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

''' Contains PercentileDiscretizerFeature and PercentileDiscretizerCalibrator used \

for PercentileDiscretizer calibration '''

from .calibrator import CalibrationFeature, Calibrator

import os

import numpy as np

import tensorflow.compat.v1 as tf

import tensorflow\_hub as hub

import twml

import twml.layers

DEFAULT\_SAMPLE\_WEIGHT = 1

class PercentileDiscretizerFeature(CalibrationFeature):

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

@staticmethod

def \_gather\_debug\_info(values, indices, bin\_vals, bin\_counts\_buffer):

'''

Determine how many training values fell into a given bin during calibration.

This is calculated by finding the index of the first appearance of each bin

boundary in values (values may repeat, so that isn't trivially in indices.)

Subtracting each bin boundary index from the next tells you how many values fall in

that bin.

To get this to calculate the last bin correctly, len(values) is appended to the

list of bound indices.

This assumes that ``bin\_vals`` excludes np.inf bin boundaries when

PercentileDiscretizer was calibrated

with fewer values than bins.

Arguments:

values:

1D ndarray of the PercentileDiscretizerFeature's accumulated values, sorted ascending

indices:

1D int32 ndarray of the indices (in values) of the bin boundaries

bin\_vals:

1D ndarray containing the bin boundaries

bin\_counts\_buffer:

ndarray buffer for returning the PercentileDiscretizer histogram

'''

# np.flatnonzero(np.diff(x)) gives you the indices i in x s.t. x[i] != x[i+1]

# append index of the last bin since that cannot be empty with how

# PercentileDiscretizer is implemented

nonempty\_bins = np.append(np.flatnonzero(np.diff(bin\_vals)), len(bin\_vals) - 1)

bin\_start\_indices = indices.take(nonempty\_bins)

# if multiples of a bin's lower bound value exist, find the first one

for (i, idx) in enumerate(bin\_start\_indices):

cur\_idx = idx

while cur\_idx > 0 and values[cur\_idx] == values[cur\_idx - 1]:

bin\_start\_indices[i] = cur\_idx = cur\_idx - 1

# the end of each bin is the start of the next bin,

# until the last, which is the end of the array

# broadcast the counts to the nonempty bins, 0 otherwise

bin\_counts\_buffer[:] = 0

bin\_counts\_buffer[nonempty\_bins] = np.diff(np.append(bin\_start\_indices, values.size))

def calibrate(

self,

bin\_vals, percentiles, percentile\_indices,

bin\_counts\_buffer=None):

'''Calibrates the PercentileDiscretizerFeature into bin values for

use in PercentileDiscretizerCalibrator.

Note that this method can only be called once.

Arguments:

bin\_vals:

Row in the PercentileDiscretizerCalibrator.bin\_vals matrix corresponding to this feature.

Will be updated with the results of the calibration.

A 1D ndarray.

percentiles:

1D array of size n\_bin with values ranging from 0 to 1.

For example, ``percentiles = np.linspace(0, 1, num=self.\_n\_bin+1, dtype=np.float32)``

percentile\_indices:

Empty 1D array of size n\_bin used to store intermediate results when

calling twml.twml\_optim\_nearest\_interpolation().

For example, np.empty(self.\_n\_bin + 1, dtype=np.float32).

bin\_counts\_buffer:

optional ndarray buffer used for retaining count of values per PercentileDiscretizer

bucket (for debug and feature exploration purposes)

Returns:

calibrated bin\_vals for use by ``PercentileDiscretizerCalibrator``

'''

if self.\_calibrated:

raise RuntimeError("Can only calibrate once")

if bin\_vals.ndim != 1:

raise RuntimeError("Expecting bin\_vals row")

# # concatenate values and weights buffers

self.\_concat\_arrays()

feature\_values = self.\_features\_dict['values']

feature\_weights = self.\_features\_dict['weights']

