# pylint: disable=arguments-differ, invalid-name

"""

This module contains the ``DataRecordTrainer``.

Unlike the parent ``Trainer`` class, the ``DataRecordTrainer``

is used specifically for processing data records.

It abstracts away a lot of the intricacies of working with DataRecords.

`DataRecord <http://go/datarecord>`\_ is the main piping format for data samples.

The `DataRecordTrainer` assumes training data and production responses and requests

to be organized as the `Thrift prediction service API

A ``DataRecord`` is a Thrift struct that defines how to encode the data:

::

struct DataRecord {

1: optional set<i64> binaryFeatures; // stores BINARY features

2: optional map<i64, double> continuousFeatures; // stores CONTINUOUS features

3: optional map<i64, i64> discreteFeatures; // stores DISCRETE features

4: optional map<i64, string> stringFeatures; // stores STRING features

5: optional map<i64, set<string>> sparseBinaryFeatures; // stores sparse BINARY features

6: optional map<i64, map<string, double>> sparseContinuousFeatures; // sparse CONTINUOUS feature

7: optional map<i64, binary> blobFeatures; // stores features as BLOBs (binary large objects)

8: optional map<i64, tensor.GeneralTensor> tensors; // stores TENSOR features

9: optional map<i64, tensor.SparseTensor> sparseTensors; // stores SPARSE\_TENSOR features

}

A significant portion of Twitter data is hydrated

and then temporarily stored on HDFS as DataRecords.

The files are compressed (.gz or .lzo) partitions of data records.

These form supervised datasets. Each sample captures the relationship

between input and output (cause and effect).

To create your own dataset, please see https://github.com/twitter/elephant-bird.

The default ``DataRecordTrainer.[train,evaluate,learn]()`` reads these datarecords.

The data is a read from multiple ``part-\*.[compression]`` files.

The default behavior of ``DataRecordTrainer`` is to read sparse features from ``DataRecords``.

This is a legacy default piping format at Twitter.

The ``DataRecordTrainer`` is flexible enough for research and yet simple enough

for a new beginner ML practioner.

By means of the feature string to key hashing function,

the ``[train,eval]\_feature\_config`` constructor arguments

control which features can be used as sample labels, sample weights,

or sample features.

Samples ids, and feature keys, feature values and feature weights

can be skipped, included, excluded or used as labels, weights, or features.

This allows you to easily define and control sparse distributions of

named features.

Yet sparse data is difficult to work with. We are currently working to

optimize the sparse operations due to inefficiencies in the gradient descent

and parameter update processes. There are efforts underway

to minimize the footprint of sparse data as it is inefficient to process.

CPUs and GPUs much prefer dense tensor data.

"""

import datetime

import tensorflow.compat.v1 as tf

from twitter.deepbird.io.dal import dal\_to\_hdfs\_path, is\_dal\_path

import twml

from twml.trainers import Trainer

from twml.contrib.feature\_importances.feature\_importances import (

compute\_feature\_importances,

TREE,

write\_feature\_importances\_to\_hdfs,

write\_feature\_importances\_to\_ml\_dash)

from absl import logging

class DataRecordTrainer(Trainer): # pylint: disable=abstract-method

"""

The ``DataRecordTrainer`` implementation is intended to satisfy the most common use cases

at Twitter where only the build\_graph methods needs to be overridden.

For this reason, ``Trainer.[train,eval]\_input\_fn`` methods

assume a DataRecord dataset partitioned into part files stored in compressed (e.g. gzip) format.

For use-cases that differ from this common Twitter use-case,

further Trainer methods can be overridden.

If that still doesn't provide enough flexibility, the user can always

use the tf.estimator.Esimator or tf.session.run directly.

"""

def \_\_init\_\_(

self, name, params,

build\_graph\_fn,

feature\_config=None,

\*\*kwargs):

"""

The DataRecordTrainer constructor builds a

``tf.estimator.Estimator`` and stores it in self.estimator.

For this reason, DataRecordTrainer accepts the same Estimator constructor arguments.

It also accepts additional arguments to facilitate metric evaluation and multi-phase training

(init\_from\_dir, init\_map).

