#!/usr/bin/env python

# coding: utf-8

# # Software Vulnerability Detection using Deep Learning

#

# ## Pre-processing

# Converting the HDF5 files for training/validation/testing datasets to python pickle for ease of future usage

import h5py

import pandas as pd

# 3 datasets available

data = h5py.File("VDISC\_train.hdf5",'r')

#data = h5py.File("VDISC\_validate.hdf5",'r')

#data = h5py.File("VDISC\_test.hdf5",'r')

# List all groups

data.visit(print)

# Create a new dataframe from the HDF5 file

df = pd.DataFrame(list(data['functionSource']))

df['CWE-119']=list(data['CWE-119']);

df['CWE-120']=list(data['CWE-120']);

df['CWE-469']=list(data['CWE-469']);

df['CWE-476']=list(data['CWE-476']);

df['CWE-other']=list(data['CWE-other'])

df.rename(columns={0:'functionSource'},inplace=True)

df.iloc[0:5,0:]

df.to\_pickle("VDISC\_train.pickle")

#df.to\_pickle("VDISC\_validate.pickle")

#df.to\_pickle("VDISC\_test.pickle")

# ## Exploratory Data Analysis

# ### Importing processed datasets

train=pd.read\_pickle("VDISC\_train.pickle")

validate=pd.read\_pickle("VDISC\_validate.pickle")

test=pd.read\_pickle("VDISC\_test.pickle")

# ## Learning Phase

# ### Importing libraries

import tensorflow as tf

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import sklearn.metrics

import pickle

print("Tensorlfow version: ", tf.\_\_version\_\_)

print("Eager mode: ", tf.executing\_eagerly())

print("GPU is", "available" if tf.test.is\_gpu\_available() else "NOT AVAILABLE")

# ### Setting static and global variables

# Generate random seed

#rand=np.random.randint(1, 99999 + 1)

rand=71926

np.random.seed(rand)

tf.random.set\_seed(rand)

print("Random seed is:",rand)

# Set the global value

WORDS\_SIZE=10000

INPUT\_SIZE=500

NUM\_CLASSES=2

MODEL\_NUM=0

EPOCHS=10

# ### Importing processed datasets

train=pd.read\_pickle("VDISC\_train.pickle")

validate=pd.read\_pickle("VDISC\_validate.pickle")

test=pd.read\_pickle("VDISC\_test.pickle")

for dataset in [train,validate,test]:

for col in range(1,6):

dataset.iloc[:,col] = dataset.iloc[:,col].map({False: 0, True: 1})

# Create source code sdata for tokenization

x\_all = train['functionSource']

#x\_all = x\_all.append(validate['functionSource'])

#x\_all = x\_all.append(test['functionSource'])

# Overview of the datasets

pd.value\_counts(train.iloc[:,1])

one = train[train.iloc[:,1]==1].index.values.astype(int)

zero = train[train.iloc[:,1]==0].index.values.astype(int)

# ### Tokenizing the source codes

# Tokenizer with word-level

tokenizer = tf.keras.preprocessing.text.Tokenizer(char\_level=False)

tokenizer.fit\_on\_texts(list(x\_all))

del(x\_all)

print('Number of tokens: ',len(tokenizer.word\_counts))

# Reducing to top N words

tokenizer.num\_words = WORDS\_SIZE

# Top 10 words

sorted(tokenizer.word\_counts.items(), key=lambda x:x[1], reverse=True)[0:10]

# ### Create sequence files from the tokens

## Tokkenizing train data and create matrix

list\_tokenized\_train = tokenizer.texts\_to\_sequences(train['functionSource'])

x\_train = tf.keras.preprocessing.sequence.pad\_sequences(list\_tokenized\_train,

maxlen=INPUT\_SIZE,

padding='post')

x\_train = x\_train.astype(np.int64)

## Tokkenizing test data and create matrix

list\_tokenized\_test = tokenizer.texts\_to\_sequences(test['functionSource'])

x\_test = tf.keras.preprocessing.sequence.pad\_sequences(list\_tokenized\_test,

maxlen=INPUT\_SIZE,

padding='post')

x\_test = x\_test.astype(np.int64)

