"""

This module contains custom tensorflow metrics used at Twitter.

Its components conform to conventions used by the ``tf.metrics`` module.

"""

from collections import OrderedDict

from functools import partial

import numpy as np

import tensorboard as tb

import tensorflow.compat.v1 as tf

CLAMP\_EPSILON = 0.00001

def total\_weight\_metric(

labels,

predictions,

weights=None,

metrics\_collections=None,

updates\_collections=None,

name=None):

with tf.variable\_scope(name, 'total\_weight', (labels, predictions, weights)):

total\_weight = \_metric\_variable(name='total\_weight', shape=[], dtype=tf.float64)

if weights is None:

weights = tf.cast(tf.size(labels), total\_weight.dtype, name="default\_weight")

else:

weights = tf.cast(weights, total\_weight.dtype)

# add up the weights to get total weight of the eval set

update\_total\_weight = tf.assign\_add(total\_weight, tf.reduce\_sum(weights), name="update\_op")

value\_op = tf.identity(total\_weight)

update\_op = tf.identity(update\_total\_weight)

if metrics\_collections:

tf.add\_to\_collections(metrics\_collections, value\_op)

if updates\_collections:

tf.add\_to\_collections(updates\_collections, update\_op)

return value\_op, update\_op

def num\_samples\_metric(

labels,

predictions,

weights=None,

metrics\_collections=None,

updates\_collections=None,

name=None):

with tf.variable\_scope(name, 'num\_samples', (labels, predictions, weights)):

num\_samples = \_metric\_variable(name='num\_samples', shape=[], dtype=tf.float64)

update\_num\_samples = tf.assign\_add(num\_samples, tf.cast(tf.size(labels), num\_samples.dtype), name="update\_op")

value\_op = tf.identity(num\_samples)

update\_op = tf.identity(update\_num\_samples)

if metrics\_collections:

tf.add\_to\_collections(metrics\_collections, value\_op)

if updates\_collections:

tf.add\_to\_collections(updates\_collections, update\_op)

return value\_op, update\_op

def ctr(labels, predictions,

weights=None,

metrics\_collections=None,

updates\_collections=None,

name=None):

# pylint: disable=unused-argument

"""

Compute the weighted average positive sample ratio based on labels

(i.e. weighted average percentage of positive labels).

The name `ctr` (click-through-rate) is from legacy.

Args:

labels: the ground truth value.

predictions: the predicted values, whose shape must match labels. Ignored for CTR computation.

weights: optional weights, whose shape must match labels . Weight is 1 if not set.

metrics\_collections: optional list of collections to add this metric into.

updates\_collections: optional list of collections to add the associated update\_op into.

name: an optional variable\_scope name.

Return:

ctr: A `Tensor` representing positive sample ratio.

update\_op: A update operation used to accumulate data into this metric.

"""

return tf.metrics.mean(

values=labels,

weights=weights,

metrics\_collections=metrics\_collections,

updates\_collections=updates\_collections,

name=name)

def predicted\_ctr(labels, predictions,

weights=None,

metrics\_collections=None,

updates\_collections=None,

name=None):

# pylint: disable=unused-argument

"""

Compute the weighted average positive ratio based on predictions,

(i.e. weighted averaged predicted positive probability).

The name `ctr` (click-through-rate) is from legacy.

Args:

labels: the ground truth value.

predictions: the predicted values, whose shape must match labels. Ignored for CTR computation.

weights: optional weights, whose shape must match labels . Weight is 1 if not set.

metrics\_collections: optional list of collections to add this metric into.

updates\_collections: optional list of collections to add the associated update\_op into.

name: an optional variable\_scope name.

Return:

predicted\_ctr: A `Tensor` representing the predicted positive ratio.

update\_op: A update operation used to accumulate data into this metric.

"""

return tf.metrics.mean(

values=predictions,

weights=weights,

metrics\_collections=metrics\_collections,

updates\_collections=updates\_collections,

name=name)

def prediction\_std\_dev(labels, predictions,

weights=None,

metrics\_collections=None,

updates\_collections=None,

name=None):

"""

Compute the weighted standard deviation of the predictions.

Note - this is not a confidence interval metric.

Args:

labels: the ground truth value.

predictions: the predicted values, whose shape must match labels. Ignored for CTR computation.

weights: optional weights, whose shape must match labels . Weight is 1 if not set.

metrics\_collections: optional list of collections to add this metric into.

updates\_collections: optional list of collections to add the associated update\_op into.

name: an optional variable\_scope name.

Return:

metric value: A `Tensor` representing the value of the metric on the data accumulated so far.

update\_op: A update operation used to accumulate data into this metric.

"""

with tf.variable\_scope(name, 'pred\_std\_dev', (labels, predictions, weights)):

labels = tf.cast(labels, tf.float64)

predictions = tf.cast(predictions, tf.float64)

if weights is None:

weights = tf.ones(shape=tf.shape(labels), dtype=tf.float64, name="default\_weight")

else:

weights = tf.cast(weights, tf.float64)

# State kept during streaming of examples

total\_weighted\_preds = \_metric\_variable(

name='total\_weighted\_preds', shape=[], dtype=tf.float64)

total\_weighted\_preds\_sq = \_metric\_variable(

name='total\_weighted\_preds\_sq', shape=[], dtype=tf.float64)

total\_weights = \_metric\_variable(

name='total\_weights', shape=[], dtype=tf.float64)

# Update state

update\_total\_weighted\_preds = tf.assign\_add(total\_weighted\_preds, tf.reduce\_sum(weights \* predictions))

update\_total\_weighted\_preds\_sq = tf.assign\_add(total\_weighted\_preds\_sq, tf.reduce\_sum(weights \* predictions \* predictions))

update\_total\_weights = tf.assign\_add(total\_weights, tf.reduce\_sum(weights))

# Compute output

def compute\_output(tot\_w, tot\_wp, tot\_wpp):

return tf.math.sqrt(tot\_wpp / tot\_w - (tot\_wp / tot\_w) \*\* 2)

std\_dev\_est = compute\_output(total\_weights, total\_weighted\_preds, total\_weighted\_preds\_sq)

update\_std\_dev\_est = compute\_output(update\_total\_weights, update\_total\_weighted\_preds, update\_total\_weighted\_preds\_sq)

if metrics\_collections:

tf.add\_to\_collections(metrics\_collections, std\_dev\_est)

if updates\_collections:

tf.add\_to\_collections(updates\_collections, update\_std\_dev\_est)

return std\_dev\_est, update\_std\_dev\_est

def \_get\_arce\_predictions(predictions, weights, label\_weighted, labels,

up\_weight, deprecated\_rce,

total\_positive, update\_total\_positive):

"""

Returns the ARCE predictions, total\_positive, update\_total\_positive and weights

used by the rest of the twml.metrics.rce metric computation.

