# pylint: disable=invalid-name, no-member, unused-argument

"""

This module contains common calibrate and export functions for calibrators.

"""

# These 3 TODO are encapsulated by CX-11446

# TODO: many of these functions hardcode datarecords yet don't allow passing a parse\_fn.

# TODO: provide more generic (non DataRecord specific) functions

# TODO: many of these functions aren't common at all.

# For example, Discretizer functions should be moved to PercentileDiscretizer.

import copy

import os

import time

from absl import logging

import tensorflow.compat.v1 as tf

import tensorflow\_hub as hub

import twml

from twml.argument\_parser import SortingHelpFormatter

from twml.input\_fns import data\_record\_input\_fn

from twml.util import list\_files\_by\_datetime, sanitize\_hdfs\_path

from twml.contrib.calibrators.isotonic import IsotonicCalibrator

def calibrator\_arguments(parser):

"""

Calibrator Parameters to add to relevant parameters to the DataRecordTrainerParser.

Otherwise, if alone in a file, it just creates its own default parser.

Arguments:

parser:

Parser with the options to the model

"""

parser.add\_argument("--calibrator.save\_dir", type=str,

dest="calibrator\_save\_dir",

help="Path to save or load calibrator calibration")

parser.add\_argument("--calibrator\_batch\_size", type=int, default=128,

dest="calibrator\_batch\_size",

help="calibrator batch size")

parser.add\_argument("--calibrator\_parts\_downsampling\_rate", type=float, default=1,

dest="calibrator\_parts\_downsampling\_rate",

help="Parts downsampling rate")

parser.add\_argument("--calibrator\_max\_steps", type=int, default=None,

dest="calibrator\_max\_steps",

help="Max Steps taken by calibrator to accumulate samples")

parser.add\_argument("--calibrator\_num\_bins", type=int, default=22,

dest="calibrator\_num\_bins",

help="Num bins of calibrator")

parser.add\_argument("--isotonic\_calibrator", dest='isotonic\_calibrator', action='store\_true',

help="Isotonic Calibrator present")

parser.add\_argument("--calibrator\_keep\_rate", type=float, default=1.0,

dest="calibrator\_keep\_rate",

help="Keep rate")

return parser

def \_generate\_files\_by\_datetime(params):

files = list\_files\_by\_datetime(

base\_path=sanitize\_hdfs\_path(params.train\_data\_dir),

start\_datetime=params.train\_start\_datetime,

end\_datetime=params.train\_end\_datetime,

datetime\_prefix\_format=params.datetime\_format,

extension="lzo",

parallelism=1,

hour\_resolution=params.hour\_resolution,

sort=True)

return files

def get\_calibrate\_input\_fn(parse\_fn, params):

"""

Default input function used for the calibrator.

Arguments:

parse\_fn:

Parse\_fn

params:

Parameters

Returns:

input\_fn

"""

return lambda: data\_record\_input\_fn(

files=\_generate\_files\_by\_datetime(params),

batch\_size=params.calibrator\_batch\_size,

parse\_fn=parse\_fn,

num\_threads=1,

repeat=False,

keep\_rate=params.calibrator\_keep\_rate,

parts\_downsampling\_rate=params.calibrator\_parts\_downsampling\_rate,

shards=None,

shard\_index=None,

shuffle=True,

shuffle\_files=True,

interleave=True)

def get\_discretize\_input\_fn(parse\_fn, params):

"""

Default input function used for the calibrator.

Arguments:

parse\_fn:

Parse\_fn

params:

Parameters

Returns:

input\_fn

"""

return lambda: data\_record\_input\_fn(

files=\_generate\_files\_by\_datetime(params),

batch\_size=params.discretizer\_batch\_size,

parse\_fn=parse\_fn,

num\_threads=1,

repeat=False,

keep\_rate=params.discretizer\_keep\_rate,

parts\_downsampling\_rate=params.discretizer\_parts\_downsampling\_rate,

shards=None,

shard\_index=None,

shuffle=True,

shuffle\_files=True,

interleave=True)

def discretizer\_arguments(parser=None):

"""

Discretizer Parameters to add to relevant parameters to the DataRecordTrainerParser.