Args:

parent arguments:

See the `Trainer constructor <#twml.trainers.Trainer.\_\_init\_\_>`\_ documentation

for a full list of arguments accepted by the parent class.

name, params, build\_graph\_fn (and other parent class args):

see documentation for twml.Trainer doc.

feature\_config:

An object of type FeatureConfig describing what features to decode.

Defaults to None. But it is needed in the following cases:

- `get\_train\_input\_fn()` / `get\_eval\_input\_fn()` is called without a `parse\_fn`

- `learn()`, `train()`, `eval()`, `calibrate()` are called without providing `\*input\_fn`.

\*\*kwargs:

further kwargs can be specified and passed to the Estimator constructor.

"""

# NOTE: DO NOT MODIFY `params` BEFORE THIS CALL.

super(DataRecordTrainer, self).\_\_init\_\_(

name=name, params=params, build\_graph\_fn=build\_graph\_fn, \*\*kwargs)

self.\_feature\_config = feature\_config

# date range parameters common to both training and evaluation data:

hour\_resolution = self.params.get("hour\_resolution", 1)

data\_threads = self.params.get("data\_threads", 4)

datetime\_format = self.params.get("datetime\_format", "%Y/%m/%d")

# retrieve the desired training dataset files

self.\_train\_files = self.build\_files\_list(

files\_list\_path=self.params.get("train\_files\_list", None),

data\_dir=self.params.get("train\_data\_dir", None),

start\_datetime=self.params.get("train\_start\_datetime", None),

end\_datetime=self.params.get("train\_end\_datetime", None),

datetime\_format=datetime\_format, data\_threads=data\_threads,

hour\_resolution=hour\_resolution, maybe\_save=self.is\_chief(),

overwrite=self.params.get("train\_overwrite\_files\_list", False),

)

# retrieve the desired evaluation dataset files

eval\_name = self.params.get("eval\_name", None)

if eval\_name == "train":

self.\_eval\_files = self.\_train\_files

else:

self.\_eval\_files = self.build\_files\_list(

files\_list\_path=self.params.get("eval\_files\_list", None),

data\_dir=self.params.get("eval\_data\_dir", None),

start\_datetime=self.params.get("eval\_start\_datetime", None),

end\_datetime=self.params.get("eval\_end\_datetime", None),

datetime\_format=datetime\_format, data\_threads=data\_threads,

hour\_resolution=hour\_resolution, maybe\_save=self.is\_chief(),

overwrite=self.params.get("eval\_overwrite\_files\_list", False),

)

if not self.params.get("allow\_train\_eval\_overlap"):

# if there is overlap between train and eval, error out!

if self.\_train\_files and self.\_eval\_files:

overlap\_files = set(self.\_train\_files) & set(self.\_eval\_files)

else:

overlap\_files = set()

if overlap\_files:

raise ValueError("There is an overlap between train and eval files:\n %s" %

(overlap\_files))

@staticmethod

def build\_hdfs\_files\_list(

files\_list\_path, data\_dir,

start\_datetime, end\_datetime, datetime\_format,

data\_threads, hour\_resolution, maybe\_save, overwrite):

if files\_list\_path:

files\_list\_path = twml.util.preprocess\_path(files\_list\_path)

if isinstance(start\_datetime, datetime.datetime):

start\_datetime = start\_datetime.strftime(datetime\_format)

if isinstance(end\_datetime, datetime.datetime):

end\_datetime = end\_datetime.strftime(datetime\_format)

list\_files\_by\_datetime\_args = {

"base\_path": data\_dir,

"start\_datetime": start\_datetime,

"end\_datetime": end\_datetime,

"datetime\_prefix\_format": datetime\_format,

"extension": "lzo",

"parallelism": data\_threads,

"hour\_resolution": hour\_resolution,

"sort": True,

}

# no cache of data file paths, just get the list by scraping the directory

if not files\_list\_path or not tf.io.gfile.exists(files\_list\_path):

# twml.util.list\_files\_by\_datetime returns None if data\_dir is None.

# twml.util.list\_files\_by\_datetime passes through data\_dir if data\_dir is a list

files\_list = twml.util.list\_files\_by\_datetime(\*\*list\_files\_by\_datetime\_args)

else:

# the cached data file paths file exists.

files\_info = twml.util.read\_file(files\_list\_path, decode="json")