"""

predictions\_weighted = tf.multiply(predictions, weights, name="weighted\_preds")

label\_weighted\_comp = tf.subtract(tf.reduce\_sum(weights), tf.reduce\_sum(label\_weighted))

pred\_weight\_comp = tf.subtract(tf.reduce\_sum(weights), tf.reduce\_sum(predictions\_weighted))

normalizer\_comp = label\_weighted\_comp / pred\_weight\_comp

if up\_weight is False:

total\_positive\_unweighted = \_metric\_variable(

name='total\_positive\_unweighted', shape=[], dtype=tf.float32)

update\_total\_positive\_unweighted = tf.assign\_add(

total\_positive\_unweighted, tf.reduce\_sum(labels),

name="total\_positive\_unweighted\_update")

if deprecated\_rce:

normalizer = tf.reduce\_sum(labels) / tf.reduce\_sum(label\_weighted)

else:

# sum of labels / sum of weighted labels

normalizer = update\_total\_positive\_unweighted / update\_total\_positive

label\_comp = tf.subtract(tf.to\_float(tf.size(labels)), tf.reduce\_sum(labels))

normalizer\_comp = label\_comp / label\_weighted\_comp

# note that up\_weight=True changes these for the rest of the twml.metric.rce computation

weights = tf.ones(shape=tf.shape(labels), dtype=tf.float32, name="default\_weight")

total\_positive = total\_positive\_unweighted

update\_total\_positive = update\_total\_positive\_unweighted

else:

if deprecated\_rce:

normalizer = tf.reduce\_sum(label\_weighted) / tf.reduce\_sum(predictions\_weighted)

else:

# normalizer used for NRCE (and ARCE with up\_weight=True)

total\_prediction = \_metric\_variable(name='total\_prediction', shape=[], dtype=tf.float32)

# update the variable holding the sum of weighted predictions

update\_total\_prediction = tf.assign\_add(

total\_prediction, tf.reduce\_sum(predictions\_weighted), name="total\_prediction\_update")

# this used to be tf.reduce\_sum(label\_weighted) / tf.reduce\_sum(predictions\_weighted)

# but it measure normalizer over batch was too flawed an approximation.

normalizer = update\_total\_positive / update\_total\_prediction

pred\_comp = tf.subtract(tf.ones(shape=tf.shape(labels), dtype=tf.float32), predictions)

pred\_comp\_norm = tf.multiply(pred\_comp, normalizer\_comp, name="normalized\_predictions\_comp")

pred\_num = tf.multiply(predictions, normalizer, name="normalized\_pred\_numerator")

pred\_denom = tf.add(pred\_num, pred\_comp\_norm, name="normalized\_pred\_denominator")

predictions = pred\_num / pred\_denom

return predictions, total\_positive, update\_total\_positive, weights

def rce(labels, predictions,

weights=None,

normalize=False,

arce=False,

up\_weight=True,

metrics\_collections=None,

updates\_collections=None,

name=None,

deprecated\_rce=False):

"""

Compute the relative cross entropy (RCE).

The RCE is a relative measurement compared to the baseline model's performance.

The baseline model always predicts average click-through-rate (CTR).

The RCE measures, in percentage, how much better the predictions are, compared

to the baseline model, in terms of cross entropy loss.

y = label; p = prediction;

binary cross entropy = y \* log(p) + (1-y) \* log(1-p)

Args:

labels:

the ground true value.

predictions:

the predicted values, whose shape must match labels.

weights:

optional weights, whose shape must match labels . Weight is 1 if not set.

normalize:

if set to true, produce NRCEs used at Twitter. (normalize preds by weights first)

NOTE: if you don't understand what NRCE is, please don't use it.

arce:

if set to true, produces `ARCE <http://go/arce>`\_.

This can only be activated if `normalize=True`.

up\_weight:

if set to true, produces arce in the up\_weighted space (considers CTR after up\_weighting

data), while False gives arce in the original space (only considers CTR before up\_weighting).

In the actual version, this flag can only be activated if arce is True.

Notice that the actual version of NRCE corresponds to up\_weight=True.

metrics\_collections:

optional list of collections to add this metric into.

updates\_collections:

optional list of collections to add the associated update\_op into.

name:

an optional variable\_scope name.

deprecated\_rce:

enables the previous NRCE/ARCE calculations which calculated some label metrics

on the batch instead of on all batches seen so far. Note that the older metric

calculation is less stable, especially for smaller batch sizes. You should probably

never have to set this to True.

Return:

rce\_value:

A ``Tensor`` representing the RCE.

update\_op:

A update operation used to accumulate data into this metric.

.. note:: Must have at least 1 positive and 1 negative sample accumulated,

or RCE will come out as NaN.

"""

with tf.variable\_scope(name, 'rce', (labels, predictions, weights)):

labels = tf.to\_float(labels, name="label\_to\_float")

predictions = tf.to\_float(predictions, name="predictions\_to\_float")

if weights is None:

weights = tf.ones(shape=tf.shape(labels), dtype=tf.float32, name="default\_weight")

else:

weights = tf.to\_float(weights, name="weight\_to\_float")

total\_positive = \_metric\_variable(name='total\_positive', shape=[], dtype=tf.float32)

total\_loss = \_metric\_variable(name='total\_loss', shape=[], dtype=tf.float32)

total\_weight = \_metric\_variable(name='total\_weight', shape=[], dtype=tf.float32)

label\_weighted = tf.multiply(labels, weights, name="weighted\_label")

update\_total\_positive = tf.assign\_add(

total\_positive, tf.reduce\_sum(label\_weighted), name="total\_pos\_update")

if arce:

if normalize is False:

raise ValueError('This configuration of parameters is not actually allowed')

predictions, total\_positive, update\_total\_positive, weights = \_get\_arce\_predictions(

predictions=predictions, weights=weights, deprecated\_rce=deprecated\_rce,

label\_weighted=label\_weighted, labels=labels, up\_weight=up\_weight,

total\_positive=total\_positive, update\_total\_positive=update\_total\_positive)

elif normalize:

predictions\_weighted = tf.multiply(predictions, weights, name="weighted\_preds")

if deprecated\_rce:

normalizer = tf.reduce\_sum(label\_weighted) / tf.reduce\_sum(predictions\_weighted)

else:

total\_prediction = \_metric\_variable(name='total\_prediction', shape=[], dtype=tf.float32)

# update the variable holding the sum of weighted predictions

update\_total\_prediction = tf.assign\_add(

total\_prediction, tf.reduce\_sum(predictions\_weighted), name="total\_prediction\_update")

# this used to be tf.reduce\_sum(label\_weighted) / tf.reduce\_sum(predictions\_weighted)

# but it measure normalizer over batch was too flawed an approximation.

normalizer = update\_total\_positive / update\_total\_prediction

# NRCE

predictions = tf.multiply(predictions, normalizer, name="normalized\_predictions")

# clamp predictions to keep log(p) stable

clip\_p = tf.clip\_by\_value(predictions, CLAMP\_EPSILON, 1.0 - CLAMP\_EPSILON, name="clip\_p")

logloss = \_binary\_cross\_entropy(pred=clip\_p, target=labels, name="logloss")

logloss\_weighted = tf.multiply(logloss, weights, name="weighted\_logloss")

update\_total\_loss = tf.assign\_add(

total\_loss, tf.reduce\_sum(logloss\_weighted), name="total\_loss\_update")

update\_total\_weight = tf.assign\_add(

total\_weight, tf.reduce\_sum(weights), name="total\_weight\_update")

