Cloud-based PE Malware Detection API

AI & Cybersecurity Midterm Project

DSCI6672 – Spring 2020

4-26-2020

—

Shreya Gopal Sundari

—

Prof. Vahid Behzadan

1. Project Purpose:

The purpose of this project is to deploying machine learning models for malware classification. This project comprises of three tasks. The initial task is to train a deep neural network to classify PE files as malware or benign using Ember opensource dataset, EMBER-2017 v2 available at https://github.com/endgameinc/ember.

The second task deals with deploying the model to cloud and creating an endpoint (~API) to the model. As a final task, create a client nothing but a python script that loads a PE file and classify it as malicious or benign.

2. Requirements:

The requirements of this project are access to Google CoLab and Amazon Sagemaker. Working in these services is advisable for this project.

3. Implementation:

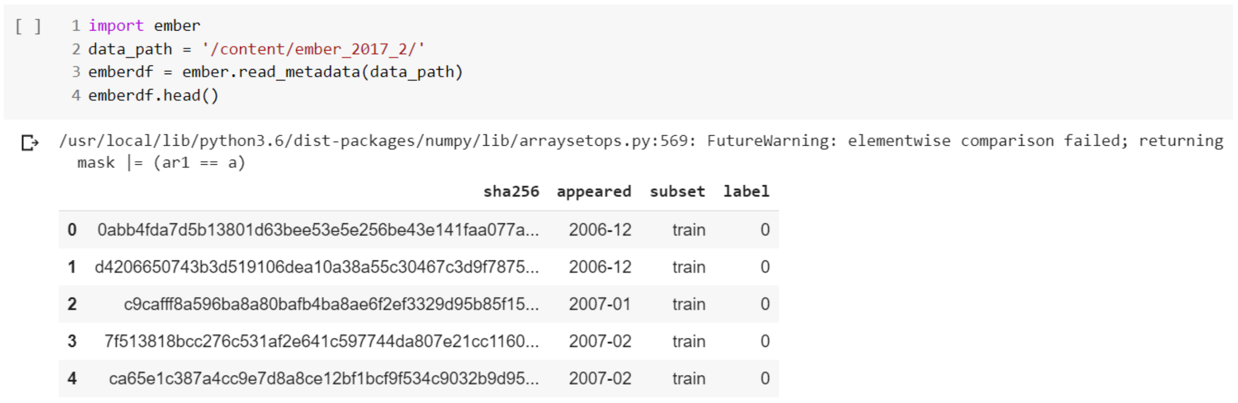
This project is implemented task wise. So, the details in the report are also given based on the tasks performed.

3.1. Task 1-Training:

This task has three parts. They are data extraction & preprocessing, model architecture & training and Testing the model. The parts of this task are detailed below:

3.1.1. Data Extraction & Preprocessing:

Initially, a Jupyter notebook is created in Google Collaboratory to perform the required executions for this task. As an initial part of this task, the data needs to be extracted and vectorized for the neural network training. For this, Ember uses LIEF project library to extract features from PE files included in the EMBER dataset. Raw features are extracted to JSON format. Vectorized features can be produced from these raw features and saved in binary format from which they can be converted to CSV, dataframe, or any other format.