# use the cached list if data params match current params,

# or if current params are None

# Not including None checks for datetime\_format and hour\_resolution,

# since those are shared between eval and training.

if (all(param is None for param in [data\_dir, start\_datetime, end\_datetime]) or

(files\_info["data\_dir"] == data\_dir and

files\_info["start\_datetime"] == start\_datetime and

files\_info["end\_datetime"] == end\_datetime and

files\_info["datetime\_format"] == datetime\_format and

files\_info["hour\_resolution"] == hour\_resolution)):

files\_list = files\_info["files"]

elif overwrite:

# current params are not none and don't match saved params

# `overwrite` indicates we should thus update the list

files\_list = twml.util.list\_files\_by\_datetime(\*\*list\_files\_by\_datetime\_args)

else:

# dont update the cached list

raise ValueError("Information in files\_list is inconsistent with provided args.\n"

"Did you intend to overwrite files\_list using "

"--train.overwrite\_files\_list or --eval.overwrite\_files\_list?\n"

"If you instead want to use the paths in files\_list, ensure that "

"data\_dir, start\_datetime, and end\_datetime are None.")

if maybe\_save and files\_list\_path and (overwrite or not tf.io.gfile.exists(files\_list\_path)):

save\_dict = {}

save\_dict["files"] = files\_list

save\_dict["data\_dir"] = data\_dir

save\_dict["start\_datetime"] = start\_datetime

save\_dict["end\_datetime"] = end\_datetime

save\_dict["datetime\_format"] = datetime\_format

save\_dict["hour\_resolution"] = hour\_resolution

twml.util.write\_file(files\_list\_path, save\_dict, encode="json")

return files\_list

@staticmethod

def build\_files\_list(files\_list\_path, data\_dir,

start\_datetime, end\_datetime, datetime\_format,

data\_threads, hour\_resolution, maybe\_save, overwrite):

'''

When specifying DAL datasets, only data\_dir, start\_dateime, and end\_datetime

should be given with the format:

dal://{cluster}/{role}/{dataset\_name}/{env}

'''

if not data\_dir or not is\_dal\_path(data\_dir):

logging.warn(f"Please consider specifying a dal:// dataset rather than passing a physical hdfs path.")

return DataRecordTrainer.build\_hdfs\_files\_list(

files\_list\_path, data\_dir,

start\_datetime, end\_datetime, datetime\_format,

data\_threads, hour\_resolution, maybe\_save, overwrite)

del datetime\_format

del data\_threads

del hour\_resolution

del maybe\_save

del overwrite

return dal\_to\_hdfs\_path(

path=data\_dir,

start\_datetime=start\_datetime,

end\_datetime=end\_datetime,

)

@property

def train\_files(self):

return self.\_train\_files

@property

def eval\_files(self):

return self.\_eval\_files

@staticmethod

def add\_parser\_arguments():

"""

Add common commandline args to parse for the Trainer class.

Typically, the user calls this function and then parses cmd-line arguments

into an argparse.Namespace object which is then passed to the Trainer constructor

via the params argument.

See the `Trainer code <\_modules/twml/trainers/trainer.html#Trainer.add\_parser\_arguments>`\_

and `DataRecordTrainer code

<\_modules/twml/trainers/trainer.html#DataRecordTrainer.add\_parser\_arguments>`\_

for a list and description of all cmd-line arguments.

Args:

learning\_rate\_decay:

Defaults to False. When True, parses learning rate decay arguments.

Returns:

argparse.ArgumentParser instance with some useful args already added.

"""

parser = super(DataRecordTrainer, DataRecordTrainer).add\_parser\_arguments()

parser.add\_argument(

"--train.files\_list", "--train\_files\_list", type=str, default=None,

dest="train\_files\_list",

help="Path for a json file storing information on training data.\n"

"Specifically, the file at files\_list should contain the dataset parameters "

"for constructing the list of data files, and the list of data file paths.\n"

"If the json file does not exist, other args are used to construct the "

"training files list, and that list will be saved to the indicated json file.\n"

"If the json file does exist, and current args are consistent with "

"saved args, or are all None, then the saved files list will be used.\n"

"If current args are not consistent with the saved args, then error out "

"if train\_overwrite\_files\_list==False, else overwrite files\_list with "

"a newly constructed list.")

parser.add\_argument(

"--train.overwrite\_files\_list", "--train\_overwrite\_files\_list", action="store\_true", default=False,

dest="train\_overwrite\_files\_list",

help="When the --train.files\_list param is used, indicates whether to "

"overwrite the existing --train.files\_list when there are differences "

"between the current and saved dataset args. Default (False) is to "

"error out if files\_list exists and differs from current params.")