# metric value retrieval subgraph

ctr1 = tf.truediv(total\_positive, total\_weight, name="ctr")

# Note: we don't have to keep running averages for computing baseline CE. Because the prediction

# is constant for every sample, we can simplify it to the formula below.

baseline\_ce = \_binary\_cross\_entropy(pred=ctr1, target=ctr1, name="baseline\_ce")

pred\_ce = tf.truediv(total\_loss, total\_weight, name="pred\_ce")

rce\_t = tf.multiply(

1.0 - tf.truediv(pred\_ce, baseline\_ce),

100,

name="rce")

# metric update subgraph

ctr2 = tf.truediv(update\_total\_positive, update\_total\_weight, name="ctr\_update")

# Note: we don't have to keep running averages for computing baseline CE. Because the prediction

# is constant for every sample, we can simplify it to the formula below.

baseline\_ce2 = \_binary\_cross\_entropy(pred=ctr2, target=ctr2, name="baseline\_ce\_update")

pred\_ce2 = tf.truediv(update\_total\_loss, update\_total\_weight, name="pred\_ce\_update")

update\_op = tf.multiply(

1.0 - tf.truediv(pred\_ce2, baseline\_ce2),

100,

name="update\_op")

if metrics\_collections:

tf.add\_to\_collections(metrics\_collections, rce\_t)

if updates\_collections:

tf.add\_to\_collections(updates\_collections, update\_op)

return rce\_t, update\_op

def ce(p\_true, p\_est=None):

if p\_est is None:

p\_est = p\_true

return \_binary\_cross\_entropy(pred=p\_est, target=p\_true, name=None)

def rce\_transform(outputs, labels, weights):

'''

Construct an OrderedDict of quantities to aggregate over eval batches

outputs, labels, weights are TensorFlow tensors, and are assumed to

be of shape [N] for batch\_size = N

Each entry in the output OrderedDict should also be of shape [N]

'''

out\_vals = OrderedDict()

out\_vals['weighted\_loss'] = weights \* ce(p\_true=labels, p\_est=outputs)

out\_vals['weighted\_labels'] = labels \* weights

out\_vals['weight'] = weights

return out\_vals

def rce\_metric(aggregates):

'''

input ``aggregates`` is an OrderedDict with the same keys as those created

by rce\_transform(). The dict values are the aggregates (reduce\_sum)

of the values produced by rce\_transform(), and should be scalars.

output is the value of RCE

'''

# cummulative weighted loss of model predictions

total\_weighted\_loss = aggregates['weighted\_loss']

total\_weighted\_labels = aggregates['weighted\_labels']

total\_weight = aggregates['weight']

model\_average\_loss = total\_weighted\_loss / total\_weight

baseline\_average\_loss = ce(total\_weighted\_labels / total\_weight)

return 100.0 \* (1 - model\_average\_loss / baseline\_average\_loss)

def metric\_std\_err(labels, predictions,

weights=None,

transform=rce\_transform, metric=rce\_metric,

metrics\_collections=None,

updates\_collections=None,

name='rce\_std\_err'):

"""

Compute the weighted standard error of the RCE metric on this eval set.

This can be used for confidence intervals and unpaired hypothesis tests.

Args:

labels: the ground truth value.

predictions: the predicted values, whose shape must match labels.

weights: optional weights, whose shape must match labels . Weight is 1 if not set.

transform: a function of the following form:

.. code-block:: python

def transform(outputs, labels, weights):

out\_vals = OrderedDict()

...

return out\_vals

where outputs, labels, and weights are all tensors of shape [eval\_batch\_size].

The returned OrderedDict() should have values that are tensors of shape [eval\_batch\_size].

These will be aggregated across many batches in the eval dataset, to produce

one scalar value per key of out\_vals.

metric: a function of the following form

.. code-block:: python

def metric(aggregates):

...

return metric\_value

where aggregates is an OrderedDict() having the same keys created by transform().

Each of the corresponding dict values is the reduce\_sum of the values produced by

transform(), and is a TF scalar. The return value should be a scalar representing

the value of the desired metric.

metrics\_collections: optional list of collections to add this metric into.

updates\_collections: optional list of collections to add the associated update\_op into.

name: an optional variable\_scope name.

Return:

metric value: A `Tensor` representing the value of the metric on the data accumulated so far.

update\_op: A update operation used to accumulate data into this metric.

"""

with tf.variable\_scope(name, 'metric\_std\_err', (labels, predictions, weights)):

labels = tf.cast(labels, tf.float64)

predictions = tf.cast(predictions, tf.float64)

if weights is None:

weights = tf.ones\_like(labels, dtype=tf.float64, name="default\_weight")

else:

weights = tf.cast(weights, tf.float64)

labels = tf.reshape(labels, [-1])

predictions = tf.reshape(predictions, [-1])

predictions = tf.clip\_by\_value(predictions, CLAMP\_EPSILON, 1.0 - CLAMP\_EPSILON, name="clip\_p")

weights = tf.reshape(weights, [-1])

# first apply the supplied transform function to the output, label, weight data

# returns an OrderedDict of 1xN tensors for N input samples

# for each sample, compute f = transform(pred, l, w)

transformed = transform(predictions, labels, weights)

# we track 3 types of aggregate information

# 1. total number of samples

# 2. aggregated transformed samples (moment1), i.e. sum(f)

# 3. aggregated crosses of transformed samples (moment2), i.e. sum(f\*f^T)

# count total number of samples

sample\_count = \_metric\_variable(

name='sample\_count', shape=[], dtype=tf.int64)

update\_sample\_count = tf.assign\_add(sample\_count, tf.size(labels, out\_type=sample\_count.dtype))

# compose the ordered dict into a single vector

# so f can be treated as a single column vector rather than a collection of scalars

N = len(transformed)

transformed\_vec = tf.stack(list(transformed.values()), axis=1)

# compute and update transformed samples (1st order statistics)

# i.e. accumulate f into F as F += sum(f)

aggregates\_1 = \_metric\_variable(

name='aggregates\_1', shape=[N], dtype=tf.float64)

update\_aggregates\_1 = tf.assign\_add(aggregates\_1, tf.reduce\_sum(transformed\_vec, axis=0))

# compute and update crossed transformed samples (2nd order statistics)

# i.e. accumulate f\*f^T into F2 as F2 += sum(f\*transpose(f))

aggregates\_2 = \_metric\_variable(

name='aggregates\_2', shape=[N, N], dtype=tf.float64)

moment\_2\_temp = (

tf.reshape(transformed\_vec, shape=[-1, N, 1])

\* tf.reshape(transformed\_vec, shape=[-1, 1, N])

)

update\_aggregates\_2 = tf.assign\_add(aggregates\_2, tf.reduce\_sum(moment\_2\_temp, axis=0))

def compute\_output(agg\_1, agg\_2, samp\_cnt):

# decompose the aggregates back into a dict to pass to the user-supplied metric fn

aggregates\_dict = OrderedDict()

for i, key in enumerate(transformed.keys()):

aggregates\_dict[key] = agg\_1[i]

metric\_value = metric(aggregates\_dict)