# get features ready for the bins, order array indices by feature values.

indices = np.argsort(feature\_values)

# get ordered values and weights using array indices

values = feature\_values.take(indices)

weights = feature\_weights.take(indices)

# Normalizes the sum of weights to be between 0 and 1

weights = np.cumsum(weights, out=feature\_weights)

weights -= weights[0]

if weights[-1] > 0: # prevent zero-division

weights /= weights[-1]

# Check if we have less values than bin\_vals

if values.size < bin\_vals.size:

# Fills all the bins with a value that won't ever be reached

bin\_vals.fill(np.inf)

# Forces the first to be -inf

bin\_vals[0] = -np.inf

# Copies the values as boundaries

bin\_vals[1:values.size + 1] = values

if bin\_counts\_buffer is not None:

# slice out bins with +/-np.inf boundary -- their count will be zero anyway

# we can't just assume all other bins will have 1 value since there can be dups

short\_indices = np.arange(values.size, dtype=np.int32)

bin\_counts\_buffer.fill(0)

self.\_gather\_debug\_info(

values, short\_indices, bin\_vals[1:values.size + 1],

bin\_counts\_buffer[1:values.size + 1])

else:

# Gets the indices for the values that define the boundary for the bins

indices\_float = np.arange(0, weights.size, dtype=np.float32)

# Gets things in the correct shape for the linear interpolation

weights = weights.reshape(1, weights.size)

indices\_float = indices\_float.reshape(1, weights.size)

# wrap ndarrays into twml.Array

percentiles\_tarray = twml.Array(percentiles.reshape(percentiles.size, 1))

weights\_tarray = twml.Array(weights)

indices\_float\_tarray = twml.Array(indices\_float)

percentile\_indices\_tarray = twml.Array(percentile\_indices.reshape(percentiles.size, 1))

# Performs the binary search to find the indices corresponding to the percentiles

err = twml.CLIB.twml\_optim\_nearest\_interpolation(

percentile\_indices\_tarray.handle, percentiles\_tarray.handle, # output, input

weights\_tarray.handle, indices\_float\_tarray.handle # xs, ys

)

if err != 1000:

raise ValueError("""twml.CLIB.twml\_optim\_nearest\_interpolation

caught an error (see previous stdout). Error code: """ % err)

indices = indices[:bin\_vals.size]

indices[:] = percentile\_indices

indices[0] = 0

indices[-1] = weights.size - 1

# Gets the values at those indices and copies them into bin\_vals

values.take(indices, out=bin\_vals)

# get # of values per bucket

if bin\_counts\_buffer is not None:

self.\_gather\_debug\_info(values, indices, bin\_vals, bin\_counts\_buffer)

self.\_calibrated = True

class PercentileDiscretizerCalibrator(Calibrator):

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

Internally, each feature's values is accumulated via its own

``PercentileDiscretizerFeature`` object.

The steps for calibration are typically as follows:

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

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

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

'''

def \_\_init\_\_(self, n\_bin, out\_bits, bin\_histogram=True,

allow\_empty\_calibration=False, \*\*kwargs):

''' Constructs an PercentileDiscretizerCalibrator instance.

Arguments:

n\_bin:

the number of bins per feature to use for PercentileDiscretizer.

Note that each feature actually maps to n\_bin+1 output IDs.

out\_bits:

The maximum number of bits to use for the output IDs.

2\*\*out\_bits must be greater than bin\_ids.size or an error is raised.

bin\_histogram:

When True (the default), gathers information during calibration

to build a bin\_histogram.

allow\_empty\_calibration:

allows operation where we might not calibrate any features.

Default False to error out if no features were calibrated.

Typically, values of uncalibrated features pass through discretizers

untouched (though the feature ids will be truncated to obey out\_bits).

