# pylint: disable=arguments-differ, unused-argument

''' Contains Isotonic Calibration'''

from .calibrator import CalibrationFeature, Calibrator

from absl import logging

import numpy as np

from sklearn.isotonic import isotonic\_regression

import tensorflow.compat.v1 as tf

import tensorflow\_hub as hub

import twml

import twml.layers

DEFAULT\_SAMPLE\_WEIGHT = 1

def sort\_values(inputs, target, weight, ascending=True):

'''

Sorts arrays based on the first array.

Arguments:

inputs:

1D array which will dictate the order which the remainder 2 arrays will be sorted

target:

1D array

weight:

1D array

ascending:

Boolean. If set to True (the default), sorts values in ascending order.

Returns:

sorted inputs:

1D array sorted by the order of `ascending`

sorted targets:

1D array

sorted weight:

1D array

'''

# assert that the length of inputs and target are the same

if len(inputs) != len(target):

raise ValueError('Expecting inputs and target sizes to match')

# assert that the length of inputs and weight are the same

if len(inputs) != len(weight):

raise ValueError('Expecting inputs and weight sizes to match')

inds = inputs.argsort()

if not ascending:

inds = inds[::-1]

return inputs[inds], target[inds], weight[inds]

class IsotonicFeature(CalibrationFeature):

'''

IsotonicFeature adds values, weights and targets to each feature and then runs

isotonic regression by calling `sklearn.isotonic.isotonic\_regression

<http://scikit-learn.org/stable/auto\_examples/plot\_isotonic\_regression.html>`\_

'''

def \_get\_bin\_boundaries(self, n\_samples, bins, similar\_bins):

"""

Calculates the sample indices that define bin boundaries

Arguments:

n\_samples:

(int) number of samples

bins:

(int) number of bins. Needs to be smaller or equal than n\_samples.

similar\_bins:

(bool) If True, samples will be distributed in bins of equal size (up to one sample).

If False bins will be filled with step = N\_samples//bins, and last bin will contain all remaining samples.

Note that equal\_bins=False can create a last bins with a very large number of samples.

Returns:

(list[int]) List of sample indices defining bin boundaries

"""

if bins > n\_samples:

raise ValueError(

"The number of bins needs to be less than or equal to the number of samples. "

"Currently bins={0} and n\_samples={1}.".format(bins, n\_samples)

)

step = n\_samples // bins

if similar\_bins:

# dtype=int will floor the linspace

bin\_boundaries = np.linspace(0, n\_samples - step, num=bins, dtype=int)

else:

bin\_boundaries = range(0, step \* bins, step)

bin\_boundaries = np.append(bin\_boundaries, n\_samples)

return bin\_boundaries

def calibrate(self, bins, similar\_bins=False, debug=False):

'''Calibrates the IsotonicFeature into calibrated weights and bias.

1. Sorts the values of the feature class, based on the order of values

2. Performs isotonic regression using sklearn.isotonic.isotonic\_regression

3. Performs the binning of the samples, in order to obtain the final weight and bias

which will be used for inference

Note that this method can only be called once.

Arguments:

bins:

number of bins.

similar\_bins:

If True, samples will be distributed in bins of equal size (up to one sample).

If False bins will be filled with step = N\_samples//bins, and last bin will contain all remaining samples.

Note that equal\_bins=False can create a last bins with a very large number of samples.

debug:

Defaults to False. If debug is set to true, output other parameters useful for debugging.

Returns:

[calibrated weight, calibrated bias]

'''

if self.\_calibrated:

raise RuntimeError("Can only calibrate once")

# parse through the dict to obtain the targets, weights and values

self.\_concat\_arrays()

feature\_targets = self.\_features\_dict['targets']

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

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

srtd\_feature\_values, srtd\_feature\_targets, srtd\_feature\_weights = sort\_values(

inputs=feature\_values,

target=feature\_targets,

weight=feature\_weights

)

calibrated\_feature\_values = isotonic\_regression(

srtd\_feature\_targets, sample\_weight=srtd\_feature\_weights)

# create the final outputs for the prediction of each class

bpreds = []

btargets = []

bweights = []

rpreds = []

# Create bin boundaries

bin\_boundaries = self.\_get\_bin\_boundaries(

len(calibrated\_feature\_values), bins, similar\_bins=similar\_bins)

for sidx, eidx in zip(bin\_boundaries, bin\_boundaries[1:]):

# separate each one of the arrays based on their respective bins

lpreds = srtd\_feature\_values[int(sidx):int(eidx)]

lrpreds = calibrated\_feature\_values[int(sidx):int(eidx)]

ltargets = srtd\_feature\_targets[int(sidx):int(eidx)]

lweights = srtd\_feature\_weights[int(sidx):int(eidx)]

# calculate the outputs (including the bpreds and rpreds)

bpreds.append(np.sum(lpreds \* lweights) / (np.squeeze(np.sum(lweights))))

rpreds.append(np.sum(lrpreds \* lweights) / (np.squeeze(np.sum(lweights))))

btargets.append(np.sum(ltargets \* lweights) / (np.squeeze(np.sum(lweights))))

bweights.append(np.squeeze(np.sum(lweights)))