Otherwise, if alone in a file, it just creates its own default parser.

Arguments:

parser:

Parser with the options to the model. Defaults to None

"""

if parser is None:

parser = twml.DefaultSubcommandArgParse(formatter\_class=SortingHelpFormatter)

parser.add\_argument(

"--overwrite\_save\_dir", dest="overwrite\_save\_dir", action="store\_true",

help="Delete the contents of the current save\_dir if it exists")

parser.add\_argument(

"--train.data\_dir", "--train\_data\_dir", type=str, default=None,

dest="train\_data\_dir",

help="Path to the training data directory."

"Supports local and HDFS (hdfs://default/<path> ) paths.")

parser.add\_argument(

"--train.start\_date", "--train\_start\_datetime",

type=str, default=None,

dest="train\_start\_datetime",

help="Starting date for training inside the train data dir."

"The start datetime is inclusive."

"e.g. 2019/01/15")

parser.add\_argument(

"--train.end\_date", "--train\_end\_datetime", type=str, default=None,

dest="train\_end\_datetime",

help="Ending date for training inside the train data dir."

"The end datetime is inclusive."

"e.g. 2019/01/15")

parser.add\_argument(

"--datetime\_format", type=str, default="%Y/%m/%d",

help="Date format for training and evaluation datasets."

"Has to be a format that is understood by python datetime."

"e.g. %Y/%m/%d for 2019/01/15."

"Used only if {train/eval}.{start/end}\_date are provided.")

parser.add\_argument(

"--hour\_resolution", type=int, default=None,

help="Specify the hourly resolution of the stored data.")

parser.add\_argument(

"--tensorboard\_port", type=int, default=None,

help="Port for tensorboard to run on.")

parser.add\_argument(

"--stats\_port", type=int, default=None,

help="Port for stats server to run on.")

parser.add\_argument(

"--health\_port", type=int, default=None,

help="Port to listen on for health-related endpoints (e.g. graceful shutdown)."

"Not user-facing as it is set automatically by the twml\_cli."

)

parser.add\_argument(

"--data\_spec", type=str, default=None,

help="Path to data specification JSON file. This file is used to decode DataRecords")

parser.add\_argument("--discretizer.save\_dir", type=str,

dest="discretizer\_save\_dir",

help="Path to save or load discretizer calibration")

parser.add\_argument("--discretizer\_batch\_size", type=int, default=128,

dest="discretizer\_batch\_size",

help="Discretizer batch size")

parser.add\_argument("--discretizer\_keep\_rate", type=float, default=0.0008,

dest="discretizer\_keep\_rate",

help="Keep rate")

parser.add\_argument("--discretizer\_parts\_downsampling\_rate", type=float, default=0.2,

dest="discretizer\_parts\_downsampling\_rate",

help="Parts downsampling rate")

parser.add\_argument("--discretizer\_max\_steps", type=int, default=None,

dest="discretizer\_max\_steps",

help="Max Steps taken by discretizer to accumulate samples")

return parser

def calibrate(trainer, params, build\_graph, input\_fn, debug=False):

"""

Calibrate Isotonic Calibration

Arguments:

trainer:

Trainer

params:

Parameters

build\_graph:

Build Graph used to be the input to the calibrator

input\_fn:

Input Function specified by the user

debug:

Defaults to False. Returns the calibrator

"""

if trainer.\_estimator.config.is\_chief:

# overwrite the current save\_dir

if params.overwrite\_save\_dir and tf.io.gfile.exists(params.calibrator\_save\_dir):

logging.info("Trainer overwriting existing save directory: %s (params.overwrite\_save\_dir)"

% params.calibrator\_save\_dir)

tf.io.gfile.rmtree(params.calibrator\_save\_dir)

calibrator = IsotonicCalibrator(params.calibrator\_num\_bins)