# derivative of metric with respect to the 1st order aggregates

# i.e. d M(agg1) / d agg1

metric\_prime = tf.gradients(metric\_value, agg\_1, stop\_gradients=agg\_1)

# estimated covariance of agg\_1

# cov(F) = sum(f\*f^T) - (sum(f) \* sum(f)^T) / N

# = agg\_2 - (agg\_1 \* agg\_1^T) / N

N\_covariance\_estimate = agg\_2 - (

tf.reshape(agg\_1, shape=[-1, 1])

@ tf.reshape(agg\_1, shape=[1, -1])

/ tf.cast(samp\_cnt, dtype=tf.float64)

)

# push N\_covariance\_estimate through a linearization of metric around agg\_1

# metric var = transpose(d M(agg1) / d agg1) \* cov(F) \* (d M(agg1) / d agg1)

metric\_variance = (

tf.reshape(metric\_prime, shape=[1, -1])

@ N\_covariance\_estimate

@ tf.reshape(metric\_prime, shape=[-1, 1])

)

# result should be a single element, but the matmul is 2D

metric\_variance = metric\_variance[0][0]

metric\_stderr = tf.sqrt(metric\_variance)

return metric\_stderr

metric\_stderr = compute\_output(aggregates\_1, aggregates\_2, sample\_count)

update\_metric\_stderr = compute\_output(update\_aggregates\_1, update\_aggregates\_2, update\_sample\_count)

if metrics\_collections:

tf.add\_to\_collections(metrics\_collections, metric\_stderr)

if updates\_collections:

tf.add\_to\_collections(updates\_collections, update\_metric\_stderr)

return metric\_stderr, update\_metric\_stderr

def lolly\_nrce(labels, predictions,

weights=None,

metrics\_collections=None,

updates\_collections=None,

name=None):

"""

Compute the Lolly NRCE.

Note: As this NRCE calculation uses Taylor expansion, it becomes inaccurate when the ctr is large,

especially when the adjusted ctr goes above 1.0.

Calculation:

::

NRCE: lolly NRCE

BCE: baseline cross entropy

NCE: normalized cross entropy

CE: cross entropy

y\_i: label of example i

p\_i: prediction of example i

y: ctr

p: average prediction

a: normalizer

Assumes any p\_i and a \* p\_i is within [0, 1)

NRCE = (1 - NCE / BCE) \* 100

BCE = - sum\_i(y\_i \* log(y) + (1 - y\_i) \* log(1 - y))

= - (y \* log(y) + (1 - y) \* log(1 - y))

a = y / p

CE = - sum\_i(y\_i \* log(p\_i) + (1 - y\_i) \* log(1 - p\_i))

NCE = - sum\_i(y\_i \* log(a \* p\_i) + (1 - y\_i) \* log(1 - a \* p\_i))

= - sum\_i(y\_i \* log(p\_i) + (1 - y\_i) \* log(1 - p\_i))

- sum\_i(y\_i \* log(a))

+ sum\_i((1 - y\_i) \* log(1 - p\_i))

- sum\_i((1 - y\_i) \* log(1 - a \* p\_i))

~= CE - sum\_i(y\_i) \* log(a)

+ sum\_i((1 - y\_i) \* (- sum\_{j=1~5}(p\_i^j / j)))

- sum\_i((1 - y\_i) \* (- sum\_{j=1~5}(a^j \* p\_i^j / j)))

# Takes 5 items from the Taylor expansion, can be increased if needed

# Error for each example is O(p\_i^6)

= CE - sum\_i(y\_i) \* log(a)

- sum\_{j=1~5}(sum\_i((1 - y\_i) \* p\_i^j) / j)

+ sum\_{j=1~5}(sum\_i((1 - y\_i) \* p\_i^j) \* a^j / j)

= CE - sum\_i(y\_i) \* log(a)

+ sum\_{j=1~5}(sum\_i((1 - y\_i) \* p\_i^j) \* (a^j - 1) / j)

Thus we keep track of CE, sum\_i(y\_i), sum\_i((1 - y\_i) \* p\_i^j) for j=1~5.

We also keep track of p and y by sum\_i(y\_i), sum\_i(p\_i), sum\_i(1) so that

we can get a at the end, which leads to this NRCE.

NRCE uses ctr and average pctr to normalize the pctrs.

It removes the impact of prediction error from RCE.

Usually NRCE is higher as the prediction error impact on RCE is negative.

Removing prediction error in our model can make RCE closer to NRCE and thus improve RCE.

In Lolly NRCE we use ctr and average pctr of the whole dataset.

We thus remove the dataset level error in NRCE calculation.

In this case, when we want to improve RCE to the level of NRCE,

it is achievable as dataset level prediction error is easy to remove by calibration.

Lolly NRCE is thus a good estimate about the potential gain by adding calibration.

In DBv2 NRCE, we use per-batch ctr and average pctr. We remove the batch level error.

This error is difficult to remove by modeling improvement,

at least not by simple calibration.

It thus cannot indicate the same opportunity as the Lolly NRCE does.

Args:

labels:

the ground true value.

predictions:

the predicted values, whose shape must match labels.

weights:

optional weights, whose shape must match labels . Weight is 1 if not set.

metrics\_collections:

optional list of collections to add this metric into.

updates\_collections:

optional list of collections to add the associated update\_op into.

name:

an optional variable\_scope name.

Return:

rce\_value:

A ``Tensor`` representing the RCE.

update\_op:

A update operation used to accumulate data into this metric.

Note: Must have at least 1 positive and 1 negative sample accumulated,

or NRCE will come out as NaN.

"""

with tf.variable\_scope(name, "lolly\_nrce", (labels, predictions, weights)):

labels = tf.to\_float(labels, name="label\_to\_float")

predictions = tf.to\_float(predictions, name="predictions\_to\_float")

if weights is None:

weights = tf.ones(shape=tf.shape(labels), dtype=tf.float32, name="default\_weight")

else:

weights = tf.to\_float(weights, name="weight\_to\_float")

positive\_weights = tf.multiply(labels, weights, name="positive\_weights")

