"""

This module contains the experiment tracker for tracking training in ML Metastore

"""

from contextlib import contextmanager

from datetime import datetime

import getpass

import hashlib

import os

import re

import sys

import time

from absl import logging

import tensorflow.compat.v1 as tf

from twml.hooks import MetricsUpdateHook

try:

from urllib import quote as encode\_url

except ImportError:

from urllib.parse import quote as encode\_url

try:

# ML Metastore packages might not be available on GCP.

# If they are not found, tracking is disabled

import requests

from com.twitter.mlmetastore.modelrepo.client import ModelRepoClient

from com.twitter.mlmetastore.modelrepo.core.path import (

check\_valid\_id, get\_components\_from\_id, generate\_id)

from com.twitter.mlmetastore.modelrepo.core import (

DeepbirdRun, Experiment, FeatureConfig, FeatureConfigFeature, Model, ProgressReport, Project, StatusUpdate)

except ImportError:

ModelRepoClient = None

class ExperimentTracker(object):

"""

A tracker that records twml runs in ML Metastore.

"""

def \_\_init\_\_(self, params, run\_config, save\_dir):

"""

Args:

params (python dict):

The trainer params. ExperimentTracker uses `params.experiment\_tracking\_path` (String) and

`params.disable\_experiment\_tracking`.

If `experiment\_tracking\_path` is set to None, the tracker tries to guess a path with

save\_dir.

If `disable\_experiment\_tracking` is True, the tracker is disabled.

run\_config (tf.estimator.RunConfig):

The run config used by the estimator.

save\_dir (str):

save\_dir of the trainer

"""

if isinstance(params, dict):

self.\_params = params

else:

# preserving backward compatibility for people still using HParams

logging.warning("Please stop using HParams and use python dicts. HParams are removed in TF 2")

self.\_params = dict((k, v) for k, v in params.values().items() if v != 'null')

self.\_run\_config = run\_config

self.\_graceful\_shutdown\_port = self.\_params.get('health\_port')

self.tracking\_path = self.\_params.get('experiment\_tracking\_path')

is\_tracking\_path\_too\_long = self.tracking\_path is not None and len(self.tracking\_path) > 256

if is\_tracking\_path\_too\_long:

raise ValueError("Experiment Tracking Path longer than 256 characters")

self.disabled = (

self.\_params.get('disable\_experiment\_tracking', False) or

not self.\_is\_env\_eligible\_for\_tracking() or

ModelRepoClient is None

)

self.\_is\_hogwild = bool(os.environ.get('TWML\_HOGWILD\_PORTS'))

self.\_is\_distributed = bool(os.environ.get('TF\_CONFIG'))

self.\_client = None if self.disabled else ModelRepoClient()

run\_name\_from\_environ = self.run\_name\_from\_environ()

run\_name\_can\_be\_inferred = (

self.tracking\_path is not None or run\_name\_from\_environ is not None)

# Turn the flags off as needed in hogwild / distributed

if self.\_is\_hogwild or self.\_is\_distributed:

self.\_env\_eligible\_for\_recording\_experiment = (

self.\_run\_config.task\_type == "evaluator")

if run\_name\_can\_be\_inferred:

self.\_env\_eligible\_for\_recording\_export\_metadata = (

self.\_run\_config.task\_type == "chief")

else:

logging.info(

'experiment\_tracking\_path is not set and can not be inferred. '

'Recording export metadata is disabled because the chief node and eval node '

'are setting different experiment tracking paths.')

self.\_env\_eligible\_for\_recording\_export\_metadata = False

else:

# Defaults to True

self.\_env\_eligible\_for\_recording\_experiment = True

self.\_env\_eligible\_for\_recording\_export\_metadata = True

if not self.disabled:

# Sanitize passed in experiment tracking paths. e.g. own:proJ:exp:Run.Name

# -> own:proj:exp:Run\_Name

if self.tracking\_path:

try:

check\_valid\_id(self.tracking\_path)

except ValueError as err:

logging.error(f'Invalid experiment tracking path provided. Sanitizing: {self.tracking\_path}\nError: {err}')

self.tracking\_path = generate\_id(

owner=self.path['owner'],

project\_name=self.path['project\_name'],

experiment\_name=self.path['experiment\_name'],

run\_name=self.path['run\_name']

)

logging.error(f'Generated sanitized experiment tracking path: {self.tracking\_path}')

else:

logging.info(

'No experiment\_tracking\_path set. Experiment Tracker will try to guess a path')

self.tracking\_path = self.guess\_path(save\_dir, run\_name\_from\_environ)

logging.info('Guessed path: %s', self.tracking\_path)