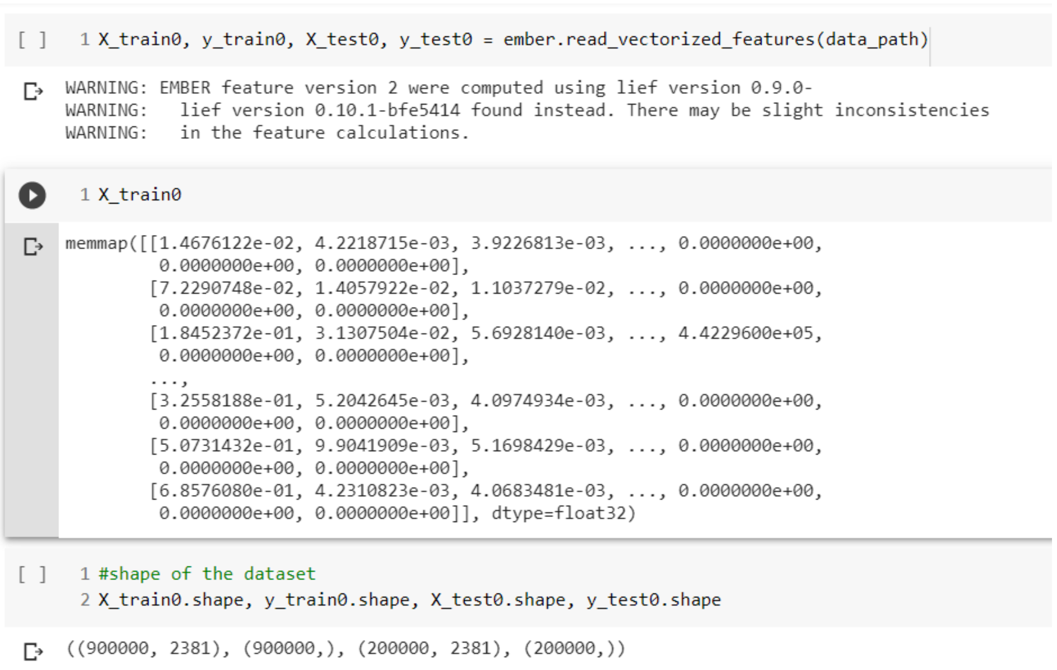
```
[ ] import ember
    data_path = '/content/ember_2017_2/'
    emberdf = ember.read_metadata(data_path)
    emberdf.head()
```

```
/usr/local/lib/python3.6/dist-packages/numpy/lib/arraysetops.py:569: FutureWarning: elementwise comparison failed; returning
  mask |= (ar1 == a)
```

	sha256	appeared	subset	label
0	0abb4fda7d5b13801d63bee53e5e256be43e141faa077a...	2006-12	train	0
1	d4206650743b3d519106dea10a38a55c30467c3d9f7875...	2006-12	train	0
2	c9cafff8a596ba8a80bafb4ba8ae6f2ef3329d95b85f15...	2007-01	train	0
3	7f513818bcc276c531af2e641c597744da807e21cc1160...	2007-02	train	0
4	ca65e1c387a4cc9e7d8a8ce12bf1bcf9f534c9032b9d95...	2007-02	train	0

Fig 1: Reading data from metadata file.

By using Ember's library, the dataset is extracted and vectorized features and is stored in four CSV files, training dataset features, training dataset labels, testing dataset features and testing dataset labels. The train dataset has 900k samples and the test dataset has 200k samples. The number of features in this dataset are 2381 excluding the label or target data. The below image shows the same:



```
[ ] X_train0, y_train0, X_test0, y_test0 = ember.read_vectorized_features(data_path)
```

```
WARNING: EMBER feature version 2 were computed using lief version 0.9.0-
WARNING:   lief version 0.10.1-bfe5414 found instead. There may be slight inconsistencies
WARNING:   in the feature calculations.
```

```
X_train0
```

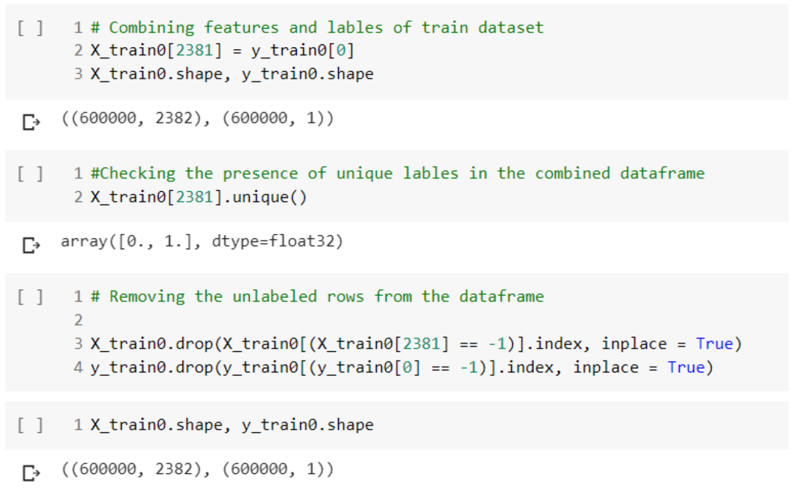
```
memmap([[1.4676122e-02, 4.2218715e-03, 3.9226813e-03, ..., 0.0000000e+00,
         0.0000000e+00, 0.0000000e+00],
        [7.2290748e-02, 1.4057922e-02, 1.1037279e-02, ..., 0.0000000e+00,
         0.0000000e+00, 0.0000000e+00],
        [1.8452372e-01, 3.1307504e-02, 5.6928140e-03, ..., 4.4229600e+05,
         0.0000000e+00, 0.0000000e+00],
        ...,
        [3.2558188e-01, 5.2042645e-03, 4.0974934e-03, ..., 0.0000000e+00,
         0.0000000e+00, 0.0000000e+00],
        [5.0731432e-01, 9.9041909e-03, 5.1698429e-03, ..., 0.0000000e+00,
         0.0000000e+00, 0.0000000e+00],
        [6.8576080e-01, 4.2310823e-03, 4.0683481e-03, ..., 0.0000000e+00,
         0.0000000e+00, 0.0000000e+00]], dtype=float32)
```

```
[ ] #shape of the dataset
    X_train0.shape, y_train0.shape, X_test0.shape, y_test0.shape
```

```
((900000, 2381), (900000,), (200000, 2381), (200000,))
```

