# pylint: disable=missing-docstring, unused-argument

''' Contains the base classes for CalibrationFeature and Calibrator '''

from collections import defaultdict

import numpy as np

import tensorflow.compat.v1 as tf

import tensorflow\_hub as hub

import twml

import twml.util

class CalibrationFeature(object):

'''

Accumulates values and weights for individual features.

Typically, each unique feature defined in the accumulated SparseTensor or Tensor

would have its own CalibrationFeature instance.

'''

def \_\_init\_\_(self, feature\_id):

''' Constructs a CalibrationFeature

Arguments:

feature\_id:

number identifying the feature.

'''

self.feature\_id = feature\_id

self.\_calibrated = False

self.\_features\_dict = defaultdict(list)

def add\_values(self, new\_features):

'''

Extends lists to contain the values in this batch

'''

for key in new\_features:

self.\_features\_dict[key].append(new\_features[key])

def \_concat\_arrays(self):

'''

This class calls this function after you have added all the values.

It creates a dictionary with the concatanated arrays

'''

self.\_features\_dict.update((k, np.concatenate(v)) for k, v in self.\_features\_dict.items())

def calibrate(self, \*args, \*\*kwargs):

raise NotImplementedError

class Calibrator(object):

'''

Accumulates features and their respective values for Calibration

The steps for calibration are typically as follows:

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

2. calibrate by calling ``calibrate()``;

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

Note you can only use one calibrator per Trainer.

'''

def \_\_init\_\_(self, calibrator\_name=None, \*\*kwargs):

'''

Arguments:

calibrator\_name.

Default: if set to None it will be the same as the class name.

Please be reminded that if in the model there are many calibrators

of the same type the calibrator\_name should be changed to avoid confusion.

'''

self.\_calibrated = False

if calibrator\_name is None:

calibrator\_name = twml.util.to\_snake\_case(self.\_\_class\_\_.\_\_name\_\_)

self.\_calibrator\_name = calibrator\_name

self.\_kwargs = kwargs

@property

def is\_calibrated(self):

return self.\_calibrated

@property

def name(self):

return self.\_calibrator\_name

def accumulate(self, \*args, \*\*kwargs):

'''Accumulates features and their respective values for Calibration.'''

raise NotImplementedError

def calibrate(self):

'''Calibrates after the accumulation has ended.'''

self.\_calibrated = True

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

'''

Returns a twml.layers.Layer instance with the result of calibrator.

Arguments:

name:

name-scope of the layer

'''

raise NotImplementedError

def get\_layer\_args(self):

'''

Returns layer arguments required to implement multi-phase training.

Returns:

dictionary of Layer constructor arguments to initialize the

layer Variables. Typically, this should contain enough information

to initialize empty layer Variables of the correct size, which will then

be filled with the right data using init\_map.

'''

raise NotImplementedError

def save(self, save\_dir, name="default", verbose=False):

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

Arguments:

save\_dir:

name of the saving directory. Default (string): "default".

name:

name for the calibrator.

'''

if not self.\_calibrated:

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

# This module allows for the calibrator to save be saved as part of

# Tensorflow Hub (this will allow it to be used in further steps)

def calibrator\_module():

# Note that this is usually expecting a sparse\_placeholder

inputs = tf.sparse\_placeholder(tf.float32)

calibrator\_layer = self.to\_layer()

output = calibrator\_layer(inputs)

# creates the signature to the calibrator module

hub.add\_signature(inputs=inputs, outputs=output, name=name)

# exports the module to the save\_dir

spec = hub.create\_module\_spec(calibrator\_module)

with tf.Graph().as\_default():

module = hub.Module(spec)

with tf.Session() as session:

module.export(save\_dir, session)

def write\_summary(self, writer, sess=None):

"""

This method is called by save() to write tensorboard summaries to disk.

See MDLCalibrator.write\_summary for an example.

By default, the method does nothing. It can be overloaded by child-classes.

Arguments:

writer:

`tf.summary.FilteWriter

<https://www.tensorflow.org/versions/master/api\_docs/python/tf/summary/FileWriter>`\_

instance.

The ``writer`` is used to add summaries to event files for inclusion in tensorboard.

sess (optional):

`tf.Session <https://www.tensorflow.org/versions/master/api\_docs/python/tf/Session>`\_

instance. The ``sess`` is used to produces summaries for the writer.

"""