# additional check to see if generated path is valid

try:

check\_valid\_id(self.tracking\_path)

except ValueError as err:

logging.error(

'Could not generate valid experiment tracking path. Disabling tracking. ' +

'Error:\n{}'.format(err)

)

self.disabled = True

self.project\_id = None if self.disabled else '{}:{}'.format(

self.path['owner'], self.path['project\_name'])

self.base\_run\_id = None if self.disabled else self.tracking\_path

self.\_current\_run\_name\_suffix = None

self.\_current\_tracker\_hook = None

if self.disabled:

logging.info('Experiment Tracker is disabled')

else:

logging.info('Experiment Tracker initialized with base run id: %s', self.base\_run\_id)

@contextmanager

def track\_experiment(self, eval\_hooks, get\_estimator\_spec\_fn, name=None):

"""

A context manager for tracking experiment. It should wrap the training loop.

An experiment tracker eval hook is appended to eval\_hooks to collect metrics.

Args:

eval\_hooks (list):

The list of eval\_hooks to be used. When it's not None, and does not contain any ,

MetricsUpdateHook an experiment tracker eval hook is appended to it. When it contains

any MetricsUpdateHook, this tracker is disabled to avoid conflict with legacy Model Repo

tracker (`TrackRun`).

get\_estimator\_spec\_fn (func):

A function to get the current EstimatorSpec of the trainer, used by the eval hook.

name (str);

Name of this training or evaluation. Used as a suffix of the run\_id.

Returns:

The tracker's eval hook which is appended to eval\_hooks.

"""

# disable this tracker if legacy TrackRun hook is present

# TODO: remove this once we completely deprecate the old TrackRun interface

if eval\_hooks is not None:

self.disabled = self.disabled or any(isinstance(x, MetricsUpdateHook) for x in eval\_hooks)

logging.info('Is environment eligible for recording experiment: %s',

self.\_env\_eligible\_for\_recording\_experiment)

if self.\_env\_eligible\_for\_recording\_experiment and self.\_graceful\_shutdown\_port:

requests.post('http://localhost:{}/track\_training\_start'.format(

self.\_graceful\_shutdown\_port

))

if self.disabled or eval\_hooks is None:

yield None

else:

assert self.\_current\_tracker\_hook is None, 'experiment tracking has been started already'

if name is not None:

self.\_current\_run\_name\_suffix = '\_' + name

logging.info('Starting experiment tracking. Path: %s', self.\_current\_run\_id)

logging.info('Is environment eligible for recording export metadata: %s',

self.\_env\_eligible\_for\_recording\_export\_metadata)

logging.info('This run will be available at: http://go/mldash/experiments/%s',

encode\_url(self.experiment\_id))

try:

self.\_record\_run()

self.\_add\_run\_status(StatusUpdate(self.\_current\_run\_id, status='RUNNING'))

self.\_register\_for\_graceful\_shutdown()

self.\_current\_tracker\_hook = self.create\_eval\_hook(get\_estimator\_spec\_fn)

except Exception as err:

logging.error(

'Failed to record run. This experiment will not be tracked. Error: %s', str(err))

self.\_current\_tracker\_hook = None

if self.\_current\_tracker\_hook is None:

yield None

else:

try:

eval\_hooks.append(self.\_current\_tracker\_hook)

yield self.\_current\_tracker\_hook

except Exception as err:

self.\_add\_run\_status(

StatusUpdate(self.\_current\_run\_id, status='FAILED', description=str(err)))

self.\_deregister\_for\_graceful\_shutdown()

self.\_current\_tracker\_hook = None

self.\_current\_run\_name\_suffix = None

logging.error('Experiment tracking done. Experiment failed.')

raise

try:

if self.\_current\_tracker\_hook.metric\_values:

self.\_record\_update(self.\_current\_tracker\_hook.metric\_values)

self.\_add\_run\_status(StatusUpdate(self.\_current\_run\_id, status='SUCCESS'))

logging.info('Experiment tracking done. Experiment succeeded.')

except Exception as err:

logging.error(

'Failed to update mark run as successful. Error: %s', str(err))

finally:

self.\_deregister\_for\_graceful\_shutdown()

self.\_current\_tracker\_hook = None

self.\_current\_run\_name\_suffix = None

def create\_eval\_hook(self, get\_estimator\_spec\_fn):

"""

Create an eval\_hook to track eval metrics

Args:

get\_estimator\_spec\_fn (func):

A function that returns the current EstimatorSpec of the trainer.

"""

return MetricsUpdateHook(

get\_estimator\_spec\_fn=get\_estimator\_spec\_fn,

add\_metrics\_fn=self.\_record\_update)

def register\_model(self, export\_path):

"""

Record the exported model.

Args:

export\_path (str):

The path to the exported model.