# clamp predictions to keep log(p) stable

clip\_predictions = tf.clip\_by\_value(

predictions,

CLAMP\_EPSILON,

1.0 - CLAMP\_EPSILON,

name="clip\_predictions")

weighted\_predictions = tf.multiply(

predictions, weights,

name="weighted\_predictions")

logloss = \_binary\_cross\_entropy(pred=clip\_predictions, target=labels, name="logloss")

weighted\_logloss = tf.multiply(logloss, weights, name="weighted\_logloss")

negatives = tf.subtract(

tf.ones(shape=tf.shape(labels), dtype=tf.float32),

labels,

name="negatives")

negative\_predictions = tf.multiply(

predictions,

negatives,

name="negative\_predictions")

weighted\_negative\_predictions = tf.multiply(

negative\_predictions, weights,

name="weighted\_negative\_predictions")

negative\_squared\_predictions = tf.multiply(

negative\_predictions,

negative\_predictions,

name="negative\_squared\_predictions")

weighted\_negative\_squared\_predictions = tf.multiply(

negative\_squared\_predictions, weights,

name="weighted\_negative\_squared\_predictions")

negative\_cubed\_predictions = tf.multiply(

negative\_squared\_predictions,

negative\_predictions,

name="negative\_cubed\_predictions")

weighted\_negative\_cubed\_predictions = tf.multiply(

negative\_cubed\_predictions, weights,

name="weighted\_negative\_cubed\_predictions")

negative\_quartic\_predictions = tf.multiply(

negative\_cubed\_predictions,

negative\_predictions,

name="negative\_quartic\_predictions")

weighted\_negative\_quartic\_predictions = tf.multiply(

negative\_quartic\_predictions, weights,

name="weighted\_negative\_quartic\_predictions")

negative\_quintic\_predictions = tf.multiply(

negative\_quartic\_predictions,

negative\_predictions,

name="negative\_quintic\_predictions")

weighted\_negative\_quintic\_predictions = tf.multiply(

negative\_quintic\_predictions, weights,

name="weighted\_negative\_quintic\_predictions")

# Tracked stats

total\_positive = \_metric\_variable(name="total\_positive", shape=[], dtype=tf.float32)

total\_weight = \_metric\_variable(name="total\_weight", shape=[], dtype=tf.float32)

total\_prediction = \_metric\_variable(name="total\_prediction", shape=[], dtype=tf.float32)

total\_negative\_prediction = \_metric\_variable(

name="total\_negative\_prediction",

shape=[], dtype=tf.float32)

total\_negative\_squared\_prediction = \_metric\_variable(

name="total\_negative\_squared\_prediction",

shape=[], dtype=tf.float32)

total\_negative\_cubed\_prediction = \_metric\_variable(

name="total\_negative\_cubed\_prediction",

shape=[], dtype=tf.float32)

total\_negative\_quartic\_prediction = \_metric\_variable(

name="total\_negative\_quartic\_prediction",

shape=[], dtype=tf.float32)

total\_negative\_quintic\_prediction = \_metric\_variable(

name="total\_negative\_quintic\_prediction",

shape=[], dtype=tf.float32)

total\_loss = \_metric\_variable(name="total\_loss", shape=[], dtype=tf.float32)

# Update tracked stats

update\_total\_positive = tf.assign\_add(

total\_positive, tf.reduce\_sum(positive\_weights), name="total\_positive\_update")

update\_total\_weight = tf.assign\_add(

total\_weight, tf.reduce\_sum(weights), name="total\_weight\_update")

update\_total\_prediction = tf.assign\_add(

total\_prediction, tf.reduce\_sum(weighted\_predictions), name="total\_prediction\_update")

update\_total\_negative\_prediction = tf.assign\_add(

total\_negative\_prediction,

tf.reduce\_sum(weighted\_negative\_predictions), name="total\_negative\_prediction\_update")

update\_total\_negative\_squared\_prediction = tf.assign\_add(

total\_negative\_squared\_prediction,

tf.reduce\_sum(weighted\_negative\_squared\_predictions),

name="total\_negative\_squared\_prediction\_update")

update\_total\_negative\_cubed\_prediction = tf.assign\_add(

total\_negative\_cubed\_prediction,

tf.reduce\_sum(weighted\_negative\_cubed\_predictions),

name="total\_negative\_cubed\_prediction\_update")

update\_total\_negative\_quartic\_prediction = tf.assign\_add(

total\_negative\_quartic\_prediction,

tf.reduce\_sum(weighted\_negative\_quartic\_predictions),

name="total\_negative\_quartic\_prediction\_update")

update\_total\_negative\_quintic\_prediction = tf.assign\_add(

total\_negative\_quintic\_prediction,

tf.reduce\_sum(weighted\_negative\_quintic\_predictions),

name="total\_negative\_quintic\_prediction\_update")

update\_total\_loss = tf.assign\_add(

total\_loss, tf.reduce\_sum(weighted\_logloss), name="total\_loss\_update")

# metric value retrieval subgraph

# ctr of this batch

positive\_rate = tf.truediv(total\_positive, total\_weight, name="positive\_rate")

# Note: we don't have to keep running averages for computing baseline CE. Because the prediction

# is constant for every sample, we can simplify it to the formula below.

baseline\_loss = \_binary\_cross\_entropy(

pred=positive\_rate,

target=positive\_rate,

name="baseline\_loss")

# normalizing ratio for nrce

# calculated using total ctr and pctr so the last batch has the dataset ctr and pctr

normalizer = tf.truediv(total\_positive, total\_prediction, name="normalizer")

# Taylor expansion to calculate nl = - sum(y \* log(p \* a) + (1 - y) \* log (1 - p \* a))

# log(1 - p \* a) = -sum\_{i=1~+inf}(a^i \* x^i / i)

# log(1 - p) = -sum\_{i=1~+inf}(a^i \* x^i / i)

normalized\_loss = (

total\_loss -

total\_positive \* tf.log(normalizer) +

total\_negative\_prediction \* (normalizer - 1) +

total\_negative\_squared\_prediction \* (normalizer \* normalizer - 1) / 2 +

total\_negative\_cubed\_prediction \*

(normalizer \* normalizer \* normalizer - 1) / 3 +

total\_negative\_quartic\_prediction \*

(normalizer \* normalizer \* normalizer \* normalizer - 1) / 4 +

total\_negative\_quintic\_prediction \*

(normalizer \* normalizer \* normalizer \* normalizer \* normalizer - 1) / 5)

# average normalized loss

avg\_loss = tf.truediv(normalized\_loss, total\_weight, name="avg\_loss")

nrce\_t = tf.multiply(

1.0 - tf.truediv(avg\_loss, baseline\_loss),

100,

name="lolly\_nrce")

# metric update subgraph

update\_positive\_rate = tf.truediv(

update\_total\_positive,

update\_total\_weight,

name="update\_positive\_rate")

# Note: we don't have to keep running averages for computing baseline CE. Because the prediction

# is constant for every sample, we can simplify it to the formula below.

update\_baseline\_loss = \_binary\_cross\_entropy(

pred=update\_positive\_rate,

target=update\_positive\_rate,

name="update\_baseline\_loss")

update\_normalizer = tf.truediv(

update\_total\_positive,

update\_total\_prediction,

name="update\_normalizer")

update\_normalized\_loss = (

update\_total\_loss -

update\_total\_positive \* tf.log(update\_normalizer) +

update\_total\_negative\_prediction \*

(update\_normalizer - 1) +

update\_total\_negative\_squared\_prediction \*

(update\_normalizer \* update\_normalizer - 1) / 2 +

update\_total\_negative\_cubed\_prediction \*

(update\_normalizer \* update\_normalizer \* update\_normalizer - 1) / 3 +

update\_total\_negative\_quartic\_prediction \*

(update\_normalizer \* update\_normalizer \* update\_normalizer \*

update\_normalizer - 1) / 4 +

update\_total\_negative\_quintic\_prediction \*

(update\_normalizer \* update\_normalizer \* update\_normalizer \*

update\_normalizer \* update\_normalizer - 1) / 5)

update\_avg\_loss = tf.truediv(

update\_normalized\_loss,

update\_total\_weight,

name="update\_avg\_loss")

update\_op = tf.multiply(

1.0 - tf.truediv(update\_avg\_loss, update\_baseline\_loss),

100,

name="update\_op")

if metrics\_collections:

tf.add\_to\_collections(metrics\_collections, nrce\_t)

if updates\_collections:

tf.add\_to\_collections(updates\_collections, update\_op)

return nrce\_t, update\_op

def \_binary\_cross\_entropy(pred, target, name):

return - tf.add(

target \* tf.log(pred),

(1.0 - target) \* tf.log(1.0 - pred),

name=name)

