# checkstyle: noqa

import time

from collections import defaultdict

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

from com.twitter.mlmetastore.modelrepo.core import FeatureImportance, FeatureNames

from twitter.deepbird.io.util import match\_feature\_regex\_list

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

\_get\_feature\_name\_from\_config,

\_get\_feature\_types\_from\_records,

\_get\_metrics\_hook,

\_expand\_prefix,

longest\_common\_prefix,

write\_list\_to\_hdfs\_gfile)

from twml.contrib.feature\_importances.feature\_permutation import PermutedInputFnFactory

from twml.tracking import ExperimentTracker

from tensorflow.compat.v1 import logging

from requests.exceptions import HTTPError, RetryError

from queue import Queue

SERIAL = "serial"

TREE = "tree"

INDIVIDUAL = "Individual"

GROUP = "Group"

ROC\_AUC = "roc\_auc"

RCE = "rce"

LOSS = "loss"

def \_repartition(feature\_list\_queue, fnames\_ftypes, split\_feature\_group\_on\_period):

"""

Iterate through letters to partition each feature by prefix, and then put each tuple

(prefix, feature\_partition) into the feature\_list\_queue

Args:

prefix (str): The prefix shared by each feature in list\_of\_feature\_types

feature\_list\_queue (Queue<(str, list<(str, str)>)>): The queue of feature groups

fnames\_ftypes (list<(str, str)>): List of (fname, ftype) pairs. Each fname begins with prefix

split\_feature\_group\_on\_period (str): If true, require that feature groups end in a period

Returns:

Updated queue with each group in fnames\_ftypes

"""

assert len(fnames\_ftypes) > 1

split\_character = "." if split\_feature\_group\_on\_period else None

# Compute the longest prefix of the words

prefix = longest\_common\_prefix(

strings=[fname for fname, \_ in fnames\_ftypes], split\_character=split\_character)

# Separate the features by prefix

prefix\_to\_features = defaultdict(list)

for fname, ftype in fnames\_ftypes:

assert fname.startswith(prefix)

new\_prefix = \_expand\_prefix(fname=fname, prefix=prefix, split\_character=split\_character)

prefix\_to\_features[new\_prefix].append((fname, ftype))

# Add all of the new partitions to the queue

for new\_prefix, fname\_ftype\_list in prefix\_to\_features.items():

extended\_new\_prefix = longest\_common\_prefix(

strings=[fname for fname, \_ in fname\_ftype\_list], split\_character=split\_character)

assert extended\_new\_prefix.startswith(new\_prefix)

feature\_list\_queue.put((extended\_new\_prefix, fname\_ftype\_list))

return feature\_list\_queue

def \_infer\_if\_is\_metric\_larger\_the\_better(stopping\_metric):

# Infers whether a metric should be interpreted such that larger numbers are better (e.g. ROC\_AUC), as opposed to

# larger numbers being worse (e.g. LOSS)

if stopping\_metric is None:

raise ValueError("Error: Stopping Metric cannot be None")

elif stopping\_metric.startswith(LOSS):

logging.info("Interpreting {} to be a metric where larger numbers are worse".format(stopping\_metric))

is\_metric\_larger\_the\_better = False

else:

logging.info("Interpreting {} to be a metric where larger numbers are better".format(stopping\_metric))

is\_metric\_larger\_the\_better = True

return is\_metric\_larger\_the\_better

def \_check\_whether\_tree\_should\_expand(baseline\_performance, computed\_performance, sensitivity, stopping\_metric, is\_metric\_larger\_the\_better):

"""

Returns True if

- the metric is positive (e.g. ROC\_AUC) and computed\_performance is nontrivially smaller than the baseline\_performance

- the metric is negative (e.g. LOSS) and computed\_performance is nontrivially larger than the baseline\_performance

"""

difference = ((baseline\_performance[stopping\_metric] - computed\_performance[stopping\_metric]) /

baseline\_performance[stopping\_metric])

if not is\_metric\_larger\_the\_better:

difference = -difference

logging.info(

"Found a {} difference of {}. Sensitivity is {}.".format("positive" if is\_metric\_larger\_the\_better else "negative", difference, sensitivity))

return difference > sensitivity

def \_compute\_multiple\_permuted\_performances\_from\_trainer(

factory, fname\_ftypes, trainer, parse\_fn, record\_count):