## Tokkenizing validate data and create matrix

list\_tokenized\_validate = tokenizer.texts\_to\_sequences(validate['functionSource'])

x\_validate = tf.keras.preprocessing.sequence.pad\_sequences(list\_tokenized\_validate,

maxlen=INPUT\_SIZE,

padding='post')

x\_validate = x\_validate.astype(np.int64)

# Example data

print('Train', pd.value\_counts(train.iloc[:,1]))

print('\nTest', pd.value\_counts(test.iloc[:,1]))

print('\nValidate', pd.value\_counts(validate.iloc[:,1]))

# ### One-Hot-Enconding (OHE) on the datasets

y\_train=[]

y\_test=[]

y\_validate=[]

for col in range(1,6):

y\_train.append(tf.keras.utils.to\_categorical(train.iloc[:,col], num\_classes=NUM\_CLASSES).astype(np.int64))

y\_test.append(tf.keras.utils.to\_categorical(test.iloc[:,col], num\_classes=NUM\_CLASSES).astype(np.int64))

y\_validate.append(tf.keras.utils.to\_categorical(validate.iloc[:,col], num\_classes=NUM\_CLASSES).astype(np.int64))

# Example data

pd.value\_counts(y\_test[0][:,1])

# ### Model Definition (CNN with Gaussian Noise and 1 Output Split)

# Create a random weights matrix

random\_weights = np.random.normal(size=(WORDS\_SIZE, 13),scale=0.01)

# Must use non-sequential model building to create branches in the output layer

model = tf.keras.Sequential(name="CNN")

model.add(tf.keras.layers.Embedding(input\_dim = WORDS\_SIZE,

output\_dim = 13,

weights=[random\_weights],

input\_length = INPUT\_SIZE))

#model.add(tf.keras.layers.GaussianNoise(stddev=0.01))

model.add(tf.keras.layers.Convolution1D(filters=512, kernel\_size=(9), padding='same', activation='relu'))

model.add(tf.keras.layers.MaxPool1D(pool\_size=5))

model.add(tf.keras.layers.Dropout(0.5))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(64, activation='relu'))

model.add(tf.keras.layers.Dense(16, activation='relu'))

model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

# Define custom optimizers

adam = tf.keras.optimizers.Adam(lr=0.005, beta\_1=0.9, beta\_2=0.999, epsilon=1, decay=0.0, amsgrad=False)

## Compile model with metrics

model.compile(optimizer=adam, loss='binary\_crossentropy', metrics=['accuracy'])

print("CNN model built: ")

model.summary()

# ### Tensorboard Callbacks

## Create TensorBoard callbacks

callbackdir= 'D:\\temp\\cb'

tbCallback = tf.keras.callbacks.TensorBoard(log\_dir=callbackdir,

histogram\_freq=1,

embeddings\_freq=1,

write\_graph=True,

write\_images=True)

tbCallback.set\_model(model)

mld = 'model/model-epoch-100-{epoch:02d}-single.hdf5'

## Create best model callback

mcp = tf.keras.callbacks.ModelCheckpoint(filepath=mld,

monitor="val\_loss",

save\_best\_only=True,

mode='auto',

save\_freq='epoch',

verbose=1)

# ### Model Training

#ceiling = 38572

#ceiling =

ceiling = 1019471

class\_weights = {0: 1., 1: 5.}

history = model.fit(x = x\_train[[\*one,\*zero[0:ceiling]],:],

y = train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(),

validation\_data = (x\_validate, validate.iloc[:,1].to\_numpy()),

epochs = 40,

batch\_size = 128,

verbose =2,

class\_weight= class\_weights,

callbacks=[mcp,tbCallback])

with open('history/history-Epoch40-CNN-single', 'wb') as file\_pi:

pickle.dump(history.history, file\_pi)

### Code to Continue Training (Optional)

# Continue training for another 100 epochs

history40 = model.fit(x = x\_train[[\*one,\*zero[0:ceiling]],:],

y = train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(),

validation\_data = (x\_validate, validate.iloc[:,1].to\_numpy()),

epochs = 20,

batch\_size = 128,

verbose =2,

class\_weight= class\_weights,

callbacks=[mcp,tbCallback])

with open('history/history-Epoch40-CNN-single', 'wb') as file\_pi:

pickle.dump(history40.history, file\_pi)