# chief trains discretizer

logging.info("Chief training calibrator")

# Accumulate the features for each calibrator

features, labels = input\_fn()

if 'weights' not in features:

raise ValueError("Weights need to be returned as part of the parse\_fn")

weights = features.pop('weights')

preds = build\_graph(features=features, label=None, mode='infer', params=params, config=None)

init = tf.global\_variables\_initializer()

table\_init = tf.tables\_initializer()

with tf.Session() as sess:

sess.run(init)

sess.run(table\_init)

count = 0

max\_steps = params.calibrator\_max\_steps or -1

while max\_steps <= 0 or count <= max\_steps:

try:

weights\_vals, labels\_vals, preds\_vals = sess.run([weights, labels, preds['output']])

calibrator.accumulate(preds\_vals, labels\_vals, weights\_vals.flatten())

except tf.errors.OutOfRangeError:

break

count += 1

calibrator.calibrate()

calibrator.save(params.calibrator\_save\_dir)

trainer.estimator.\_params.isotonic\_calibrator = True

if debug:

return calibrator

else:

calibrator\_save\_dir = twml.util.sanitize\_hdfs\_path(params.calibrator\_save\_dir)

# workers wait for calibration to be ready

while not tf.io.gfile.exists(calibrator\_save\_dir + os.path.sep + "tfhub\_module.pb"):

logging.info("Worker waiting for calibration at %s" % calibrator\_save\_dir)

time.sleep(60)

def discretize(params, feature\_config, input\_fn, debug=False):

"""

Discretizes continuous features

Arguments:

params:

Parameters

input\_fn:

Input Function specified by the user

debug:

Defaults to False. Returns the calibrator

"""

if (os.environ.get("TWML\_HOGWILD\_TASK\_TYPE") == "chief" or "num\_workers" not in params or

params.num\_workers is None):

# overwrite the current save\_dir

if params.overwrite\_save\_dir and tf.io.gfile.exists(params.discretizer\_save\_dir):

logging.info("Trainer overwriting existing save directory: %s (params.overwrite\_save\_dir)"

% params.discretizer\_save\_dir)

tf.io.gfile.rmtree(params.discretizer\_save\_dir)

config\_map = feature\_config()

discretize\_dict = config\_map['discretize\_config']

# chief trains discretizer

logging.info("Chief training discretizer")

batch = input\_fn()

# Accumulate the features for each calibrator

with tf.Session() as sess:

count = 0

max\_steps = params.discretizer\_max\_steps or -1

while max\_steps <= 0 or count <= max\_steps:

try:

inputs = sess.run(batch)

for name, clbrt in discretize\_dict.items():

clbrt.accumulate\_features(inputs[0], name)

except tf.errors.OutOfRangeError:

break

count += 1

# 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

for name, clbrt in discretize\_dict.items():

clbrt.calibrate()

clbrt.add\_hub\_signatures(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(params.discretizer\_save\_dir, session)

for name, clbrt in discretize\_dict.items():

clbrt.write\_summary\_json(params.discretizer\_save\_dir, name)

if debug:

return discretize\_dict

else:

# wait for the file to be removed (if necessary)

# should be removed after an actual fix applied

time.sleep(60)

discretizer\_save\_dir = twml.util.sanitize\_hdfs\_path(params.discretizer\_save\_dir)

# workers wait for calibration to be ready

while not tf.io.gfile.exists(discretizer\_save\_dir + os.path.sep + "tfhub\_module.pb"):

logging.info("Worker waiting for calibration at %s" % discretizer\_save\_dir)

time.sleep(60)

def add\_discretizer\_arguments(parser):

"""

Add discretizer-specific command-line arguments to a Trainer parser.