"""

if self.disabled:

return None

try:

logging.info('Model is exported to %s. Computing hash of the model.', export\_path)

model\_hash = self.compute\_model\_hash(export\_path)

logging.info('Model hash: %s. Registering it in ML Metastore.', model\_hash)

self.\_client.register\_model(Model(model\_hash, self.path['owner'], self.base\_run\_id))

except Exception as err:

logging.error('Failed to register model. Error: %s', str(err))

def export\_feature\_spec(self, feature\_spec\_dict):

"""

Export feature spec to ML Metastore (go/ml-metastore).

Please note that the feature list in FeatureConfig only keeps the list of feature hash ids due

to the 1mb upper limit for values in manhattan, and more specific information (feature type,

feature name) for each feature config feature is stored separately in FeatureConfigFeature dataset.

Args:

feature\_spec\_dict (dict): A dictionary obtained from FeatureConfig.get\_feature\_spec()

"""

if self.disabled or not self.\_env\_eligible\_for\_recording\_export\_metadata:

return None

try:

logging.info('Exporting feature spec to ML Metastore.')

feature\_list = feature\_spec\_dict['features']

label\_list = feature\_spec\_dict['labels']

weight\_list = feature\_spec\_dict['weight']

self.\_client.add\_feature\_config(FeatureConfig(self.\_current\_run\_id, list(feature\_list.keys()),

list(label\_list.keys()), list(weight\_list.keys())))

feature\_config\_features = [

FeatureConfigFeature(

hash\_id=\_feature\_hash\_id,

feature\_name=\_feature['featureName'],

feature\_type=\_feature['featureType']

)

for \_feature\_hash\_id, \_feature in zip(feature\_list.keys(), feature\_list.values())

]

self.\_client.add\_feature\_config\_features(list(feature\_list.keys()), feature\_config\_features)

feature\_config\_labels = [

FeatureConfigFeature(

hash\_id=\_label\_hash\_id,

feature\_name=\_label['featureName']

)

for \_label\_hash\_id, \_label in zip(label\_list.keys(), label\_list.values())

]

self.\_client.add\_feature\_config\_features(list(label\_list.keys()), feature\_config\_labels)

feature\_config\_weights = [

FeatureConfigFeature(

hash\_id=\_weight\_hash\_id,

feature\_name=\_weight['featureName'],

feature\_type=\_weight['featureType']

)

for \_weight\_hash\_id, \_weight in zip(weight\_list.keys(), weight\_list.values())

]

self.\_client.add\_feature\_config\_features(list(weight\_list.keys()), feature\_config\_weights)

except Exception as err:

logging.error('Failed to export feature spec. Error: %s', str(err))

@property

def path(self):

if self.disabled:

return None

return get\_components\_from\_id(self.tracking\_path, ensure\_valid\_id=False)

@property

def experiment\_id(self):

if self.disabled:

return None

return '%s:%s:%s' % (self.path['owner'], self.path['project\_name'],

self.path['experiment\_name'])

@property

def \_current\_run\_name(self):

"""

Return the current run name.

"""

if self.\_current\_run\_name\_suffix is not None:

return self.path['run\_name'] + self.\_current\_run\_name\_suffix

else:

return self.path['run\_name']

@property

def \_current\_run\_id(self):

"""

Return the current run id.

"""

if self.\_current\_run\_name\_suffix is not None:

return self.base\_run\_id + self.\_current\_run\_name\_suffix

else:

return self.base\_run\_id

def get\_run\_status(self) -> str:

if not self.disabled:

return self.\_client.get\_latest\_dbv2\_status(self.\_current\_run\_id)

def \_add\_run\_status(self, status):

"""

Add run status with underlying client.

Args:

status (StatusUpdate):

The status update to add.

"""

if not self.disabled and self.\_env\_eligible\_for\_recording\_experiment:

self.\_client.add\_run\_status(status)

def \_record\_run(self):

"""

Record the run in ML Metastore.

"""

if self.disabled or not self.\_env\_eligible\_for\_recording\_experiment:

return None

if not self.\_client.project\_exists(self.project\_id):

self.\_client.add\_project(Project(self.path['project\_name'], self.path['owner']))

time.sleep(1)

if not self.\_client.experiment\_exists(self.experiment\_id):

self.\_client.add\_experiment(Experiment(

self.path['experiment\_name'], self.path['owner'], self.project\_id, ''))

time.sleep(1)

run = DeepbirdRun(self.experiment\_id, self.\_current\_run\_name, '',

{'raw\_command': ' '.join(sys.argv)}, self.\_params)

self.\_client.add\_deepbird\_run(run, force=True)

time.sleep(1)

def \_record\_update(self, metrics):

"""

Record metrics update in ML Metastore.

Args:

metrics (dict):

The dict of the metrics and their values.