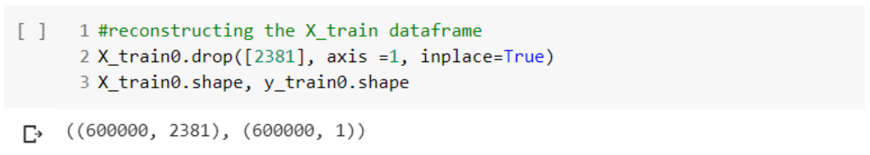
Fig 2: Reading Vectorized features into 4 data files and the shape of the files

It is known that the EMBER train dataset has three sample categories, namely unlabeled, benign and malicious. They are represented as -1, 0 and 1 respectively. The train dataset is equally divided among the three categories i.e., 300k samples of each category. It is the same in test dataset also. But it can be seen that the test dataset has only benign and malicious samples. So, the test dataset has 100k benign samples and 100k malicious sample. In this project, I am ignoring the unlabeled samples from the train dataset for the better performance of the model.



```
[ ]  # Combining features and lables of train dataset
     X_train0[2381] = y_train0[0]
     X_train0.shape, y_train0.shape

     ((600000, 2382), (600000, 1))

[ ]  #Checking the presence of unique lables in the combined dataframe
     X_train0[2381].unique()

     array([0., 1.], dtype=float32)

[ ]  # Removing the unlabeled rows from the dataframe

     X_train0.drop(X_train0[(X_train0[2381] == -1)].index, inplace = True)
     y_train0.drop(y_train0[(y_train0[0] == -1)].index, inplace = True)

[ ]  X_train0.shape, y_train0.shape

     ((600000, 2382), (600000, 1))
```

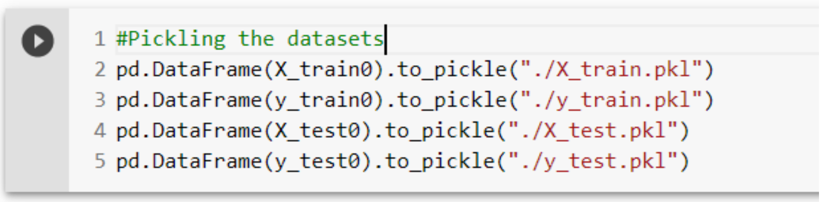
Fig 3: Dropping the unlabeled rows



```
[ ]  #reconstructing the X_train dataframe
     X_train0.drop([2381], axis =1, inplace=True)
     X_train0.shape, y_train0.shape

     ((600000, 2381), (600000, 1))
```