Arguments:

parser: argparse.ArgumentParser instance obtained from Trainer.get\_trainer\_parser

Returns:

argparse.ArgumentParser instance with discretizer-specific arguments added

"""

parser.add\_argument("--discretizer.save\_dir", type=str,

dest="discretizer\_save\_dir",

help="Path to save or load discretizer calibration")

parser.add\_argument("--discretizer.batch\_size", type=int, default=128,

dest="discretizer\_batch\_size",

help="Discretizer batch size")

parser.add\_argument("--discretizer.keep\_rate", type=float, default=0.0008,

dest="discretizer\_keep\_rate",

help="Keep rate")

parser.add\_argument("--discretizer.parts\_downsampling\_rate", type=float, default=0.2,

dest="discretizer\_parts\_downsampling\_rate",

help="Parts downsampling rate")

parser.add\_argument("--discretizer.num\_bins", type=int, default=20,

dest="discretizer\_num\_bins",

help="Number of bins per feature")

parser.add\_argument("--discretizer.output\_size\_bits", type=int, default=22,

dest="discretizer\_output\_size\_bits",

help="Number of bits allocated to the output size")

return parser

def add\_isotonic\_calibrator\_arguments(parser):

"""

Add discretizer-specific command-line arguments to a Trainer parser.

Arguments:

parser: argparse.ArgumentParser instance obtained from Trainer.get\_trainer\_parser

Returns:

argparse.ArgumentParser instance with discretizer-specific arguments added

"""

parser.add\_argument("--calibrator.num\_bins", type=int,

default=25000, dest="calibrator\_num\_bins",

help="number of bins for isotonic calibration")

parser.add\_argument("--calibrator.parts\_downsampling\_rate", type=float, default=0.1,

dest="calibrator\_parts\_downsampling\_rate", help="Parts downsampling rate")

parser.add\_argument("--calibrator.save\_dir", type=str,

dest="calibrator\_save\_dir", help="Path to save or load calibrator output")

parser.add\_argument("--calibrator.load\_tensorflow\_module", type=str, default=None,

dest="calibrator\_load\_tensorflow\_module",

help="Location from where to load a pretrained graph from. \

Typically, this is where the MLP graph is saved")

parser.add\_argument("--calibrator.export\_mlp\_module\_name", type=str, default='tf\_hub\_mlp',

help="Name for loaded hub signature",

dest="export\_mlp\_module\_name")

parser.add\_argument("--calibrator.export\_isotonic\_module\_name",

type=str, default="tf\_hub\_isotonic",

dest="calibrator\_export\_module\_name",

help="export module name")

parser.add\_argument("--calibrator.final\_evaluation\_steps", type=int,

dest="calibrator\_final\_evaluation\_steps", default=None,

help="number of steps for final evaluation")

parser.add\_argument("--calibrator.train\_steps", type=int, default=-1,

dest="calibrator\_train\_steps",

help="number of steps for calibration")

parser.add\_argument("--calibrator.batch\_size", type=int, default=1024,

dest="calibrator\_batch\_size",

help="Calibrator batch size")

parser.add\_argument("--calibrator.is\_calibrating", action='store\_true',

dest="is\_calibrating",

help="Dummy argument to allow running in chief worker")

return parser

def calibrate\_calibrator\_and\_export(name, calibrator, build\_graph\_fn, params, feature\_config,

run\_eval=True, input\_fn=None, metric\_fn=None,

export\_task\_type\_overrider=None):

"""

Pre-set `isotonic calibrator` calibrator.

Args:

name:

scope name used for the calibrator

calibrator:

calibrator that will be calibrated and exported.

build\_graph\_fn:

build graph function for the calibrator

params:

params passed to the calibrator

feature\_config:

feature config which will be passed to the trainer

export\_task\_type\_overrider:

the task type for exporting the calibrator

if specified, this will override the default export task type in trainer.hub\_export(..)