'''

super(PercentileDiscretizerCalibrator, self).\_\_init\_\_(\*\*kwargs)

self.\_n\_bin = n\_bin

self.\_out\_bits = out\_bits

self.\_bin\_ids = None

self.\_bin\_vals = np.empty(0, dtype=np.float32) # Note changed from 64 (v1) to 32 (v2)

self.\_bin\_histogram = bin\_histogram

self.\_bin\_histogram\_dict = None

self.\_hash\_map\_counter = 0

self.\_hash\_map = {}

self.\_discretizer\_feature\_dict = {}

self.\_allow\_empty\_calibration = allow\_empty\_calibration

@property

def bin\_ids(self):

'''

Gets bin\_ids

'''

return self.\_bin\_ids

@property

def bin\_vals(self):

'''

Gets bin\_vals

'''

return self.\_bin\_vals

@property

def hash\_map(self):

'''

Gets hash\_map

'''

return self.\_hash\_map

@property

def discretizer\_feature\_dict(self):

'''

Gets feature\_dict

'''

return self.\_discretizer\_feature\_dict

def accumulate\_features(self, inputs, name):

'''

Wrapper around accumulate for PercentileDiscretizer.

Arguments:

inputs:

batch that will be accumulated

name:

name of the tensor that will be accumulated

'''

sparse\_tf = inputs[name]

indices = sparse\_tf.indices[:, 1]

ids = sparse\_tf.indices[:, 0]

weights = np.take(inputs["weights"], ids)

return self.accumulate(indices, sparse\_tf.values, weights)

def accumulate\_feature(self, output):

'''

Wrapper around accumulate for trainer API.

Arguments:

output:

output of prediction of build\_graph for calibrator

'''

return self.accumulate(output['feature\_ids'], output['feature\_values'], output['weights'])

def accumulate(self, feature\_keys, feature\_vals, weights=None):

'''Accumulate a single batch of feature keys, values and weights.

These are accumulate until ``calibrate()`` is called.

Arguments:

feature\_keys:

1D int64 array of feature keys.

feature\_vals:

1D float array of feature values. Each element of this array

maps to the commensurate element in ``feature\_keys``.

weights:

Defaults to weights of 1.

1D array containing the weights of each feature key, value pair.

Typically, this is the weight of each sample (but you still need

to provide one weight per key,value pair).

Each element of this array maps to the commensurate element in feature\_keys.

'''

if feature\_keys.ndim != 1:

raise ValueError('Expecting 1D feature\_keys, got %dD' % feature\_keys.ndim)

if feature\_vals.ndim != 1:

raise ValueError('Expecting 1D feature\_values, got %dD' % feature\_vals.ndim)

if feature\_vals.size != feature\_keys.size:

raise ValueError(

'Expecting feature\_keys.size == feature\_values.size, got %d != %d' %

(feature\_keys.size, feature\_vals.size))

if weights is not None:

weights = np.squeeze(weights)

if weights.ndim != 1:

raise ValueError('Expecting 1D weights, got %dD' % weights.ndim)

elif weights.size != feature\_keys.size:

raise ValueError(

'Expecting feature\_keys.size == weights.size, got %d != %d' %

(feature\_keys.size, weights.size))

if weights is None:

weights = np.full(feature\_vals.size, fill\_value=DEFAULT\_SAMPLE\_WEIGHT)

unique\_keys = np.unique(feature\_keys)

for feature\_id in unique\_keys:

idx = np.where(feature\_keys == feature\_id)

if feature\_id not in self.\_discretizer\_feature\_dict:

self.\_hash\_map[feature\_id] = self.\_hash\_map\_counter

# unlike v1, the hash\_map\_counter is incremented AFTER assignment.

# This makes the hash\_map features zero-indexed: 0, 1, 2 instead of 1, 2, 3

self.\_hash\_map\_counter += 1

# creates a new cache if we never saw the feature before

discretizer\_feature = PercentileDiscretizerFeature(feature\_id)

self.\_discretizer\_feature\_dict[feature\_id] = discretizer\_feature

else:

discretizer\_feature = self.\_discretizer\_feature\_dict[feature\_id]

discretizer\_feature.add\_values({'values': feature\_vals[idx], 'weights': weights[idx]})

def calibrate(self, debug=False):

'''

Calibrates each PercentileDiscretizer feature after accumulation is complete.

