from importlib import import\_module

import os

from toxicity\_ml\_pipeline.settings.default\_settings\_tox import (

INNER\_CV,

LOCAL\_DIR,

MAX\_SEQ\_LENGTH,

NUM\_PREFETCH,

NUM\_WORKERS,

OUTER\_CV,

TARGET\_POS\_PER\_EPOCH,

)

from toxicity\_ml\_pipeline.utils.helpers import execute\_command

import numpy as np

import pandas

from sklearn.model\_selection import StratifiedKFold

import tensorflow as tf

try:

from transformers import AutoTokenizer, DataCollatorWithPadding

except ModuleNotFoundError:

print("...")

else:

from datasets import Dataset

class BalancedMiniBatchLoader(object):

def \_\_init\_\_(

self,

fold,

mb\_size,

seed,

perc\_training\_tox,

scope="TOX",

project=...,

dual\_head=None,

n\_outer\_splits=None,

n\_inner\_splits=None,

sample\_weights=None,

huggingface=False,

):

if 0 >= perc\_training\_tox or perc\_training\_tox > 0.5:

raise ValueError("Perc\_training\_tox should be in ]0; 0.5]")

self.perc\_training\_tox = perc\_training\_tox

if not n\_outer\_splits:

n\_outer\_splits = OUTER\_CV

if isinstance(n\_outer\_splits, int):

self.n\_outer\_splits = n\_outer\_splits

self.get\_outer\_fold = self.\_get\_outer\_cv\_fold

if fold < 0 or fold >= self.n\_outer\_splits or int(fold) != fold:

raise ValueError(f"Number of fold should be an integer in [0 ; {self.n\_outer\_splits} [.")

elif n\_outer\_splits == "time":

self.get\_outer\_fold = self.\_get\_time\_fold

if fold != "time":

raise ValueError(

"To avoid repeating the same run many times, the external fold"

"should be time when test data is split according to dates."

)

try:

setting\_file = import\_module(f"toxicity\_ml\_pipeline.settings.{scope.lower()}{project}\_settings")

except ModuleNotFoundError:

raise ValueError(f"You need to define a setting file for your project {project}.")

self.test\_begin\_date = setting\_file.TEST\_BEGIN\_DATE

self.test\_end\_date = setting\_file.TEST\_END\_DATE

else:

raise ValueError(

f"Argument n\_outer\_splits should either an integer or 'time'. Provided: {n\_outer\_splits}"

)

self.n\_inner\_splits = n\_inner\_splits if n\_inner\_splits is not None else INNER\_CV

self.seed = seed

self.mb\_size = mb\_size

self.fold = fold

self.sample\_weights = sample\_weights

self.dual\_head = dual\_head

self.huggingface = huggingface

if self.huggingface:

self.\_load\_tokenizer()

def \_load\_tokenizer(self):

print("Making a local copy of Bertweet-base model")

local\_model\_dir = os.path.join(LOCAL\_DIR, "models")

cmd = f"mkdir {local\_model\_dir} ; gsutil -m cp -r gs://... {local\_model\_dir}"

execute\_command(cmd)

self.tokenizer = AutoTokenizer.from\_pretrained(

os.path.join(local\_model\_dir, "bertweet-base"), normalization=True

)

def tokenize\_function(self, el):

return self.tokenizer(

el["text"],

max\_length=MAX\_SEQ\_LENGTH,

padding="max\_length",

truncation=True,

add\_special\_tokens=True,

return\_token\_type\_ids=False,

return\_attention\_mask=False,

)

def \_get\_stratified\_kfold(self, n\_splits):

return StratifiedKFold(shuffle=True, n\_splits=n\_splits, random\_state=self.seed)

def \_get\_time\_fold(self, df):

test\_begin\_date = pandas.to\_datetime(self.test\_begin\_date).date()

test\_end\_date = pandas.to\_datetime(self.test\_end\_date).date()

print(f"Test is going from {test\_begin\_date} to {test\_end\_date}.")

test\_data = df.query("@test\_begin\_date <= date <= @test\_end\_date")

query = "date < @test\_begin\_date"

other\_set = df.query(query)

return other\_set, test\_data

def \_get\_outer\_cv\_fold(self, df):

labels = df.int\_label

stratifier = self.\_get\_stratified\_kfold(n\_splits=self.n\_outer\_splits)

k = 0

for train\_index, test\_index in stratifier.split(np.zeros(len(labels)), labels):

if k == self.fold:

break

k += 1

train\_data = df.iloc[train\_index].copy()

test\_data = df.iloc[test\_index].copy()

return train\_data, test\_data

def get\_steps\_per\_epoch(self, nb\_pos\_examples):

return int(max(TARGET\_POS\_PER\_EPOCH, nb\_pos\_examples) / self.mb\_size / self.perc\_training\_tox)

def make\_huggingface\_tensorflow\_ds(self, group, mb\_size=None, shuffle=True):

huggingface\_ds = Dataset.from\_pandas(group).map(self.tokenize\_function, batched=True)

data\_collator = DataCollatorWithPadding(tokenizer=self.tokenizer, return\_tensors="tf")

tensorflow\_ds = huggingface\_ds.to\_tf\_dataset(

columns=["input\_ids"],

label\_cols=["labels"],

shuffle=shuffle,

batch\_size=self.mb\_size if mb\_size is None else mb\_size,

collate\_fn=data\_collator,

)

if shuffle:

return tensorflow\_ds.repeat()

return tensorflow\_ds

def make\_pure\_tensorflow\_ds(self, df, nb\_samples):

buffer\_size = nb\_samples \* 2

if self.sample\_weights is not None:

if self.sample\_weights not in df.columns:

raise ValueError

ds = tf.data.Dataset.from\_tensor\_slices(

(df.text.values, df.label.values, df[self.sample\_weights].values)