Fig 4: Reconstructing the X_train dataset to its original shape

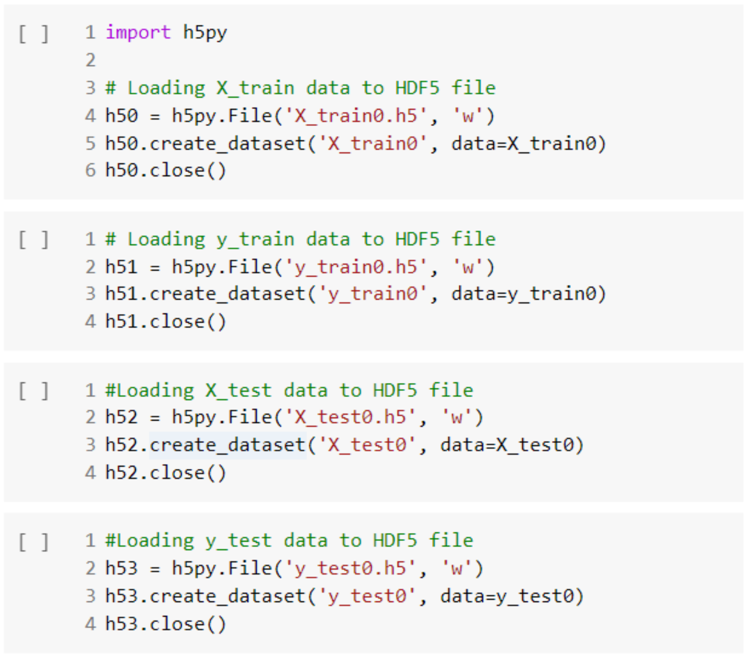
By executing the required code snippets to just load and vectorize the data in Google CoLab notebook, the notebook session kept on crashing. This results in rerunning the entire process again. So, as soon as the data files are loaded, they are pickled to avoid RAM crashes and the long execution time. The pickled data files are then uploaded into Google Drive for easy access to Google CoLab.



```
#Pickling the datasets
pd.DataFrame(X_train0).to_pickle("./X_train.pkl")
pd.DataFrame(y_train0).to_pickle("./y_train.pkl")
pd.DataFrame(X_test0).to_pickle("./X_test.pkl")
pd.DataFrame(y_test0).to_pickle("./y_test.pkl")
```

Fig 5: Serializing the datasets by pickle method

The dataset is huge and so when pickling the data files, can see that this process used most of the 25GB RAM available in CoLab. So, even though the datasets are pickled, the RAM crashes. The alternative for this is to create HDF5 files for all the four dataset files as shown in the image. The h5py package is a Pythonic interface to the HDF5 binary data format.



```
[ ]  import h5py

     # Loading X_train data to HDF5 file
     h50 = h5py.File('X_train0.h5', 'w')
     h50.create_dataset('X_train0', data=X_train0)
     h50.close()
```

```
[ ]  # Loading y_train data to HDF5 file
     h51 = h5py.File('y_train0.h5', 'w')
     h51.create_dataset('y_train0', data=y_train0)
     h51.close()
```

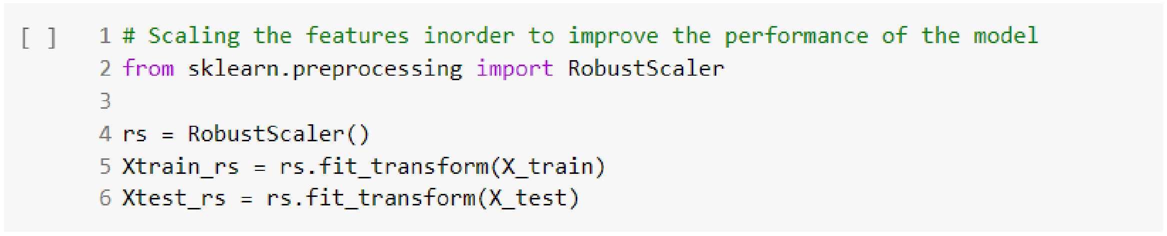
```
[ ]  #Loading X_test data to HDF5 file
     h52 = h5py.File('X_test0.h5', 'w')
     h52.create_dataset('X_test0', data=X_test0)
     h52.close()
```

```
[ ]  #Loading y_test data to HDF5 file
     h53 = h5py.File('y_test0.h5', 'w')
     h53.create_dataset('y_test0', data=y_test0)
     h53.close()
```

Fig 6: Serializing the dataset by using h5py library

The serialized h5 files are uploaded into Google drive or can also be downloaded into local computer for future use.

As a part of preprocessing the vectorized features, scaling of the data is done using Scikit Learn's different scalars like Standard Scalar, Minmax Scalar and Robust Scalar. Out of them, based on my experience, I picked Robust Scalar for the feature scaling of this dataset. Both the training and testing feature dataset are scaled separately using Robust Scalar. Then the scaled data is loaded into the neural network model.