Arguments:

debug:

Boolean to request debug info be returned by the method.

(see Returns section below)

The calibration results are stored in two matrices:

bin\_ids:

2D array of size number of accumulate ``features x n\_bin+1``.

Contains the new IDs generated by PercentileDiscretizer. Each row maps to a feature.

Each row maps to different value bins. The IDs

are in the range ``1 -> bin\_ids.size+1``

bin\_vals:

2D array of the same size as bin\_ids.

Each row maps to a feature. Each row contains the bin boundaries.

These boundaries represent feature values.

Returns:

if debug is True, the method returns

- 1D int64 array of feature\_ids

- 2D float32 array copy of bin\_vals (the bin boundaries) for each feature

- 2D int64 array of bin counts corresponding to the bin boundaries

'''

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

if n\_feature == 0 and not self.\_allow\_empty\_calibration:

raise RuntimeError("Need to accumulate some features for calibration\n"

"Likely, the calibration data is empty. This can\n"

"happen if the dataset is small, or if the following\n"

"cli args are set too low:\n"

" --discretizer\_keep\_rate (default=0.0008)\n"

" --discretizer\_parts\_downsampling\_rate (default=0.2)\n"

"Consider increasing the values of these args.\n"

"To allow empty calibration data (and degenerate discretizer),\n"

"use the allow\_empty\_calibration input of the constructor.")

self.\_bin\_ids = np.arange(1, n\_feature \* (self.\_n\_bin + 1) + 1)

self.\_bin\_ids = self.\_bin\_ids.reshape(n\_feature, self.\_n\_bin + 1)

self.\_bin\_vals.resize(n\_feature, self.\_n\_bin + 1)

# buffers shared by PercentileDiscretizerFeature.calibrate()

percentile\_indices = np.empty(self.\_n\_bin + 1, dtype=np.float32)

# Tensor from 0 to 1 in the number of steps provided

percentiles = np.linspace(0, 1, num=self.\_n\_bin + 1, dtype=np.float32)

if debug or self.\_bin\_histogram:

debug\_feature\_ids = np.empty(n\_feature, dtype=np.int64)

bin\_counts = np.empty((n\_feature, self.\_n\_bin + 1), dtype=np.int64)

# progress bar for calibration phase

progress\_bar = tf.keras.utils.Progbar(n\_feature)

discretizer\_features\_dict = self.\_discretizer\_feature\_dict

for i, feature\_id in enumerate(discretizer\_features\_dict):

if debug or self.\_bin\_histogram:

debug\_feature\_ids[self.\_hash\_map[feature\_id]] = feature\_id

bin\_counts\_buffer = bin\_counts[self.\_hash\_map[feature\_id]]

else:

bin\_counts\_buffer = None

# calibrate each PercentileDiscretizer feature (puts results in bin\_vals)

discretizer\_features\_dict[feature\_id].calibrate(

self.\_bin\_vals[self.\_hash\_map[feature\_id]], # Gets feature-values

percentiles, percentile\_indices,

bin\_counts\_buffer=bin\_counts\_buffer

)

# update progress bar 20 times

if (i % max(1.0, round(n\_feature / 20)) == 0) or (i == n\_feature - 1):

progress\_bar.update(i + 1)

super(PercentileDiscretizerCalibrator, self).calibrate()

if self.\_bin\_histogram:

# save bin histogram data for later

self.\_bin\_histogram\_dict = {

'feature\_ids': debug\_feature\_ids,

'bin\_counts': bin\_counts,

'bin\_vals': self.\_bin\_vals,

'out\_bits': self.\_out\_bits,

}

if debug:

return debug\_feature\_ids, self.\_bin\_vals.copy(), bin\_counts

return None

def \_create\_discretizer\_layer(self, n\_feature, hash\_map\_keys, hash\_map\_values,

feature\_offsets, name):

return twml.layers.PercentileDiscretizer(

n\_feature=n\_feature,

n\_bin=self.\_n\_bin,

out\_bits=self.\_out\_bits,

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

hash\_keys=hash\_map\_keys,

hash\_values=hash\_map\_values.astype(np.int64),

bin\_ids=self.\_bin\_ids.flatten().astype(np.int64),

feature\_offsets=feature\_offsets,

name=name,

\*\*self.\_kwargs

)

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 = self.\_create\_discretizer\_layer(n\_feature, hash\_map\_keys,

hash\_map\_values, feature\_offsets, name)

return discretizer

def get\_layer\_args(self):

'''

Returns layer arguments required to implement multi-phase training.

See twml.calibrator.Calibrator.get\_layer\_args for more detailed documentation.

'''

layer\_args = {

'n\_feature': len(self.\_discretizer\_feature\_dict),

'n\_bin': self.\_n\_bin,

'out\_bits': self.\_out\_bits,

}

return layer\_args

def add\_hub\_signatures(self, name):

"""

Add Hub Signatures for each calibrator

Arguments:

name:

Calibrator name

"""

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

calibrator\_layer = self.to\_layer()

hub.add\_signature(

inputs=sparse\_tf,

outputs=calibrator\_layer(sparse\_tf, keep\_inputs=False),

name=name)

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

"""

This method is called by save() to write a histogram of

PercentileDiscretizer feature bins to disk. A histogram is included for each

feature.

Arguments:

writer:

tf.summary.FilteWriter instance.

used to add summaries to event files for inclusion in tensorboard.

sess:

tf.Session instance. Used to produces summaries for the writer.

"""

bin\_counts\_ph = tf.placeholder(tf.int64)

bin\_counts = self.\_bin\_histogram\_dict['bin\_counts']

# Record that distribution into a histogram summary

histo = tf.summary.histogram("discretizer\_feature\_bin\_counts", bin\_counts\_ph)

for i in range(bin\_counts.shape[0]):

bin\_counts\_summary = sess.run(histo, feed\_dict={bin\_counts\_ph: bin\_counts[i]})

writer.add\_summary(bin\_counts\_summary, global\_step=i)

def write\_summary\_json(self, save\_dir, name="default"):

"""

Export bin information to HDFS.

Arguments:

save\_dir:

name of the saving directory.

name:

prefix of the saved hub signature. Default (string): "default".

"""

# Since the size is small: (# of bins) \* (# of features), we always dump the file.

discretizer\_export\_bin\_filename = os.path.join(save\_dir, name + '\_bin.json')

discretizer\_export\_bin\_dict = {

'feature\_ids': self.\_bin\_histogram\_dict['feature\_ids'].tolist(),

'bin\_boundaries': self.\_bin\_histogram\_dict['bin\_vals'].tolist(),

'output\_bits': self.\_bin\_histogram\_dict['out\_bits']

}

twml.write\_file(discretizer\_export\_bin\_filename, discretizer\_export\_bin\_dict, encode='json')

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

'''Save the calibrator into the given save\_directory using TF Hub.

Arguments:

save\_dir:

name of the saving directory.

name:

prefix of the saved hub signature. Default (string): "default".

'''

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()

# creates the signature to the calibrator module

hub.add\_signature(

inputs=inputs,

outputs=calibrator\_layer(inputs, keep\_inputs=False),

name=name)

# and another signature for keep\_inputs mode

hub.add\_signature(

inputs=inputs,

outputs=calibrator\_layer(inputs, keep\_inputs=True),

name=name + '\_keep\_inputs')

# 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)

self.write\_summary\_json(save\_dir, name)