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, dal://{cluster}-{region}/{role}/{dataset\_name}/{environment}, "

"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(

"--eval.files\_list", "--eval\_files\_list", type=str, default=None,

dest="eval\_files\_list",

help="Path for a json file storing information on evaluation data.\n"

"Specifically, the file at files\_list should contain the dataset parameters "

"for constructing the list of data files, and the list of data file paths.\n"

"If the json file does not exist, other args are used to construct the "

"evaluation files list, and that list will be saved to the indicated json file.\n"

"If the json file does exist, and current args are consistent with "

"saved args, or are all None, then the saved files list will be used.\n"

"If current args are not consistent with the saved args, then error out "

"if eval\_overwrite\_files\_list==False, else overwrite files\_list with "

"a newly constructed list.")

parser.add\_argument(

"--eval.overwrite\_files\_list", "--eval\_overwrite\_files\_list", action="store\_true", default=False,

dest="eval\_overwrite\_files\_list",

help="When the --eval.files\_list param is used, indicates whether to "

"overwrite the existing --eval.files\_list when there are differences "

"between the current and saved dataset args. Default (False) is to "

"error out if files\_list exists and differs from current params.")

parser.add\_argument(

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

dest="eval\_data\_dir",

help="Path to the cross-validation data directory."

"Supports local, dal://{cluster}-{region}/{role}/{dataset\_name}/{environment}, "

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

parser.add\_argument(

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

type=str, default=None,

dest="eval\_start\_datetime",

help="Starting date for evaluating inside the eval data dir."

"The start datetime is inclusive."

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

parser.add\_argument(

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

dest="eval\_end\_datetime",

help="Ending date for evaluating inside the eval 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(

"--data\_spec", type=str, required=True,

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

parser.add\_argument(

"--train.keep\_rate", "--train\_keep\_rate", type=float, default=None,

dest="train\_keep\_rate",

help="A float value in (0.0, 1.0] that indicates to drop records according to the Bernoulli \

distribution with p = 1 - keep\_rate.")

parser.add\_argument(

"--eval.keep\_rate", "--eval\_keep\_rate", type=float, default=None,

dest="eval\_keep\_rate",

help="A float value in (0.0, 1.0] that indicates to drop records according to the Bernoulli \

distribution with p = 1 - keep\_rate.")

parser.add\_argument(

"--train.parts\_downsampling\_rate", "--train\_parts\_downsampling\_rate",

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

type=float, default=None,

help="A float value in (0.0, 1.0] that indicates the factor by which to downsample part \

files. For example, a value of 0.2 means only 20 percent of part files become part of the \

dataset.")

parser.add\_argument(

"--eval.parts\_downsampling\_rate", "--eval\_parts\_downsampling\_rate",

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

type=float, default=None,

help="A float value in (0.0, 1.0] that indicates the factor by which to downsample part \

files. For example, a value of 0.2 means only 20 percent of part files become part of the \

dataset.")

parser.add\_argument(

"--allow\_train\_eval\_overlap",

dest="allow\_train\_eval\_overlap",

action="store\_true",

help="Allow overlap between train and eval datasets."

)

parser.add\_argument(

"--eval\_name", type=str, default=None,

help="String denoting what we want to name the eval. If this is `train`, then we eval on \

the training dataset."

)

return parser

def contrib\_run\_feature\_importances(self, feature\_importances\_parse\_fn=None, write\_to\_hdfs=True, extra\_groups=None, datarecord\_filter\_fn=None, datarecord\_filter\_run\_name=None):

"""Compute feature importances on a trained model (this is a contrib feature)

Args:

feature\_importances\_parse\_fn (fn): The same parse\_fn that we use for training/evaluation.