"""

# create calibrator params

params\_c = copy.deepcopy(params)

params\_c.data\_threads = 1

params\_c.num\_workers = 1

params\_c.continue\_from\_checkpoint = True

params\_c.overwrite\_save\_dir = False

params\_c.stats\_port = None

# Automatically load from the saved Tensorflow Hub module if not specified.

if params\_c.calibrator\_load\_tensorflow\_module is None:

path\_saved\_tensorflow\_model = os.path.join(params.save\_dir, params.export\_mlp\_module\_name)

params\_c.calibrator\_load\_tensorflow\_module = path\_saved\_tensorflow\_model

if "calibrator\_parts\_downsampling\_rate" in params\_c:

params\_c.train\_parts\_downsampling\_rate = params\_c.calibrator\_parts\_downsampling\_rate

if "calibrator\_save\_dir" in params\_c:

params\_c.save\_dir = params\_c.calibrator\_save\_dir

if "calibrator\_batch\_size" in params\_c:

params\_c.train\_batch\_size = params\_c.calibrator\_batch\_size

params\_c.eval\_batch\_size = params\_c.calibrator\_batch\_size

# TODO: Deprecate this option. It is not actually used. Calibrator

# simply iterates until the end of input\_fn.

if "calibrator\_train\_steps" in params\_c:

params\_c.train\_steps = params\_c.calibrator\_train\_steps

if metric\_fn is None:

metric\_fn = twml.metrics.get\_multi\_binary\_class\_metric\_fn(None)

# Common Trainer which will also be used by all workers

trainer = twml.trainers.DataRecordTrainer(

name=name,

params=params\_c,

feature\_config=feature\_config,

build\_graph\_fn=build\_graph\_fn,

save\_dir=params\_c.save\_dir,

metric\_fn=metric\_fn

)

if trainer.\_estimator.config.is\_chief:

# Chief trains calibrator

logging.info("Chief training calibrator")

# Disregard hogwild config

os\_twml\_hogwild\_ports = os.environ.get("TWML\_HOGWILD\_PORTS")

os.environ["TWML\_HOGWILD\_PORTS"] = ""

hooks = None

if params\_c.calibrator\_train\_steps > 0:

hooks = [twml.hooks.StepProgressHook(params\_c.calibrator\_train\_steps)]

def parse\_fn(input\_x):

fc\_parse\_fn = feature\_config.get\_parse\_fn()

features, labels = fc\_parse\_fn(input\_x)

features['labels'] = labels

return features, labels

if input\_fn is None:

input\_fn = trainer.get\_train\_input\_fn(parse\_fn=parse\_fn, repeat=False)

# Calibrate stage

trainer.estimator.\_params.mode = 'calibrate'

trainer.calibrate(calibrator=calibrator,

input\_fn=input\_fn,

steps=params\_c.calibrator\_train\_steps,

hooks=hooks)

# Save Checkpoint

# We need to train for 1 step, to save the graph to checkpoint.

# This is done just by the chief.

# We need to set the mode to evaluate to save the graph that will be consumed

# In the final evaluation

trainer.estimator.\_params.mode = 'evaluate'

trainer.train(input\_fn=input\_fn, steps=1)

# Restore hogwild setup

if os\_twml\_hogwild\_ports is not None:

os.environ["TWML\_HOGWILD\_PORTS"] = os\_twml\_hogwild\_ports

else:

# Workers wait for calibration to be ready

final\_calibrator\_path = os.path.join(params\_c.calibrator\_save\_dir,

params\_c.calibrator\_export\_module\_name)

final\_calibrator\_path = twml.util.sanitize\_hdfs\_path(final\_calibrator\_path)

while not tf.io.gfile.exists(final\_calibrator\_path + os.path.sep + "tfhub\_module.pb"):

logging.info("Worker waiting for calibration at %s" % final\_calibrator\_path)

time.sleep(60)

# Evaluate stage

if run\_eval:

trainer.estimator.\_params.mode = 'evaluate'

# This will allow the Evaluate method to be run in Hogwild

# trainer.estimator.\_params.continue\_from\_checkpoint = True

trainer.evaluate(name='test', input\_fn=input\_fn, steps=params\_c.calibrator\_final\_evaluation\_steps)

trainer.hub\_export(name=params\_c.calibrator\_export\_module\_name,

export\_task\_type\_overrider=export\_task\_type\_overrider,

serving\_input\_receiver\_fn=feature\_config.get\_serving\_input\_receiver\_fn())

return trainer

def calibrate\_discretizer\_and\_export(name, calibrator, build\_graph\_fn, params, feature\_config):

"""

Pre-set percentile discretizer calibrator.