"""Compute performances with fname and fype permuted

"""

metrics\_hook = \_get\_metrics\_hook(trainer)

trainer.\_estimator.evaluate(

input\_fn=factory.get\_permuted\_input\_fn(

batch\_size=trainer.\_params.eval\_batch\_size, parse\_fn=parse\_fn, fname\_ftypes=fname\_ftypes),

steps=(record\_count + trainer.\_params.eval\_batch\_size) // trainer.\_params.eval\_batch\_size,

hooks=[metrics\_hook],

checkpoint\_path=trainer.best\_or\_latest\_checkpoint)

return metrics\_hook.metric\_values

def \_get\_extra\_feature\_group\_performances(factory, trainer, parse\_fn, extra\_groups, feature\_to\_type, record\_count):

"""Compute performance differences for the extra feature groups

"""

extra\_group\_feature\_performance\_results = {}

for group\_name, raw\_feature\_regex\_list in extra\_groups.items():

start = time.time()

fnames = match\_feature\_regex\_list(

features=feature\_to\_type.keys(),

feature\_regex\_list=[regex for regex in raw\_feature\_regex\_list],

preprocess=False,

as\_dict=False)

fnames\_ftypes = [(fname, feature\_to\_type[fname]) for fname in fnames]

logging.info("Extracted extra group {} with features {}".format(group\_name, fnames\_ftypes))

extra\_group\_feature\_performance\_results[group\_name] = \_compute\_multiple\_permuted\_performances\_from\_trainer(

factory=factory, fname\_ftypes=fnames\_ftypes,

trainer=trainer, parse\_fn=parse\_fn, record\_count=record\_count)

logging.info("\n\nImportances computed for {} in {} seconds \n\n".format(

group\_name, int(time.time() - start)))

return extra\_group\_feature\_performance\_results

def \_feature\_importances\_tree\_algorithm(

data\_dir, trainer, parse\_fn, fnames, stopping\_metric, file\_list=None, datarecord\_filter\_fn=None, split\_feature\_group\_on\_period=True,

record\_count=99999, is\_metric\_larger\_the\_better=None, sensitivity=0.025, extra\_groups=None, dont\_build\_tree=False):

"""Tree algorithm for feature and feature group importances. This algorithm build a prefix tree of

the feature names and then traverses the tree with a BFS. At each node (aka group of features with

a shared prefix) the algorithm computes the performance of the model when we permute all features

in the group. The algorithm only zooms-in on groups that impact the performance by more than

sensitivity. As a result, features that affect the model performance by less than sensitivity will

not have an exact importance.

Args:

data\_dir: (str): The location of the training or testing data to compute importances over.

If None, the trainer.\_eval\_files are used

trainer: (DataRecordTrainer): A DataRecordTrainer object

parse\_fn: (function): The parse\_fn used by eval\_input\_fn

fnames (list<string>): The list of feature names

stopping\_metric (str): The metric to use to determine when to stop expanding trees

file\_list (list<str>): The list of filenames. Exactly one of file\_list and data\_dir should be

provided

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.

split\_feature\_group\_on\_period (boolean): If true, split feature groups by period rather than on

optimal prefix

record\_count (int): The number of records to compute importances over

is\_metric\_larger\_the\_better (boolean): If true, assume that stopping\_metric is a metric where larger

values are better (e.g. ROC-AUC)

sensitivity (float): The smallest change in performance to continue to expand the tree

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. You should only supply a value for this argument if you have a set

of features that you want to evaluate as a group but don't share a prefix

dont\_build\_tree (boolean): If True, don't build the tree and only compute the extra\_groups importances

Returns:

A dictionary that contains the individual and group feature importances

"""

factory = PermutedInputFnFactory(

data\_dir=data\_dir, record\_count=record\_count, file\_list=file\_list, datarecord\_filter\_fn=datarecord\_filter\_fn)

baseline\_performance = \_compute\_multiple\_permuted\_performances\_from\_trainer(

factory=factory, fname\_ftypes=[],

trainer=trainer, parse\_fn=parse\_fn, record\_count=record\_count)

out = {"None": baseline\_performance}

if stopping\_metric not in baseline\_performance:

raise ValueError("The stopping metric '{}' not found in baseline\_performance. Metrics are {}".format(

stopping\_metric, list(baseline\_performance.keys())))

is\_metric\_larger\_the\_better = (

is\_metric\_larger\_the\_better if is\_metric\_larger\_the\_better is not None else \_infer\_if\_is\_metric\_larger\_the\_better(stopping\_metric))

logging.info("Using {} as the stopping metric for the tree algorithm".format(stopping\_metric))

feature\_to\_type = \_get\_feature\_types\_from\_records(records=factory.records, fnames=fnames)

all\_feature\_types = list(feature\_to\_type.items())

individual\_feature\_performances = {}

feature\_group\_performances = {}

if dont\_build\_tree:

logging.info("Not building feature importance trie. Will only compute importances for the extra\_groups")

else:

logging.info("Building feature importance trie")

# Each element in the Queue will be a tuple of (prefix, list\_of\_feature\_type\_pairs) where

# each feature in list\_of\_feature\_type\_pairs will have have the prefix "prefix"

feature\_list\_queue = \_repartition(

feature\_list\_queue=Queue(), fnames\_ftypes=all\_feature\_types, split\_feature\_group\_on\_period=split\_feature\_group\_on\_period)

while not feature\_list\_queue.empty():

# Pop the queue. We should never have an empty list in the queue

prefix, fnames\_ftypes = feature\_list\_queue.get()

assert len(fnames\_ftypes) > 0

# Compute performance from permuting all features in fname\_ftypes

logging.info(

"\n\nComputing importances for {} ({}...). {} elements left in the queue \n\n".format(

prefix, fnames\_ftypes[:5], feature\_list\_queue.qsize()))

start = time.time()

computed\_performance = \_compute\_multiple\_permuted\_performances\_from\_trainer(

factory=factory, fname\_ftypes=fnames\_ftypes,

trainer=trainer, parse\_fn=parse\_fn, record\_count=record\_count)

logging.info("\n\nImportances computed for {} in {} seconds \n\n".format(

prefix, int(time.time() - start)))

if len(fnames\_ftypes) == 1:

individual\_feature\_performances[fnames\_ftypes[0][0]] = computed\_performance

else:

feature\_group\_performances[prefix] = computed\_performance

# Dig deeper into the features in fname\_ftypes only if there is more than one feature in the

# list and the performance drop is nontrivial

logging.info("Checking performance for {} ({}...)".format(prefix, fnames\_ftypes[:5]))

check = \_check\_whether\_tree\_should\_expand(

baseline\_performance=baseline\_performance, computed\_performance=computed\_performance,

sensitivity=sensitivity, stopping\_metric=stopping\_metric, is\_metric\_larger\_the\_better=is\_metric\_larger\_the\_better)

if len(fnames\_ftypes) > 1 and check:

logging.info("Expanding {} ({}...)".format(prefix, fnames\_ftypes[:5]))

feature\_list\_queue = \_repartition(

feature\_list\_queue=feature\_list\_queue, fnames\_ftypes=fnames\_ftypes, split\_feature\_group\_on\_period=split\_feature\_group\_on\_period)

else:

logging.info("Not expanding {} ({}...)".format(prefix, fnames\_ftypes[:5]))