# ### Load the model

# Load model

model = tf.keras.models.load\_model("model/model-epoch-100-36-single.hdf5")

# ### Model Evaluation using Training Set

results = model.evaluate(x\_train[[\*one,\*zero[0:ceiling]],:], train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(), verbose=0, batch\_size=128, )

for num in range(0,len(model.metrics\_names)):

print(model.metrics\_names[num]+': '+str(results[num]))

predicted = model.predict\_classes(x\_train[[\*one,\*zero[0:ceiling]],:])

predicted\_prob = model.predict(x\_train[[\*one,\*zero[0:ceiling]],:])

print('\nConfusion Matrix')

#predicted = model.predict\_classes(x\_test)

confusion = sklearn.metrics.confusion\_matrix(y\_true=train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(), y\_pred=predicted)

print(confusion)

tn, fp, fn, tp = confusion.ravel()

print('\nTP:',tp)

print('FP:',fp)

print('TN:',tn)

print('FN:',fn)

## Performance measure

print('\nAccuracy: '+ str(sklearn.metrics.accuracy\_score(y\_true=train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(), y\_pred=predicted)))

print('Precision: '+ str(sklearn.metrics.precision\_score(y\_true=train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(), y\_pred=predicted)))

print('Recall: '+ str(sklearn.metrics.recall\_score(y\_true=train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(), y\_pred=predicted)))

print('F-measure: '+ str(sklearn.metrics.f1\_score(y\_true=train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(), y\_pred=predicted)))

print('Precision-Recall AUC: '+ str(sklearn.metrics.average\_precision\_score(y\_true=train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(), y\_score=predicted\_prob)))

print('AUC: '+ str(sklearn.metrics.roc\_auc\_score(y\_true=train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(), y\_score=predicted\_prob)))

print('MCC: '+ str(sklearn.metrics.matthews\_corrcoef(y\_true=train.iloc[[\*one,\*zero[0:ceiling]],1].to\_numpy(), y\_pred=predicted)))

# ### Model Evaluation using Testing Set

## Test data

results = model.evaluate(x\_test, test.iloc[:,1].to\_numpy(), batch\_size=128)

for num in range(0,len(model.metrics\_names)):

print(model.metrics\_names[num]+': '+str(results[num]))

predicted = model.predict\_classes(x\_test)

predicted\_prob = model.predict(x\_test)

confusion = sklearn.metrics.confusion\_matrix(y\_true=test.iloc[:,1].to\_numpy(), y\_pred=predicted)

print(confusion)

tn, fp, fn, tp = confusion.ravel()

print('\nTP:',tp)

print('FP:',fp)

print('TN:',tn)

print('FN:',fn)

## Performance measure

print('\nAccuracy: '+ str(sklearn.metrics.accuracy\_score(y\_true=test.iloc[:,1].to\_numpy(), y\_pred=predicted)))

print('Precision: '+ str(sklearn.metrics.precision\_score(y\_true=test.iloc[:,1].to\_numpy(), y\_pred=predicted)))

print('Recall: '+ str(sklearn.metrics.recall\_score(y\_true=test.iloc[:,1].to\_numpy(), y\_pred=predicted)))

print('F-measure: '+ str(sklearn.metrics.f1\_score(y\_true=test.iloc[:,1].to\_numpy(), y\_pred=predicted)))

print('Precision-Recall AUC: '+ str(sklearn.metrics.average\_precision\_score(y\_true=test.iloc[:,1].to\_numpy(), y\_score=predicted\_prob)))

print('AUC: '+ str(sklearn.metrics.roc\_auc\_score(y\_true=test.iloc[:,1].to\_numpy(), y\_score=predicted\_prob)))

print('MCC: '+ str(sklearn.metrics.matthews\_corrcoef(y\_true=test.iloc[:,1].to\_numpy(), y\_pred=predicted)))