Defaults to feature\_config.get\_parse\_fn()

write\_to\_hdfs (bool): Setting this to True writes the feature importance metrics to HDFS

extra\_groups (dict<str, list<str>>): A dictionary mapping the name of extra feature groups to the list of

the names of the features in the group

datarecord\_filter\_fn (function): a function takes a single data sample in com.twitter.ml.api.ttypes.DataRecord format

and return a boolean value, to indicate if this data record should be kept in feature importance module or not.

"""

logging.info("Computing feature importance")

algorithm = self.\_params.feature\_importance\_algorithm

kwargs = {}

if algorithm == TREE:

kwargs["split\_feature\_group\_on\_period"] = self.\_params.split\_feature\_group\_on\_period

kwargs["stopping\_metric"] = self.\_params.feature\_importance\_metric

kwargs["sensitivity"] = self.\_params.feature\_importance\_sensitivity

kwargs["dont\_build\_tree"] = self.\_params.dont\_build\_tree

kwargs["extra\_groups"] = extra\_groups

if self.\_params.feature\_importance\_is\_metric\_larger\_the\_better:

# The user has specified that the stopping metric is one where larger values are better (e.g. ROC\_AUC)

kwargs["is\_metric\_larger\_the\_better"] = True

elif self.\_params.feature\_importance\_is\_metric\_smaller\_the\_better:

# The user has specified that the stopping metric is one where smaller values are better (e.g. LOSS)

kwargs["is\_metric\_larger\_the\_better"] = False

else:

# The user has not specified which direction is better for the stopping metric

kwargs["is\_metric\_larger\_the\_better"] = None

logging.info("Using the tree algorithm with kwargs {}".format(kwargs))

feature\_importances = compute\_feature\_importances(

trainer=self,

data\_dir=self.\_params.get('feature\_importance\_data\_dir'),

feature\_config=self.\_feature\_config,

algorithm=algorithm,

record\_count=self.\_params.feature\_importance\_example\_count,

parse\_fn=feature\_importances\_parse\_fn,

datarecord\_filter\_fn=datarecord\_filter\_fn,

\*\*kwargs)

if not feature\_importances:

logging.info("Feature importances returned None")

else:

if write\_to\_hdfs:

logging.info("Writing feature importance to HDFS")

write\_feature\_importances\_to\_hdfs(

trainer=self,

feature\_importances=feature\_importances,

output\_path=datarecord\_filter\_run\_name,

metric=self.\_params.get('feature\_importance\_metric'))

else:

logging.info("Not writing feature importance to HDFS")

logging.info("Writing feature importance to ML Metastore")

write\_feature\_importances\_to\_ml\_dash(

trainer=self, feature\_importances=feature\_importances)

return feature\_importances

def export\_model(self, serving\_input\_receiver\_fn=None,

export\_output\_fn=None,

export\_dir=None, checkpoint\_path=None,

feature\_spec=None):

"""

Export the model for prediction. Typically, the exported model

will later be run in production servers. This method is called

by the user to export the PREDICT graph to disk.

Internally, this method calls `tf.estimator.Estimator.export\_savedmodel

<https://www.tensorflow.org/api\_docs/python/tf/estimator/Estimator#export\_savedmodel>`\_.

Args:

serving\_input\_receiver\_fn (Function):

function preparing the model for inference requests.

If not set; defaults to the the serving input receiver fn set by the FeatureConfig.

export\_output\_fn (Function):

Function to export the graph\_output (output of build\_graph) for

prediction. Takes a graph\_output dict as sole argument and returns

the export\_output\_fns dict.

Defaults to ``twml.export\_output\_fns.batch\_prediction\_continuous\_output\_fn``.

export\_dir:

directory to export a SavedModel for prediction servers.

Defaults to ``[save\_dir]/exported\_models``.

checkpoint\_path:

the checkpoint path to export. If None (the default), the most recent checkpoint

found within the model directory ``save\_dir`` is chosen.

Returns:

The export directory where the PREDICT graph is saved.

"""

if serving\_input\_receiver\_fn is None:

if self.\_feature\_config is None:

raise ValueError("`feature\_config` was not passed to `DataRecordTrainer`")

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

if feature\_spec is None:

if self.\_feature\_config is None:

raise ValueError("feature\_spec can not be inferred."