Args:

name:

scope name used for the calibrator

calibrator:

calibrator that will be calibrated and exported.

build\_graph\_fn:

build graph function for the calibrator

params:

params passed to the calibrator

feature\_config:

feature config or input\_fn which will be passed to the trainer.

"""

if (os.environ.get("TWML\_HOGWILD\_TASK\_TYPE") == "chief" or "num\_workers" not in params or

params.num\_workers is None):

# chief trains discretizer

logging.info("Chief training discretizer")

# disregard hogwild config

os\_twml\_hogwild\_ports = os.environ.get("TWML\_HOGWILD\_PORTS")

os.environ["TWML\_HOGWILD\_PORTS"] = ""

# create discretizer params

params\_c = copy.deepcopy(params)

params\_c.data\_threads = 1

params\_c.train\_steps = -1

params\_c.train\_max\_steps = None

params\_c.eval\_steps = -1

params\_c.num\_workers = 1

params\_c.tensorboard\_port = None

params\_c.stats\_port = None

if "discretizer\_batch\_size" in params\_c:

params\_c.train\_batch\_size = params\_c.discretizer\_batch\_size

params\_c.eval\_batch\_size = params\_c.discretizer\_batch\_size

if "discretizer\_keep\_rate" in params\_c:

params\_c.train\_keep\_rate = params\_c.discretizer\_keep\_rate

if "discretizer\_parts\_downsampling\_rate" in params\_c:

params\_c.train\_parts\_downsampling\_rate = params\_c.discretizer\_parts\_downsampling\_rate

if "discretizer\_save\_dir" in params\_c:

params\_c.save\_dir = params\_c.discretizer\_save\_dir

# train discretizer

trainer = twml.trainers.DataRecordTrainer(

name=name,

params=params\_c,

build\_graph\_fn=build\_graph\_fn,

save\_dir=params\_c.save\_dir,

)

if isinstance(feature\_config, twml.feature\_config.FeatureConfig):

parse\_fn = twml.parsers.get\_continuous\_parse\_fn(feature\_config)

input\_fn = trainer.get\_train\_input\_fn(parse\_fn=parse\_fn, repeat=False)

elif callable(feature\_config):

input\_fn = feature\_config

else:

got\_type = type(feature\_config).\_\_name\_\_

raise ValueError(

"Expecting feature\_config to be FeatureConfig or function got %s" % got\_type)

hooks = None

if params\_c.train\_steps > 0:

hooks = [twml.hooks.StepProgressHook(params\_c.train\_steps)]

trainer.calibrate(calibrator=calibrator, input\_fn=input\_fn,

steps=params\_c.train\_steps, hooks=hooks)

# restore hogwild setup

if os\_twml\_hogwild\_ports is not None:

os.environ["TWML\_HOGWILD\_PORTS"] = os\_twml\_hogwild\_ports

else:

discretizer\_save\_dir = twml.util.sanitize\_hdfs\_path(params.discretizer\_save\_dir)

# workers wait for calibration to be ready

while not tf.io.gfile.exists(discretizer\_save\_dir + os.path.sep + "tfhub\_module.pb"):

logging.info("Worker waiting for calibration at %s" % discretizer\_save\_dir)

time.sleep(60)

def build\_percentile\_discretizer\_graph(features, label, mode, params, config=None):

"""

Pre-set Percentile Discretizer Build Graph

Follows the same signature as build\_graph

"""

sparse\_tf = twml.util.convert\_to\_sparse(features, params.input\_size\_bits)

weights = tf.reshape(features['weights'], tf.reshape(features['batch\_size'], [1]))

if isinstance(sparse\_tf, tf.SparseTensor):