# Copied from metrics\_impl.py with minor modifications.

# https://github.com/tensorflow/tensorflow/blob/v1.5.0/tensorflow/python/ops/metrics\_impl.py#L39

def \_metric\_variable(shape, dtype, validate\_shape=True, name=None):

"""Create variable in `GraphKeys.(LOCAL|METRIC\_VARIABLES`) collections."""

return tf.Variable(

lambda: tf.zeros(shape, dtype),

trainable=False,

collections=[tf.GraphKeys.LOCAL\_VARIABLES, tf.GraphKeys.METRIC\_VARIABLES],

validate\_shape=validate\_shape,

name=name)

PERCENTILES = np.linspace(0, 1, 101, dtype=np.float32)

# metric\_name: (metric, requires thresholded output)

SUPPORTED\_BINARY\_CLASS\_METRICS = {

# TWML metrics

'total\_weight': (total\_weight\_metric, False),

'num\_samples': (num\_samples\_metric, False),

'rce': (rce, False),

'rce\_std\_err': (partial(metric\_std\_err, transform=rce\_transform, metric=rce\_metric, name='rce\_std\_err'), False),

'nrce': (partial(rce, normalize=True), False),

'lolly\_nrce': (lolly\_nrce, False),

'arce': (partial(rce, normalize=True, arce=True), False),

'arce\_original': (partial(rce, normalize=True, arce=True, up\_weight=False), False),

# CTR measures positive sample ratio. This terminology is inherited from Ads.

'ctr': (ctr, False),

# predicted CTR measures predicted positive ratio.

'predicted\_ctr': (predicted\_ctr, False),

'pred\_std\_dev': (prediction\_std\_dev, False),

# thresholded metrics

'accuracy': (tf.metrics.accuracy, True),

'precision': (tf.metrics.precision, True),

'recall': (tf.metrics.recall, True),

'false\_positives': (tf.metrics.false\_positives, True),

'false\_negatives': (tf.metrics.false\_negatives, True),

'true\_positives': (tf.metrics.true\_positives, True),

'true\_negatives': (tf.metrics.true\_negatives, True),

'precision\_at\_percentiles': (partial(tf.metrics.precision\_at\_thresholds, thresholds=PERCENTILES), False),

'recall\_at\_percentiles': (partial(tf.metrics.recall\_at\_thresholds, thresholds=PERCENTILES), False),

'false\_positives\_at\_percentiles': (partial(tf.metrics.false\_positives\_at\_thresholds, thresholds=PERCENTILES), False),

'false\_negatives\_at\_percentiles': (partial(tf.metrics.false\_negatives\_at\_thresholds, thresholds=PERCENTILES), False),

'true\_positives\_at\_percentiles': (partial(tf.metrics.true\_positives\_at\_thresholds, thresholds=PERCENTILES), False),

'true\_negatives\_at\_percentiles': (partial(tf.metrics.true\_negatives\_at\_thresholds, thresholds=PERCENTILES), False),

# tensorflow metrics

'roc\_auc': (partial(tf.metrics.auc, curve='ROC',

summation\_method='careful\_interpolation'), False),

'pr\_auc': (partial(tf.metrics.auc, curve='PR',

summation\_method='careful\_interpolation'), False),

# tensorboard curves

'pr\_curve': (tb.summary.v1.pr\_curve\_streaming\_op, False),

# deprecated metrics

'deprecated\_nrce': (partial(rce, normalize=True, deprecated\_rce=True), False),

'deprecated\_arce': (partial(rce, normalize=True, arce=True, deprecated\_rce=True), False),

'deprecated\_arce\_original': (partial(rce, normalize=True, arce=True,

up\_weight=False, deprecated\_rce=True), False)

}

# default metrics provided by get\_binary\_class\_metric\_fn

DEFAULT\_BINARY\_CLASS\_METRICS = ['total\_weight', 'num\_samples', 'rce', 'rce\_std\_err',

'nrce', 'arce', 'ctr', 'predicted\_ctr', 'pred\_std\_dev',

'accuracy', 'precision', 'recall', 'roc\_auc', 'pr\_auc']

def get\_binary\_class\_metric\_fn(metrics=None):

"""

Returns a function having signature:

.. code-block:: python

def get\_eval\_metric\_ops(graph\_output, labels, weights):

...

return eval\_metric\_ops

where the returned eval\_metric\_ops is a dict of common evaluation metric

Ops for binary classification. See `tf.estimator.EstimatorSpec

<https://www.tensorflow.org/api\_docs/python/tf/estimator/EstimatorSpec>`\_

for a description of eval\_metric\_ops. The graph\_output is a the result

dict returned by build\_graph. Labels and weights are tf.Tensors.

The following graph\_output keys are recognized:

output:

the raw predictions between 0 and 1. Required.

threshold:

A value between 0 and 1 used to threshold the output into a hard\_output.

Defaults to 0.5 when threshold and hard\_output are missing.

Either threshold or hard\_output can be provided, but not both.

hard\_output:

A thresholded output. Either threshold or hard\_output can be provided, but not both.

Args:

metrics (list of String):

a list of metrics of interest. E.g. ['ctr', 'accuracy', 'rce']

Element in the list can be a string from following supported metrics, or can be a tuple

with three items: metric name, metric function, bool for thresholded output.

These metrics are evaluated and reported to tensorboard \*during the eval phases only\*.

Supported metrics:

- ctr (same as positive sample ratio.)

- rce (cross entropy loss compared to the baseline model of always predicting ctr)

- nrce (normalized rce, do not use this one if you do not understand what it is)

- `arce <http://go/arce>`\_ (a more recent proposed improvment over NRCE)

- arce\_original

- lolly\_nrce (NRCE as it is computed in Lolly, with Taylor expansion)

- pr\_auc

- roc\_auc

- accuracy (percentage of predictions that are correct)

- precision (true positives) / (true positives + false positives)

- recall (true positives) / (true positives + false negatives)

- pr\_curve (precision-recall curve)

- deprecated\_arce (ARCE as it was calculated before a stability fix)

- deprecated\_nrce (NRCE as it was calculated before a stability fix)

Example of metrics list with mixture of string and tuple:

metrics = [

'rce','nrce',

'roc\_auc', # default roc\_auc metric

(

'roc\_auc\_500', # give this metric a name

partial(tf.metrics.auc, curve='ROC', summation\_method='careful\_interpolation', num\_thresholds=500), # the metric fn

False, # whether the metric requires thresholded output

)]

NOTE: When predicting rare events roc\_auc can be underestimated. Increasing num\_threshold

can reduce the underestimation. See go/roc-auc-pitfall for more details.