"Please pass feature\_spec=feature\_config.get\_feature\_spec() to the trainer.export\_model method")

else:

feature\_spec = self.\_feature\_config.get\_feature\_spec()

if isinstance(serving\_input\_receiver\_fn, twml.feature\_config.FeatureConfig):

raise ValueError("Cannot pass FeatureConfig as a parameter to serving\_input\_receiver\_fn")

elif not callable(serving\_input\_receiver\_fn):

raise ValueError("Expecting Function for serving\_input\_receiver\_fn")

if export\_output\_fn is None:

export\_output\_fn = twml.export\_output\_fns.batch\_prediction\_continuous\_output\_fn

return super(DataRecordTrainer, self).export\_model(

export\_dir=export\_dir,

serving\_input\_receiver\_fn=serving\_input\_receiver\_fn,

checkpoint\_path=checkpoint\_path,

export\_output\_fn=export\_output\_fn,

feature\_spec=feature\_spec,

)

def get\_train\_input\_fn(

self, parse\_fn=None, repeat=None, shuffle=True, interleave=True, shuffle\_files=None,

initializable=False, log\_tf\_data\_summaries=False, \*\*kwargs):

"""

This method is used to create input function used by estimator.train().

Args:

parse\_fn:

Function to parse a data record into a set of features.

Defaults to the parser returned by the FeatureConfig selected

repeat (optional):

Specifies if the dataset is to be repeated. Defaults to `params.train\_steps > 0`.

This ensures the training is run for atleast `params.train\_steps`.

Toggling this to `False` results in training finishing when one of the following happens:

- The entire dataset has been trained upon once.

- `params.train\_steps` has been reached.

shuffle (optional):

Specifies if the files and records in the files need to be shuffled.

When `True`, files are shuffled, and records of each files are shuffled.

When `False`, files are read in alpha-numerical order. Also when `False`

the dataset is sharded among workers for Hogwild and distributed training

if no sharding configuration is provided in `params.train\_dataset\_shards`.

Defaults to `True`.

interleave (optional):

Specifies if records from multiple files need to be interleaved in parallel.

Defaults to `True`.

shuffle\_files (optional):

Shuffle the list of files. Defaults to 'Shuffle' if not provided.

initializable (optional):

A boolean indicator. When the parsing function depends on some resource, e.g. a HashTable or

a Tensor, i.e. it's an initializable iterator, set it to True. Otherwise, default value

(false) is used for most plain iterators.

log\_tf\_data\_summaries (optional):

A boolean indicator denoting whether to add a `tf.data.experimental.StatsAggregator` to the

tf.data pipeline. This adds summaries of pipeline utilization and buffer sizes to the output

events files. This requires that `initializable` is `True` above.

Returns:

An input\_fn that can be consumed by `estimator.train()`.

"""

if parse\_fn is None:

if self.\_feature\_config is None:

raise ValueError("`feature\_config` was not passed to `DataRecordTrainer`")

parse\_fn = self.\_feature\_config.get\_parse\_fn()

if not callable(parse\_fn):

raise ValueError("Expecting parse\_fn to be a function.")

if log\_tf\_data\_summaries and not initializable:

raise ValueError("Require `initializable` if `log\_tf\_data\_summaries`.")

if repeat is None:

repeat = self.params.train\_steps > 0 or self.params.get('distributed', False)

if not shuffle and self.num\_workers > 1 and self.params.train\_dataset\_shards is None:

num\_shards = self.num\_workers

shard\_index = self.worker\_index

else:

num\_shards = self.params.train\_dataset\_shards

shard\_index = self.params.train\_dataset\_shard\_index

return lambda: twml.input\_fns.default\_input\_fn(

files=self.\_train\_files,

batch\_size=self.params.train\_batch\_size,

parse\_fn=parse\_fn,

num\_threads=self.params.num\_threads,

repeat=repeat,

keep\_rate=self.params.train\_keep\_rate,

parts\_downsampling\_rate=self.params.train\_parts\_downsampling\_rate,

shards=num\_shards,

shard\_index=shard\_index,

shuffle=shuffle,

shuffle\_files=(shuffle if shuffle\_files is None else shuffle\_files),

interleave=interleave,

initializable=initializable,

log\_tf\_data\_summaries=log\_tf\_data\_summaries,

\*\*kwargs)

def get\_eval\_input\_fn(

self, parse\_fn=None, repeat=None,

shuffle=True, interleave=True,

shuffle\_files=None, initializable=False, log\_tf\_data\_summaries=False, \*\*kwargs):

"""

This method is used to create input function used by estimator.eval().

