix + "num\_models");

this.numLoads = statsReceiver.getCounter(counterPrefix + "num\_loads");

this.numErrors = statsReceiver.getCounter(counterPrefix + "num\_errors");

}

public Optional<LightweightLinearModel> getModel(String name) {

return Optional.fromNullable(models.get(name));

}

/\*\*

\* Loads the models from the base directory.

\*

\* It doesn't load a model if its file has not been modified since the last time it was loaded.

\*

\* This method doesn't delete previously loaded models if their directories are not available.

\*/

@Override

public void run() {

try {

AbstractFile baseDirectory = directorySupplier.get();

List<AbstractFile> modelDirectories =

Lists.newArrayList(baseDirectory.listFiles(IS\_MODEL\_DIR));

for (AbstractFile directory : modelDirectories) {

try {

// Note that the modelName is the directory name, if it ends with ".schema\_based", the

// model will be loaded as a schema-based model.

String modelName = directory.getName();

AbstractFile modelFile = directory.getChild(MODEL\_FILE\_NAME);

long currentLastModified = modelFile.getLastModified();

Long lastModified = lastModifiedMsByModel.get(modelName);

if (lastModified == null || lastModified < currentLastModified) {

LightweightLinearModel model =

LightweightLinearModel.load(modelName, featureContext, modelFile);

if (!models.containsKey(modelName)) {

LOG.info("Loading model {}.", modelName);

}

models.put(modelName, model);

lastModifiedMsByModel.put(modelName, currentLastModified);

lastModelLoadedAtMs.set(System.currentTimeMillis());

numLoads.increment();

LOG.debug("Model: {}", model);

} else {

LOG.debug("Directory for model {} has not changed.", modelName);

}

} catch (Exception e) {

LOG.error("Error loading model from directory: " + directory.getPath(), e);

this.numErrors.increment();

}

}

if (numModels.get() != models.size()) {

LOG.info("Finished loading models. Model names: {}", models.keySet());

}

this.numModels.set(models.size());

} catch (IOException e) {

LOG.error("Error loading models", e);

this.numErrors.increment();

}

}

/\*\*

\* Creates an instance that loads models from a directory (local or from HDFS).

\*/

public static ModelLoader forDirectory(

final AbstractFile directory,

CompositeFeatureContext featureContext,

String counterPrefix,

SearchStatsReceiver statsReceiver) {

Supplier<AbstractFile> directorySupplier = Suppliers.ofInstance(directory);

return new ModelLoader(featureContext, directorySupplier, counterPrefix, statsReceiver);

}

/\*\*

\* Creates an instance that loads models from HDFS.

\*/

public static ModelLoader forHdfsDirectory(

final String nameNode,

final String directory,

CompositeFeatureContext featureContext,

String counterPrefix,

SearchStatsReceiver statsReceiver) {

Supplier<AbstractFile> directorySupplier =

() -> FileUtils.getHdfsFileHandle(directory, nameNode);

return new ModelLoader(featureContext, directorySupplier, counterPrefix, statsReceiver);

}

private static final AbstractFile.Filter IS\_MODEL\_DIR = file -> {

try {

if (file.isDirectory()) {

AbstractFile modelFile = file.getChild(MODEL\_FILE\_NAME);

return (modelFile != null) && modelFile.canRead();

}

} catch (IOException e) {

LOG.error("Error reading file: " + file, e);

}

return false;

};

}