"""

if self.disabled or not self.\_env\_eligible\_for\_recording\_experiment:

return None

reported\_metrics = {}

for k, v in metrics.items():

if hasattr(v, 'item'):

reported\_metrics[k] = v.item() if v.size == 1 else str(v.tolist())

else:

logging.warning("Ignoring %s because the value (%s) is not valid" % (k, str(v)))

report = ProgressReport(self.\_current\_run\_id, reported\_metrics)

try:

self.\_client.add\_progress\_report(report)

except Exception as err:

logging.error('Failed to record metrics in ML Metastore. Error: {}'.format(err))

logging.error('Run ID: {}'.format(self.\_current\_run\_id))

logging.error('Progress Report: {}'.format(report.to\_json\_string()))

def \_register\_for\_graceful\_shutdown(self):

"""

Register the tracker with the health server, enabling graceful shutdown.

Returns:

(Response) health server response

"""

if self.\_graceful\_shutdown\_port and not self.disabled and self.\_env\_eligible\_for\_recording\_experiment:

return requests.post('http://localhost:{}/register\_id/{}'.format(

self.\_graceful\_shutdown\_port,

self.\_current\_run\_id

))

def \_deregister\_for\_graceful\_shutdown(self):

"""

Deregister the tracker with the health server, disabling graceful shutdown.

Returns:

(Response) health server response

"""

if self.\_graceful\_shutdown\_port and not self.disabled and self.\_env\_eligible\_for\_recording\_experiment:

return requests.post('http://localhost:{}/deregister\_id/{}'.format(

self.\_graceful\_shutdown\_port,

self.\_current\_run\_id

))

def \_is\_env\_eligible\_for\_tracking(self):

"""

Determine if experiment tracking should run in the env.

"""

is\_unit\_test = (

os.environ.get('PYTEST\_CURRENT\_TEST') is not None and

os.environ.get('TEST\_EXP\_TRACKER') is None

)

is\_running\_on\_ci = (

getpass.getuser() == 'scoot-service' and

os.environ.get('TEST\_EXP\_TRACKER') is None

)

return (

not is\_unit\_test and

not is\_running\_on\_ci

)

@classmethod

def run\_name\_from\_environ(cls):

"""

Create run id from environment if possible.

"""

job\_name = os.environ.get("TWML\_JOB\_NAME")

job\_launch\_time = os.environ.get("TWML\_JOB\_LAUNCH\_TIME")

if not job\_name or not job\_launch\_time:

return None

try:

# job\_launch\_time should be in isoformat

# python2 doesnt support datetime.fromisoformat, so use hardcoded format string.

job\_launch\_time\_formatted = datetime.strptime(job\_launch\_time,

"%Y-%m-%dT%H:%M:%S.%f")

except ValueError:

# Fallback in case aurora config is generating datetime in a different format.

job\_launch\_time\_formatted = (job\_launch\_time

.replace("-", "\_").replace("T", "\_")

.replace(":", "\_").replace(".", "\_"))

return '{}\_{}'.format(

job\_name, job\_launch\_time\_formatted.strftime('%m\_%d\_%Y\_%I\_%M\_%p'))

@classmethod

def guess\_path(cls, save\_dir, run\_name=None):

"""

Guess an experiment tracking path based on save\_dir.

Returns:

(str) guessed path

"""

if not run\_name:

run\_name = 'Unnamed\_{}'.format(datetime.now().strftime('%m\_%d\_%Y\_%I\_%M\_%p'))

if save\_dir.startswith('hdfs://'):

path\_match = re.search(r'/user/([a-z0-9\-\_]+)/([a-z0-9\-\_]+)', save\_dir)

if path\_match:

groups = path\_match.groups()

user = groups[0]

project\_name = groups[1]

return generate\_id(user, 'default', project\_name, run\_name)

user = getpass.getuser()

project\_name = re.sub(r'^[a-z0-9\-\_]', os.path.basename(save\_dir), '')

if not project\_name:

project\_name = 'unnamed'

return generate\_id(user, 'default', project\_name, run\_name)

@classmethod

def compute\_model\_hash(cls, export\_path):

"""

Computes the hash of an exported model. This is a gfile version of

twitter.mlmetastore.common.versioning.compute\_hash. The two functions should generate

the same hash when given the same model.

Args:

export\_path (str):

The path to the exported model.

Returns:

(str) hash of the exported model

"""

paths = []

for path, subdirs, files in tf.io.gfile.walk(export\_path):

for name in sorted(files):

paths.append(os.path.join(path, name))

paths.sort()

hash\_object = hashlib.new('sha1')

for path in paths:

with tf.io.gfile.GFile(path, "rb") as file:

hash\_object.update(file.read())

return hash\_object.hexdigest()