Args:

parse\_fn:

Function to parse a data record into a set of features.

Defaults to twml.parsers.get\_sparse\_parse\_fn(feature\_config).

repeat (optional):

Specifies if the dataset is to be repeated. Defaults to `params.eval\_steps > 0`.

This ensures the evaluation is run for atleast `params.eval\_steps`.

Toggling this to `False` results in evaluation finishing when one of the following happens:

- The entire dataset has been evaled upon once.

- `params.eval\_steps` has been reached.

shuffle (optional):

Specifies if the files and records in the files need to be shuffled.

When `False`, files are read in alpha-numerical order.

When `True`, files are shuffled, and records of each files are shuffled.

Defaults to `True`.

interleave (optional):

Specifies if records from multiple files need to be interleaved in parallel.

Defaults to `True`.

shuffle\_files (optional):

Shuffles the list of files. Defaults to 'Shuffle' if not provided.

initializable (optional):

A boolean indicator. When the parsing function depends on some resource, e.g. a HashTable or

a Tensor, i.e. it's an initializable iterator, set it to True. Otherwise, default value

(false) is used for most plain iterators.

log\_tf\_data\_summaries (optional):

A boolean indicator denoting whether to add a `tf.data.experimental.StatsAggregator` to the

tf.data pipeline. This adds summaries of pipeline utilization and buffer sizes to the output

events files. This requires that `initializable` is `True` above.

Returns:

An input\_fn that can be consumed by `estimator.eval()`.

"""

if parse\_fn is None:

if self.\_feature\_config is None:

raise ValueError("`feature\_config` was not passed to `DataRecordTrainer`")

parse\_fn = self.\_feature\_config.get\_parse\_fn()

if not self.\_eval\_files:

raise ValueError("`eval\_files` was not present in `params` passed to `DataRecordTrainer`")

if not callable(parse\_fn):

raise ValueError("Expecting parse\_fn to be a function.")

if log\_tf\_data\_summaries and not initializable:

raise ValueError("Require `initializable` if `log\_tf\_data\_summaries`.")

if repeat is None:

repeat = self.params.eval\_steps > 0

return lambda: twml.input\_fns.default\_input\_fn(

files=self.\_eval\_files,

batch\_size=self.params.eval\_batch\_size,

parse\_fn=parse\_fn,

num\_threads=self.params.num\_threads,

repeat=repeat,

keep\_rate=self.params.eval\_keep\_rate,

parts\_downsampling\_rate=self.params.eval\_parts\_downsampling\_rate,

shuffle=shuffle,

shuffle\_files=(shuffle if shuffle\_files is None else shuffle\_files),

interleave=interleave,

initializable=initializable,

log\_tf\_data\_summaries=log\_tf\_data\_summaries,

\*\*kwargs

)

def \_assert\_train\_files(self):

if not self.\_train\_files:

raise ValueError("train.data\_dir was not set in params passed to DataRecordTrainer.")

def \_assert\_eval\_files(self):

if not self.\_eval\_files:

raise ValueError("eval.data\_dir was not set in params passed to DataRecordTrainer.")

def train(self, input\_fn=None, steps=None, hooks=None):

"""

Makes input functions optional. input\_fn defaults to self.get\_train\_input\_fn().

See Trainer for more detailed documentation documentation.

"""

if input\_fn is None:

self.\_assert\_train\_files()

input\_fn = input\_fn if input\_fn else self.get\_train\_input\_fn()

super(DataRecordTrainer, self).train(input\_fn=input\_fn, steps=steps, hooks=hooks)

def evaluate(self, input\_fn=None, steps=None, hooks=None, name=None):

"""

Makes input functions optional. input\_fn defaults to self.get\_eval\_input\_fn().

See Trainer for more detailed documentation.