NOTE: accuracy / precision / recall apply to binary classification problems only.

I.e. a prediction is only considered correct if it matches the label. E.g. if the label

is 1.0, and the prediction is 0.99, it does not get credit. If you want to use

precision / recall / accuracy metrics with soft predictions, you'll need to threshold

your predictions into hard 0/1 labels.

When metrics is None (the default), it defaults to:

[rce, nrce, arce, ctr, predicted\_ctr, accuracy, precision, recall, prauc, roc\_auc],

"""

# pylint: disable=dict-keys-not-iterating

if metrics is None:

# remove expensive metrics by default for faster eval

metrics = list(DEFAULT\_BINARY\_CLASS\_METRICS)

def get\_eval\_metric\_ops(graph\_output, labels, weights):

"""

graph\_output:

dict that is returned by build\_graph given input features.

labels:

target labels associated to batch.

weights:

weights of the samples..

"""

eval\_metric\_ops = OrderedDict()

preds = graph\_output['output']

threshold = graph\_output['threshold'] if 'threshold' in graph\_output else 0.5

hard\_preds = graph\_output.get('hard\_output')

if hard\_preds is None:

hard\_preds = tf.greater\_equal(preds, threshold)

# add metrics to eval\_metric\_ops dict

for metric in metrics:

if isinstance(metric, tuple) and len(metric) == 3:

metric\_name, metric\_factory, requires\_threshold = metric

metric\_name = metric\_name.lower()

elif isinstance(metric, str):

metric\_name = metric.lower() # metric name are case insensitive.

metric\_factory, requires\_threshold = SUPPORTED\_BINARY\_CLASS\_METRICS.get(metric\_name)

else:

raise ValueError("Metric should be either string or tuple of length 3.")

if metric\_name in eval\_metric\_ops:

# avoid adding duplicate metrics.

continue

if metric\_factory:

value\_op, update\_op = metric\_factory(

labels=labels,

predictions=(hard\_preds if requires\_threshold else preds),

weights=weights, name=metric\_name)

eval\_metric\_ops[metric\_name] = (value\_op, update\_op)

else:

raise ValueError('Cannot find the metric named ' + metric\_name)

return eval\_metric\_ops

return get\_eval\_metric\_ops

def get\_multi\_binary\_class\_metric\_fn(metrics, classes=None, class\_dim=1):

"""

Returns a function having signature:

.. code-block:: python

def get\_eval\_metric\_ops(graph\_output, labels, weights):

...

return eval\_metric\_ops

where the returned eval\_metric\_ops is a dict of common evaluation metric

Ops for concatenated binary classifications. See `tf.estimator.EstimatorSpec

<https://www.tensorflow.org/api\_docs/python/tf/estimator/EstimatorSpec>`\_

for a description of eval\_metric\_ops. The graph\_output is a the result

dict returned by build\_graph. Labels and weights are tf.Tensors.

In multiple binary classification problems, the

``predictions`` (that is, ``graph\_output['output']``)

are expected to have shape ``batch\_size x n\_classes``,

where ``n\_classes`` is the number of binary classification.

Binary classification at output[i] is expected to discriminate between ``classes[i]`` (1)

and NOT ``classes[i]`` (0). The labels should be of the same shape as ``graph\_output``

with binary values (0 or 1). The weights can be of size ``batch\_size`` or

``batch\_size x n\_classes``. The ``class\_dim`` contain separate probabilities,

and need to have separate metrics.

The following graph\_output keys are recognized:

output:

the raw predictions between 0 and 1. Required.

threshold:

A value between 0 and 1 used to threshold the output into a hard\_output.

Defaults to 0.5 when threshold and hard\_output are missing.

Either threshold or hard\_output can be provided, but not both.

hard\_output:

A thresholded output. Either threshold or hard\_output can be provided, but not both.

Args:

metrics (list of Metrics):

a list of metrics of interest. E.g. ['ctr', 'accuracy', 'rce']

Element in the list can be a string from following supported metrics, or can be a tuple

with three items: metric name, metric function, bool for thresholded output.

These metrics are evaluated and reported to tensorboard \*during the eval phases only\*.

Supported metrics:

- ctr (same as positive sample ratio.)

- rce (cross entropy loss compared to the baseline model of always predicting ctr)

- nrce (normalized rce, do not use this one if you do not understand what it is)

- pr\_auc

- roc\_auc

- accuracy (percentage of predictions that are correct)

- precision (true positives) / (true positives + false positives)

- recall (true positives) / (true positives + false negatives)

- pr\_curve (precision-recall curve)

Example of metrics list with mixture of string and tuple:

metrics = [

'rce','nrce',

'roc\_auc', # default roc\_auc metric

(

'roc\_auc\_500', # give this metric a name

partial(tf.metrics.auc, curve='ROC', summation\_method='careful\_interpolation', num\_thresholds=500), # the metric fn

False, # whether the metric requires thresholded output

)]

NOTE: When prediction on rare events, roc\_auc can be underestimated. Increase num\_threshold

can reduce the underestimation. See go/roc-auc-pitfall for more details.

NOTE: accuracy / precision / recall apply to binary classification problems only.

I.e. a prediction is only considered correct if it matches the label. E.g. if the label

is 1.0, and the prediction is 0.99, it does not get credit. If you want to use

precision / recall / accuracy metrics with soft predictions, you'll need to threshold

your predictions into hard 0/1 labels.

When metrics is None (the default), it defaults to:

[rce, nrce, arce, ctr, predicted\_ctr, accuracy, precision, recall, prauc, roc\_auc],

classes (list of strings):

In case of multiple binary class models, the names for each class or label.

These are used to display metrics on tensorboard.

If these are not specified, the index in the class or label dimension is used, and you'll

get metrics on tensorboard named like: accuracy\_0, accuracy\_1, etc.

class\_dim (number):

Dimension of the classes in predictions. Defaults to 1, that is, batch\_size x n\_classes.

"""

# pylint: disable=invalid-name,dict-keys-not-iterating

if metrics is None:

# remove expensive metrics by default for faster eval

metrics = list(DEFAULT\_BINARY\_CLASS\_METRICS)

def get\_eval\_metric\_ops(graph\_output, labels, weights):

"""

graph\_output:

dict that is returned by build\_graph given input features.

labels:

target labels associated to batch.

weights:

weights of the samples..