# Baseline performance is grouped in with individual\_feature\_importance\_results

individual\_feature\_performance\_results = dict(

out, \*\*{k: v for k, v in individual\_feature\_performances.items()})

group\_feature\_performance\_results = {k: v for k, v in feature\_group\_performances.items()}

if extra\_groups is not None:

logging.info("Computing performances for extra groups {}".format(extra\_groups.keys()))

for group\_name, performances in \_get\_extra\_feature\_group\_performances(

factory=factory,

trainer=trainer,

parse\_fn=parse\_fn,

extra\_groups=extra\_groups,

feature\_to\_type=feature\_to\_type,

record\_count=record\_count).items():

group\_feature\_performance\_results[group\_name] = performances

else:

logging.info("Not computing performances for extra groups")

return {INDIVIDUAL: individual\_feature\_performance\_results,

GROUP: group\_feature\_performance\_results}

def \_feature\_importances\_serial\_algorithm(

data\_dir, trainer, parse\_fn, fnames, file\_list=None, datarecord\_filter\_fn=None, factory=None, record\_count=99999):

"""Serial algorithm for feature importances. This algorithm computes the

importance of each feature.

"""

factory = PermutedInputFnFactory(

data\_dir=data\_dir, record\_count=record\_count, file\_list=file\_list, datarecord\_filter\_fn=datarecord\_filter\_fn)

feature\_to\_type = \_get\_feature\_types\_from\_records(records=factory.records, fnames=fnames)

out = {}

for fname, ftype in list(feature\_to\_type.items()) + [(None, None)]:

logging.info("\n\nComputing importances for {}\n\n".format(fname))

start = time.time()

fname\_ftypes = [(fname, ftype)] if fname is not None else []

out[str(fname)] = \_compute\_multiple\_permuted\_performances\_from\_trainer(

factory=factory, fname\_ftypes=fname\_ftypes,

trainer=trainer, parse\_fn=parse\_fn, record\_count=record\_count)

logging.info("\n\nImportances computed for {} in {} seconds \n\n".format(

fname, int(time.time() - start)))

# The serial algorithm does not compute group feature results.

return {INDIVIDUAL: out, GROUP: {}}

def \_process\_feature\_name\_for\_mldash(feature\_name):

# Using a forward slash in the name causes feature importance writing to fail because strato interprets it as

# part of a url

return feature\_name.replace("/", "\_\_")

def compute\_feature\_importances(

trainer, data\_dir=None, feature\_config=None, algorithm=TREE, parse\_fn=None, datarecord\_filter\_fn=None, \*\*kwargs):

"""Perform a feature importance analysis on a trained model

Args:

trainer: (DataRecordTrainer): A DataRecordTrainer object

data\_dir: (str): The location of the training or testing data to compute importances over.

If None, the trainer.\_eval\_files are used

feature\_config (contrib.FeatureConfig): The feature config object. If this is not provided, it

is taken from the trainer

algorithm (str): The algorithm to use

parse\_fn: (function): The parse\_fn used by eval\_input\_fn. By default this is

feature\_config.get\_parse\_fn()

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.

"""

# We only use the trainer's eval files if an override data\_dir is not provided

if data\_dir is None:

logging.info("Using trainer.\_eval\_files (found {} as files)".format(trainer.\_eval\_files))

file\_list = trainer.\_eval\_files

else:

logging.info("data\_dir provided. Looking at {} for data.".format(data\_dir))

file\_list = None

feature\_config = feature\_config or trainer.\_feature\_config

out = {}

if not feature\_config:

logging.warn("WARN: Not computing feature importance because trainer.\_feature\_config is None")

out = None

else:

parse\_fn = parse\_fn if parse\_fn is not None else feature\_config.get\_parse\_fn()

fnames = \_get\_feature\_name\_from\_config(feature\_config)

logging.info("Computing importances for {}".format(fnames))

logging.info("Using the {} feature importance computation algorithm".format(algorithm))

algorithm = {

SERIAL: \_feature\_importances\_serial\_algorithm,

TREE: \_feature\_importances\_tree\_algorithm}[algorithm]

out = algorithm(data\_dir=data\_dir, trainer=trainer, parse\_fn=parse\_fn, fnames=fnames, file\_list=file\_list, datarecord\_filter\_fn=datarecord\_filter\_fn, \*\*kwargs)

return out

def write\_feature\_importances\_to\_hdfs(

trainer, feature\_importances, output\_path=None, metric="roc\_auc"):

"""Publish a feature importance analysis to hdfs as a tsv

Args:

(see compute\_feature\_importances for other args)

trainer (Trainer)

feature\_importances (dict): Dictionary of feature importances

output\_path (str): The remote or local file to write the feature importances to. If not

provided, this is inferred to be the trainer save dir

metric (str): The metric to write to tsv

"""

# String formatting appends (Individual) or (Group) to feature name depending on type

perfs = {"{} ({})".format(k, importance\_key) if k != "None" else k: v[metric]

for importance\_key, importance\_value in feature\_importances.items()

for k, v in importance\_value.items()}

output\_path = ("{}/feature\_importances-{}".format(

trainer.\_save\_dir[:-1] if trainer.\_save\_dir.endswith('/') else trainer.\_save\_dir,

output\_path if output\_path is not None else str(time.time())))

if len(perfs) > 0:

logging.info("Writing feature\_importances for {} to hdfs".format(perfs.keys()))

entries = [

{

"name": name,

"drop": perfs["None"] - perfs[name],

"pdrop": 100 \* (perfs["None"] - perfs[name]) / (perfs["None"] + 1e-8),

"perf": perfs[name]

} for name in perfs.keys()]

out = ["Name\tPerformance Drop\tPercent Performance Drop\tPerformance"]

for entry in sorted(entries, key=lambda d: d["drop"]):

out.append("{name}\t{drop}\t{pdrop}%\t{perf}".format(\*\*entry))

logging.info("\n".join(out))

write\_list\_to\_hdfs\_gfile(out, output\_path)

logging.info("Wrote feature feature\_importances to {}".format(output\_path))

else:

logging.info("Not writing feature\_importances to hdfs")

return output\_path

def write\_feature\_importances\_to\_ml\_dash(trainer, feature\_importances, feature\_config=None):

# type: (DataRecordTrainer, FeatureConfig, dict) -> None

"""Publish feature importances + all feature names to ML Metastore

Args:

trainer: (DataRecordTrainer): A DataRecordTrainer object

feature\_config (contrib.FeatureConfig): The feature config object. If this is not provided, it

is taken from the trainer

feature\_importances (dict, default=None): Dictionary of precomputed feature importances

feature\_importance\_metric (str, default=None): The metric to write to ML Dashboard

"""

experiment\_tracking\_path = trainer.experiment\_tracker.tracking\_path\

if trainer.experiment\_tracker.tracking\_path\

else ExperimentTracker.guess\_path(trainer.\_save\_dir)

logging.info('Computing feature importances for run: {}'.format(experiment\_tracking\_path))

feature\_importance\_list = []

for key in feature\_importances:

for feature, imps in feature\_importances[key].items():

logging.info('FEATURE NAME: {}'.format(feature))

feature\_name = feature.split(' (').pop(0)

for metric\_name, value in imps.items():

try:

imps[metric\_name] = float(value)

logging.info('Wrote feature importance value {} for metric: {}'.format(str(value), metric\_name))

except Exception as ex:

logging.error("Skipping writing metric:{} to ML Metastore due to invalid metric value: {} or value type: {}. Exception: {}".format(metric\_name, str(value), type(value), str(ex)))

pass

feature\_importance\_list.append(FeatureImportance(

run\_id=experiment\_tracking\_path,

feature\_name=\_process\_feature\_name\_for\_mldash(feature\_name),

feature\_importance\_metrics=imps,

is\_group=key == GROUP

))

# setting feature config to match the one used in compute\_feature\_importances

feature\_config = feature\_config or trainer.\_feature\_config

feature\_names = FeatureNames(

run\_id=experiment\_tracking\_path,

names=list(feature\_config.features.keys())

)

try:

client = ModelRepoClient()

logging.info('Writing feature importances to ML Metastore')

client.add\_feature\_importances(feature\_importance\_list)

logging.info('Writing feature names to ML Metastore')

client.add\_feature\_names(feature\_names)

except (HTTPError, RetryError) as err:

logging.error('Feature importance is not being written due to: '

'HTTPError when attempting to write to ML Metastore: \n{}.'.format(err))