"""

if input\_fn is None:

self.\_assert\_eval\_files()

input\_fn = input\_fn if input\_fn else self.get\_eval\_input\_fn(repeat=False)

return super(DataRecordTrainer, self).evaluate(

input\_fn=input\_fn,

steps=steps,

hooks=hooks,

name=name

)

def learn(self, train\_input\_fn=None, eval\_input\_fn=None, \*\*kwargs):

"""

Overrides ``Trainer.learn`` to make ``input\_fn`` functions optional.

Respectively, ``train\_input\_fn`` and ``eval\_input\_fn`` default to

``self.train\_input\_fn`` and ``self.eval\_input\_fn``.

See ``Trainer.learn`` for more detailed documentation.

"""

if train\_input\_fn is None:

self.\_assert\_train\_files()

if eval\_input\_fn is None:

self.\_assert\_eval\_files()

train\_input\_fn = train\_input\_fn if train\_input\_fn else self.get\_train\_input\_fn()

eval\_input\_fn = eval\_input\_fn if eval\_input\_fn else self.get\_eval\_input\_fn()

super(DataRecordTrainer, self).learn(

train\_input\_fn=train\_input\_fn,

eval\_input\_fn=eval\_input\_fn,

\*\*kwargs

)

def train\_and\_evaluate(self,

train\_input\_fn=None, eval\_input\_fn=None,

\*\*kwargs):

"""

Overrides ``Trainer.train\_and\_evaluate`` to make ``input\_fn`` functions optional.

Respectively, ``train\_input\_fn`` and ``eval\_input\_fn`` default to

``self.train\_input\_fn`` and ``self.eval\_input\_fn``.

See ``Trainer.train\_and\_evaluate`` for detailed documentation.

"""

if train\_input\_fn is None:

self.\_assert\_train\_files()

if eval\_input\_fn is None:

self.\_assert\_eval\_files()

train\_input\_fn = train\_input\_fn if train\_input\_fn else self.get\_train\_input\_fn()

eval\_input\_fn = eval\_input\_fn if eval\_input\_fn else self.get\_eval\_input\_fn()

super(DataRecordTrainer, self).train\_and\_evaluate(

train\_input\_fn=train\_input\_fn,

eval\_input\_fn=eval\_input\_fn,

\*\*kwargs

)

def \_model\_fn(self, features, labels, mode, params, config=None):

"""

Overrides the \_model\_fn to correct for the features shape of the sparse features

extracted with the contrib.FeatureConfig

"""

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

# Fix the shape of the features. The features dictionary will be modified to

# contain the shape changes.

twml.util.fix\_shape\_sparse(features, self.\_feature\_config)

return super(DataRecordTrainer, self).\_model\_fn(

features=features,

labels=labels,

mode=mode,

params=params,

config=config

)

def calibrate(self,

calibrator,

input\_fn=None,

steps=None,

save\_calibrator=True,

hooks=None):

"""

Makes input functions optional. input\_fn defaults to self.train\_input\_fn.

See Trainer for more detailed documentation.

"""

if input\_fn is None:

self.\_assert\_train\_files()

input\_fn = input\_fn if input\_fn else self.get\_train\_input\_fn()

super(DataRecordTrainer, self).calibrate(calibrator=calibrator,

input\_fn=input\_fn,

steps=steps,

save\_calibrator=save\_calibrator,

hooks=hooks)

def save\_checkpoints\_and\_export\_model(self,

serving\_input\_receiver\_fn,

export\_output\_fn=None,

export\_dir=None,

checkpoint\_path=None,

input\_fn=None):

"""

Exports saved module after saving checkpoint to save\_dir.

Please note that to use this method, you need to assign a loss to the output

of the build\_graph (for the train mode).

See export\_model for more detailed information.

"""

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

self.export\_model(serving\_input\_receiver\_fn, export\_output\_fn, export\_dir, checkpoint\_path)

def save\_checkpoints\_and\_evaluate(self,

input\_fn=None,

steps=None,

hooks=None,

name=None):

"""

Evaluates model after saving checkpoint to save\_dir.

Please note that to use this method, you need to assign a loss to the output

of the build\_graph (for the train mode).

See evaluate for more detailed information.

"""

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

self.evaluate(input\_fn, steps, hooks, name)