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

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

elif isinstance(sparse\_tf, twml.SparseTensor):

indices = sparse\_tf.indices

ids = sparse\_tf.ids

# Return weights, feature\_ids, feature\_values

weights = tf.gather(params=weights, indices=ids)

feature\_ids = indices

feature\_values = sparse\_tf.values

# Update train\_op and assign dummy\_loss

train\_op = tf.assign\_add(tf.train.get\_global\_step(), 1)

loss = tf.constant(1)

if mode == 'train':

return {'train\_op': train\_op, 'loss': loss}

return {'feature\_ids': feature\_ids, 'feature\_values': feature\_values, 'weights': weights}

def isotonic\_module(mode, params):

"""

Common Isotonic Calibrator module for Hub Export

"""

inputs = tf.sparse\_placeholder(tf.float32, name="sparse\_input")

mlp = hub.Module(params.calibrator\_load\_tensorflow\_module)

logits = mlp(inputs, signature=params.export\_mlp\_module\_name)

isotonic\_calibrator = hub.Module(params.save\_dir)

output = isotonic\_calibrator(logits, signature="isotonic\_calibrator")

hub.add\_signature(inputs={"sparse\_input": inputs},

outputs={"default": output},

name=params.calibrator\_export\_module\_name)

def build\_isotonic\_graph\_from\_inputs(inputs, features, label, mode, params, config=None, isotonic\_fn=None):

"""

Helper function to build\_isotonic\_graph

Pre-set Isotonic Calibrator Build Graph

Follows the same signature as build\_graph

"""

if params.mode == 'calibrate':

mlp = hub.Module(params.calibrator\_load\_tensorflow\_module)

logits = mlp(inputs, signature=params.export\_mlp\_module\_name)

weights = tf.reshape(features['weights'], tf.reshape(features['batch\_size'], [1]))

# Update train\_op and assign dummy\_loss

train\_op = tf.assign\_add(tf.train.get\_global\_step(), 1)

loss = tf.constant(1)

if mode == 'train':

return {'train\_op': train\_op, 'loss': loss}

return {'predictions': logits, 'targets': features['labels'], 'weights': weights}

else:

if isotonic\_fn is None:

isotonic\_spec = twml.util.create\_module\_spec(mlp\_fn=isotonic\_module, mode=mode, params=params)

else:

isotonic\_spec = twml.util.create\_module\_spec(mlp\_fn=isotonic\_fn, mode=mode, params=params)

output\_hub = hub.Module(isotonic\_spec,

name=params.calibrator\_export\_module\_name)

hub.register\_module\_for\_export(output\_hub, params.calibrator\_export\_module\_name)

output = output\_hub(inputs, signature=params.calibrator\_export\_module\_name)

output = tf.clip\_by\_value(output, 0, 1)

loss = tf.reduce\_sum(tf.stop\_gradient(output))

train\_op = tf.assign\_add(tf.train.get\_global\_step(), 1)

return {'train\_op': train\_op, 'loss': loss, 'output': output}

def build\_isotonic\_graph(features, label, mode, params, config=None, export\_discretizer=True):

"""

Pre-set Isotonic Calibrator Build Graph

Follows the same signature as build\_graph

This assumes that MLP already contains all modules (include percentile

discretizer); if export\_discretizer is set

then it does not export the MDL phase.

"""

sparse\_tf = twml.util.convert\_to\_sparse(features, params.input\_size\_bits)

if export\_discretizer:

return build\_isotonic\_graph\_from\_inputs(sparse\_tf, features, label, mode, params, config)

discretizer = hub.Module(params.discretizer\_path)

if params.discretizer\_signature is None:

discretizer\_signature = "percentile\_discretizer\_calibrator"

else:

discretizer\_signature = params.discretizer\_signature

input\_sparse = discretizer(sparse\_tf, signature=discretizer\_signature)

return build\_isotonic\_graph\_from\_inputs(input\_sparse, features, label, mode, params, config)