# transposing the bpreds and rpreds which will be used as input to the inference step

bpreds = np.asarray(bpreds).T

rpreds = np.asarray(rpreds).T

btargets = np.asarray(btargets).T

bweights = np.asarray(bweights).T

# setting \_calibrated to be True which is necessary in order to prevent it to re-calibrate

self.\_calibrated = True

if debug:

return bpreds, rpreds, btargets, bweights

return bpreds, rpreds

class IsotonicCalibrator(Calibrator):

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

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

The steps for calibration are typically as follows:

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

2. calibrate all feature into Isotonic ``bpreds``, ``rpreds`` by calling ``calibrate()``; and

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

'''

def \_\_init\_\_(self, n\_bin, similar\_bins=False, \*\*kwargs):

''' Constructs an isotonicCalibrator instance.

Arguments:

n\_bin:

the number of bins per feature to use for isotonic.

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

'''

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

self.\_n\_bin = n\_bin

self.\_similar\_bins = similar\_bins

self.\_ys\_input = []

self.\_xs\_input = []

self.\_isotonic\_feature\_dict = {}

def accumulate\_feature(self, output):

'''

Wrapper around accumulate for trainer API.

Arguments:

output: output of prediction of build\_graph for calibrator

'''

weights = output['weights'] if 'weights' in output else None

return self.accumulate(output['predictions'], output['targets'], weights)

def accumulate(self, predictions, targets, weights=None):

'''

Accumulate a single batch of class predictions, class targets and class weights.

These are accumulated until calibrate() is called.

Arguments:

predictions:

float matrix of class values. Each dimension corresponds to a different class.

Shape is ``[n, d]``, where d is the number of classes.

targets:

float matrix of class targets. Each dimension corresponds to a different class.

Shape ``[n, d]``, where d is the number of classes.

weights:

Defaults to weights of 1.

1D array containing the weights of each prediction.

'''

if predictions.shape != targets.shape:

raise ValueError(

'Expecting predictions.shape == targets.shape, got %s and %s instead' %

(str(predictions.shape), str(targets.shape)))

if weights is not None:

if weights.ndim != 1:

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

elif weights.size != predictions.shape[0]:

raise ValueError(

'Expecting predictions.shape[0] == weights.size, got %d != %d instead' %

(predictions.shape[0], weights.size))

# iterate through the rows of predictions and sets one class to each row

if weights is None:

weights = np.full(predictions.shape[0], fill\_value=DEFAULT\_SAMPLE\_WEIGHT)

for class\_key in range(predictions.shape[1]):

# gets the predictions and targets for that class

class\_predictions = predictions[:, class\_key]

class\_targets = targets[:, class\_key]

if class\_key not in self.\_isotonic\_feature\_dict:

isotonic\_feature = IsotonicFeature(class\_key)

self.\_isotonic\_feature\_dict[class\_key] = isotonic\_feature

else:

isotonic\_feature = self.\_isotonic\_feature\_dict[class\_key]

isotonic\_feature.add\_values({'values': class\_predictions, 'weights': weights,

'targets': class\_targets})

def calibrate(self, debug=False):

'''

Calibrates each IsotonicFeature after accumulation is complete.

Results are stored in ``self.\_ys\_input`` and ``self.\_xs\_input``

Arguments:

debug:

Defaults to False. If set to true, returns the ``xs\_input`` and ``ys\_input``.

'''

super(IsotonicCalibrator, self).calibrate()

bias\_temp = []

weight\_temp = []

logging.info("Beginning isotonic calibration.")

isotonic\_features\_dict = self.\_isotonic\_feature\_dict

for class\_id in isotonic\_features\_dict:

bpreds, rpreds = isotonic\_features\_dict[class\_id].calibrate(bins=self.\_n\_bin, similar\_bins=self.\_similar\_bins)

weight\_temp.append(bpreds)

bias\_temp.append(rpreds)

# save isotonic results onto a matrix

self.\_xs\_input = np.array(weight\_temp, dtype=np.float32)

self.\_ys\_input = np.array(bias\_temp, dtype=np.float32)

logging.info("Isotonic calibration finished.")

if debug:

return np.array(weight\_temp), np.array(bias\_temp)

return None

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".

'''

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)

logging.info("You probably do not need to save the isotonic layer. \

So feel free to set save to False in the Trainer. \

Additionally this only saves the layer not the whole graph.")

def calibrator\_module():

'''

Way to save Isotonic layer

'''

# The input to isotonic is a dense layer

inputs = tf.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 to\_layer(self):

""" Returns a twml.layers.Isotonic Layer that can be used for feature discretization.

"""

if not self.\_calibrated:

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

isotonic\_layer = twml.layers.Isotonic(

n\_unit=self.\_xs\_input.shape[0], n\_bin=self.\_xs\_input.shape[1],

xs\_input=self.\_xs\_input, ys\_input=self.\_ys\_input,

\*\*self.\_kwargs)

return isotonic\_layer

def get\_layer\_args(self, name=None):

""" Returns layer args. See ``Calibrator.get\_layer\_args`` for more detailed documentation """

return {'n\_unit': self.\_xs\_input.shape[0], 'n\_bin': self.\_xs\_input.shape[1]}