"""

eval\_metric\_ops = OrderedDict()

preds = graph\_output['output']

threshold = graph\_output['threshold'] if 'threshold' in graph\_output else 0.5

hard\_preds = graph\_output.get('hard\_output')

if hard\_preds is None:

hard\_preds = tf.greater\_equal(preds, threshold)

shape = labels.get\_shape()

# basic sanity check: multi\_metric dimension must exist

assert len(shape) > class\_dim, "Dimension specified by class\_dim does not exist."

num\_labels = shape[class\_dim]

# If we are doing multi-class / multi-label metric, the number of classes / labels must

# be know at graph construction time. This dimension cannot have size None.

assert num\_labels is not None, "The multi-metric dimension cannot be None."

assert classes is None or len(classes) == num\_labels, (

"Number of classes must match the number of labels")

weights\_shape = weights.get\_shape() if weights is not None else None

if weights\_shape is None:

num\_weights = None

elif len(weights\_shape) > 1:

num\_weights = weights\_shape[class\_dim]

else:

num\_weights = 1

for i in range(num\_labels):

# add metrics to eval\_metric\_ops dict

for metric in metrics:

if isinstance(metric, tuple) and len(metric) == 3:

metric\_name, metric\_factory, requires\_threshold = metric

metric\_name = metric\_name.lower()

elif isinstance(metric, str):

metric\_name = metric.lower() # metric name are case insensitive.

metric\_factory, requires\_threshold = SUPPORTED\_BINARY\_CLASS\_METRICS.get(metric\_name)

else:

raise ValueError("Metric should be either string or tuple of length 3.")

class\_metric\_name = metric\_name + "\_" + (classes[i] if classes is not None else str(i))

if class\_metric\_name in eval\_metric\_ops:

# avoid adding duplicate metrics.

continue

class\_labels = tf.gather(labels, indices=[i], axis=class\_dim)

class\_preds = tf.gather(preds, indices=[i], axis=class\_dim)

class\_hard\_preds = tf.gather(hard\_preds, indices=[i], axis=class\_dim)

if num\_weights is None:

class\_weights = None

elif num\_weights == num\_labels:

class\_weights = tf.gather(weights, indices=[i], axis=class\_dim)

elif num\_weights == 1:

class\_weights = weights

else:

raise ValueError("num\_weights (%d) and num\_labels (%d) do not match"

% (num\_weights, num\_labels))

if metric\_factory:

value\_op, update\_op = metric\_factory(

labels=class\_labels,

predictions=(class\_hard\_preds if requires\_threshold else class\_preds),

weights=class\_weights, name=class\_metric\_name)

eval\_metric\_ops[class\_metric\_name] = (value\_op, update\_op)

else:

raise ValueError('Cannot find the metric named ' + metric\_name)

return eval\_metric\_ops

return get\_eval\_metric\_ops

def \_get\_uncalibrated\_metric\_fn(calibrated\_metric\_fn, keep\_weight=True):

"""

Returns a function having signature:

.. code-block:: python

def get\_eval\_metric\_ops(graph\_output, labels, weights):

...

return eval\_metric\_ops

where the returned eval\_metric\_ops is a dict of common evaluation metric

Ops with uncalibrated output.

The following graph\_output keys are recognized:

uncalibrated\_output:

the uncalibrated raw predictions between 0 and 1. Required.

output:

the calibrated predictions between 0 and 1.

threshold:

A value between 0 and 1 used to threshold the output into a hard\_output.

Defaults to 0.5 when threshold and hard\_output are missing.

Either threshold or hard\_output can be provided, but not both.

hard\_output:

A thresholded output. Either threshold or hard\_output can be provided, but not both.

Args:

calibrated\_metric\_fn: metrics function with calibration and weight.

keep\_weight: Bool indicating whether we keep weight.

"""

metric\_scope = 'uncalibrated' if keep\_weight else 'unweighted'

def get\_eval\_metric\_ops(graph\_output, labels, weights):

"""

graph\_output:

dict that is returned by build\_graph given input features.

labels:

target labels associated to batch.

weights:

weights of the samples..

"""

with tf.variable\_scope(metric\_scope):

if 'uncalibrated\_output' not in graph\_output:

raise Exception("Missing uncalibrated\_output in graph\_output!")

un\_calibrated\_weights = weights if keep\_weight else tf.ones\_like(weights)

uncalibrated\_output = {

'output': graph\_output['uncalibrated\_output'],

'threshold': graph\_output.get('threshold', 0.5),

'hard\_output': graph\_output.get('hard\_output'),

\*\*{k: v for k, v in graph\_output.items() if k not in ['output', 'threshold', 'hard\_output']}

}

eval\_metrics\_ops = calibrated\_metric\_fn(uncalibrated\_output, labels, un\_calibrated\_weights)

renamed\_metrics\_ops = {f'{metric\_scope}\_{k}': v for k, v in eval\_metrics\_ops.items()}

return renamed\_metrics\_ops

return get\_eval\_metric\_ops

def get\_multi\_binary\_class\_uncalibrated\_metric\_fn(

metrics, classes=None, class\_dim=1, keep\_weight=True):

"""

Returns a function having signature:

.. code-block:: python

def get\_eval\_metric\_ops(graph\_output, labels, weights):

...

return eval\_metric\_ops

where the returned eval\_metric\_ops is a dict of common evaluation metric

Ops for concatenated binary classifications without calibration.

Note: 'uncalibrated\_output' is required key in graph\_output.

The main use case for this function is:

1) To calculated roc-auc for rare event.

Calibrated prediction score for rare events will be concentrated near zero. As a result,

the roc-auc can be seriously underestimated with current implementation in tf.metric.auc.

Since roc-auc is invariant against calibration, we can directly use uncalibrated score for roc-auc.

For more details, please refer to: go/roc-auc-invariance.

2) To set keep\_weight=False and get unweighted and uncalibrated metrics.

This is useful to eval how the model is fitted to its actual training data, since

often time the model is trained without weight.

Args:

metrics (list of String):

a list of metrics of interest. E.g. ['ctr', 'accuracy', 'rce']

Element in the list can be a string from supported metrics, or can be a tuple

with three items: metric name, metric function, bool for thresholded output.

These metrics are evaluated and reported to tensorboard \*during the eval phases only\*.

When metrics is None (the default), it defaults to:

[rce, nrce, arce, ctr, predicted\_ctr, accuracy, precision, recall, prauc, roc\_auc],

classes (list of strings):

In case of multiple binary class models, the names for each class or label.

These are used to display metrics on tensorboard.

If these are not specified, the index in the class or label dimension is used, and you'll

get metrics on tensorboard named like: accuracy\_0, accuracy\_1, etc.

class\_dim (number):

Dimension of the classes in predictions. Defaults to 1, that is, batch\_size x n\_classes.

keep\_weight (bool):

Whether to keep weights for the metric.

"""

calibrated\_metric\_fn = get\_multi\_binary\_class\_metric\_fn(

metrics, classes=classes, class\_dim=class\_dim)

return \_get\_uncalibrated\_metric\_fn(calibrated\_metric\_fn, keep\_weight=keep\_weight)

def combine\_metric\_fns(\*fn\_list):

"""

Combine multiple metric functions.

For example, we can combine metrics function generated by

get\_multi\_binary\_class\_metric\_fn and get\_multi\_binary\_class\_uncalibrated\_metric\_fn.

Args:

\*fn\_list: Multiple metric functions to be combined

Returns:

Combined metric function.

"""

def combined\_metric\_ops(\*args, \*\*kwargs):

eval\_metric\_ops = OrderedDict()

for fn in fn\_list:

eval\_metric\_ops.update(fn(\*args, \*\*kwargs))

return eval\_metric\_ops

return combined\_metric\_ops