# pylint: disable=arguments-differ,no-member,too-many-statements

''' Contains MDLFeature and MDLCalibrator used for MDL calibration '''

import os

from .percentile\_discretizer import PercentileDiscretizerCalibrator, PercentileDiscretizerFeature

from absl import logging

import numpy as np

import tensorflow.compat.v1 as tf

import twml

import twml.layers

DEFAULT\_SAMPLE\_WEIGHT = 1

class MDLFeature(PercentileDiscretizerFeature):

''' Accumulates and calibrates a single sparse MDL feature. '''

class MDLCalibrator(PercentileDiscretizerCalibrator):

''' Accumulates features and their respective values for MDL calibration.

Internally, each feature's values is accumulated via its own ``MDLFeature`` object.

The steps for calibration are typically as follows:

1. accumulate feature values from batches by calling ``accumulate()``;

2. calibrate all feature into MDL bin\_vals by calling ``calibrate()``; and

3. convert to a twml.layers.MDL layer by calling ``to\_layer()``.

'''

def to\_layer(self, name=None):

"""

Returns a twml.layers.PercentileDiscretizer Layer

that can be used for feature discretization.

Arguments:

name:

name-scope of the PercentileDiscretizer layer

"""

n\_feature = len(self.\_discretizer\_feature\_dict)

max\_discretizer\_feature = n\_feature \* (self.\_n\_bin + 1)

if not self.\_calibrated:

raise RuntimeError("Expecting prior call to calibrate()")

if self.\_bin\_ids.shape[0] != n\_feature:

raise RuntimeError("Expecting self.\_bin\_ids.shape[0] \

!= len(self.\_discretizer\_feature\_dict)")

if self.\_bin\_vals.shape[0] != n\_feature:

raise RuntimeError("Expecting self.\_bin\_vals.shape[0] \

!= len(self.\_discretizer\_feature\_dict)")

# can add at most #features \* (n\_bin+1) new feature ids

if 2\*\*self.\_out\_bits <= max\_discretizer\_feature:

raise ValueError("""Maximum number of features created by discretizer is

%d but requested that the output be limited to %d values (%d bits),

which is smaller than that. Please ensure the output has enough bits

to represent at least the new features"""

% (max\_discretizer\_feature, 2\*\*self.\_out\_bits, self.\_out\_bits))

# build feature\_offsets, hash\_map\_keys, hash\_map\_values

feature\_offsets = np.arange(0, max\_discretizer\_feature,

self.\_n\_bin + 1, dtype='int64')

hash\_map\_keys = np.array(list(self.\_hash\_map.keys()), dtype=np.int64)

hash\_map\_values = np.array(list(self.\_hash\_map.values()), dtype=np.float32)

discretizer = twml.layers.MDL(

n\_feature=n\_feature, n\_bin=self.\_n\_bin,

name=name, out\_bits=self.\_out\_bits,

hash\_keys=hash\_map\_keys, hash\_values=hash\_map\_values,

bin\_ids=self.\_bin\_ids.flatten(), bin\_values=self.\_bin\_vals.flatten(),

feature\_offsets=feature\_offsets,

\*\*self.\_kwargs

)

return discretizer

def save(self, save\_dir, name='calibrator', verbose=False):

'''Save the calibrator into the given save\_directory.

Arguments:

save\_dir:

name of the saving directory

name:

name for the graph scope. Passed to to\_layer(name=name) to set

scope of layer.

'''

if not self.\_calibrated:

raise RuntimeError("Expecting prior call to calibrate().Cannot save() prior to calibrate()")

layer\_args = self.get\_layer\_args()

calibrator\_filename = os.path.join(save\_dir, name + '.json.tf')

calibrator\_dict = {

'layer\_args': layer\_args,

'saved\_layer\_scope': name + '/',

}

twml.write\_file(calibrator\_filename, calibrator\_dict, encode='json')

if verbose:

logging.info("The layer graph and other information necessary ")

logging.info("for multi-phase training is saved in directory:")

logging.info(save\_dir)

logging.info("This directory can be specified as --init\_from\_dir argument.")

logging.info("")

logging.info("Other information is available in: %s.json.tf", name)

logging.info("This file can be loaded with twml.read\_file(decode='json) to obtain ")

logging.info("layer\_args, saved\_layer\_scope and variable\_names")

graph = tf.Graph()

# save graph for tensorboard as well

writer = tf.summary.FileWriter(logdir=save\_dir, graph=graph)

with tf.Session(graph=graph) as sess:

self.write\_summary(writer, sess)

writer.flush()