)

elif self.dual\_head:

label\_d = {f'{e}\_output': df[f'{e}\_label'].values for e in self.dual\_head}

label\_d['content\_output'] = tf.keras.utils.to\_categorical(label\_d['content\_output'], num\_classes=3)

ds = tf.data.Dataset.from\_tensor\_slices((df.text.values, label\_d))

else:

ds = tf.data.Dataset.from\_tensor\_slices((df.text.values, df.label.values))

ds = ds.shuffle(buffer\_size, seed=self.seed, reshuffle\_each\_iteration=True).repeat()

return ds

def get\_balanced\_dataset(self, training\_data, size\_limit=None, return\_as\_batch=True):

training\_data = training\_data.sample(frac=1, random\_state=self.seed)

nb\_samples = training\_data.shape[0] if not size\_limit else size\_limit

num\_classes = training\_data.int\_label.nunique()

toxic\_class = training\_data.int\_label.max()

if size\_limit:

training\_data = training\_data[: size\_limit \* num\_classes]

print(

".... {} examples, incl. {:.2f}% tox in train, {} classes".format(

nb\_samples,

100 \* training\_data[training\_data.int\_label == toxic\_class].shape[0] / nb\_samples,

num\_classes,

)

)

label\_groups = training\_data.groupby("int\_label")

if self.huggingface:

label\_datasets = {

label: self.make\_huggingface\_tensorflow\_ds(group) for label, group in label\_groups

}

else:

label\_datasets = {

label: self.make\_pure\_tensorflow\_ds(group, nb\_samples=nb\_samples \* 2)

for label, group in label\_groups

}

datasets = [label\_datasets[0], label\_datasets[1]]

weights = [1 - self.perc\_training\_tox, self.perc\_training\_tox]

if num\_classes == 3:

datasets.append(label\_datasets[2])

weights = [1 - self.perc\_training\_tox, self.perc\_training\_tox / 2, self.perc\_training\_tox / 2]

elif num\_classes != 2:

raise ValueError("Currently it should not be possible to get other than 2 or 3 classes")

resampled\_ds = tf.data.experimental.sample\_from\_datasets(datasets, weights, seed=self.seed)

if return\_as\_batch and not self.huggingface:

return resampled\_ds.batch(

self.mb\_size, drop\_remainder=True, num\_parallel\_calls=NUM\_WORKERS, deterministic=True

).prefetch(NUM\_PREFETCH)

return resampled\_ds

@staticmethod

def \_compute\_int\_labels(full\_df):

if full\_df.label.dtype == int:

full\_df["int\_label"] = full\_df.label

elif "int\_label" not in full\_df.columns:

if full\_df.label.max() > 1:

raise ValueError("Binarizing labels that should not be.")

full\_df["int\_label"] = np.where(full\_df.label >= 0.5, 1, 0)

return full\_df

def \_\_call\_\_(self, full\_df, \*args, \*\*kwargs):

full\_df = self.\_compute\_int\_labels(full\_df)

train\_data, test\_data = self.get\_outer\_fold(df=full\_df)

stratifier = self.\_get\_stratified\_kfold(n\_splits=self.n\_inner\_splits)

for train\_index, val\_index in stratifier.split(

np.zeros(train\_data.shape[0]), train\_data.int\_label

):

curr\_train\_data = train\_data.iloc[train\_index]

mini\_batches = self.get\_balanced\_dataset(curr\_train\_data)

steps\_per\_epoch = self.get\_steps\_per\_epoch(

nb\_pos\_examples=curr\_train\_data[curr\_train\_data.int\_label != 0].shape[0]

)

val\_data = train\_data.iloc[val\_index].copy()

yield mini\_batches, steps\_per\_epoch, val\_data, test\_data

def simple\_cv\_load(self, full\_df):

full\_df = self.\_compute\_int\_labels(full\_df)

train\_data, test\_data = self.get\_outer\_fold(df=full\_df)

if test\_data.shape[0] == 0:

test\_data = train\_data.iloc[:500]

mini\_batches = self.get\_balanced\_dataset(train\_data)

steps\_per\_epoch = self.get\_steps\_per\_epoch(

nb\_pos\_examples=train\_data[train\_data.int\_label != 0].shape[0]

)

return mini\_batches, test\_data, steps\_per\_epoch

def no\_cv\_load(self, full\_df):

full\_df = self.\_compute\_int\_labels(full\_df)

val\_test = full\_df[full\_df.origin == "precision"].copy(deep=True)

val\_data, test\_data = self.get\_outer\_fold(df=val\_test)

train\_data = full\_df.drop(full\_df[full\_df.origin == "precision"].index, axis=0)

if test\_data.shape[0] == 0:

test\_data = train\_data.iloc[:500]

mini\_batches = self.get\_balanced\_dataset(train\_data)

if train\_data.int\_label.nunique() == 1:

raise ValueError('Should be at least two labels')

num\_examples = train\_data[train\_data.int\_label == 1].shape[0]

if train\_data.int\_label.nunique() > 2:

second\_most\_frequent\_label = train\_data.loc[train\_data.int\_label != 0, 'int\_label'].mode().values[0]

num\_examples = train\_data[train\_data.int\_label == second\_most\_frequent\_label].shape[0] \* 2

steps\_per\_epoch = self.get\_steps\_per\_epoch(nb\_pos\_examples=num\_examples)

return mini\_batches, steps\_per\_epoch, val\_data, test\_data