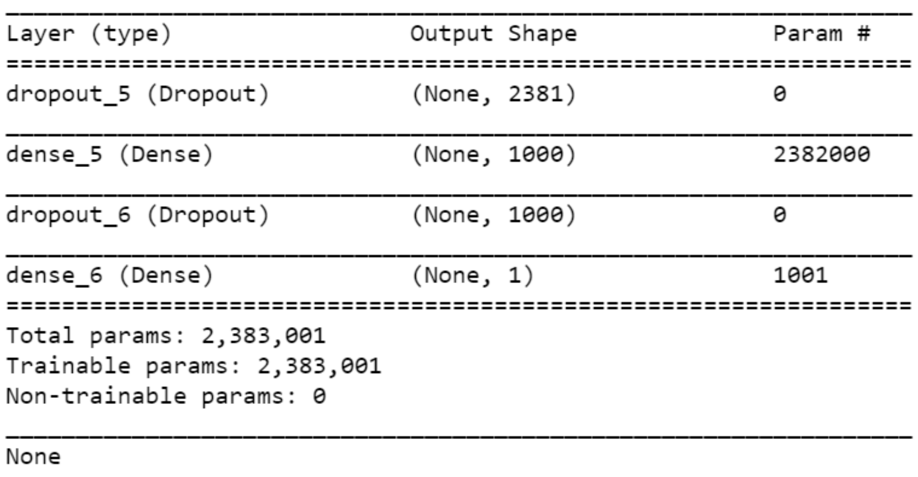
```
[ ]  # Scaling the features inorder to improve the performance of the model
     from sklearn.preprocessing import RobustScaler

     rs = RobustScaler()
     Xtrain_rs = rs.fit_transform(X_train)
     Xtest_rs = rs.fit_transform(X_test)
```

Fig 7: Scaling the features

One can even serialize the above obtained scaled feature data and store it in Google drive or download to local computer for future use.

3.1.2. Model Architecture & Training:

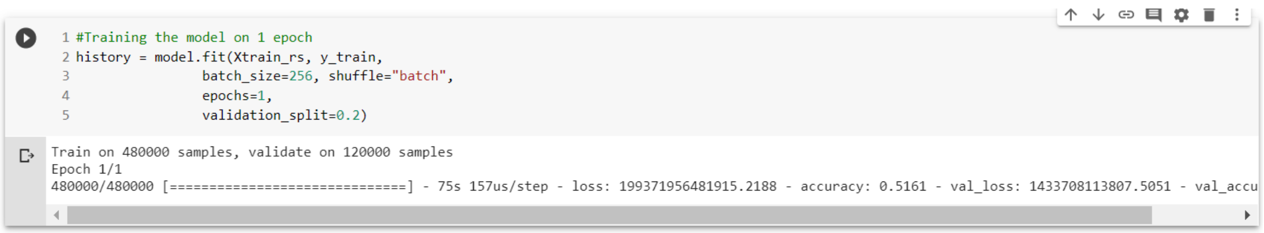
Now comes designing the architecture of the neural network model in Keras. For my model, I chose to build it with a simple and typical dense layers and dropout layers. The dropout layers in the model helps in avoiding overfitting of the model and makes the model more generalized. The number of input nodes to the model is same as the features count in the dataset i.e., 2381. My model architecture is wide instead of deep so, I chose only 2 dense layers and 2 dropout layers. The architecture of the model is shown below:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dropout_5 (Dropout)          (None, 2381)              0         
_________________________________________________________________
dense_5 (Dense)              (None, 1000)              2382000   
_________________________________________________________________
dropout_6 (Dropout)          (None, 1000)              0         
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 1001      
=================================================================
Total params: 2,383,001
Trainable params: 2,383,001
Non-trainable params: 0
_________________________________________________________________
None
```

Fig 8: My Model Architecture

Apart from the shown, the activation functions used are relu for hidden layer and sigmoid for the output layer. The optimizer used while model compilation is Adam, loss function is binary_crossentropy and the performance metric is accuracy. The model training is done in batches of 256 samples. During the training, the train data splits into train and validation data in the ratio of 8:2 i.e., out of 600k samples, 480k samples are training data and remaining 120k samples comprise of validation data. The model performance accuracy obtained at this point is 52%.



```
#Training the model on 1 epoch
history = model.fit(Xtrain_rs, y_train,
                    batch_size=256, shuffle="batch",
                    epochs=1,
                    validation_split=0.2)
```

```
Train on 480000 samples, validate on 120000 samples
Epoch 1/1
480000/480000 [==============================] - 75s 157us/step - loss: 199371956481915.2188 - accuracy: 0.5161 - val_loss: 1433708113807.5051 - val_accu
```

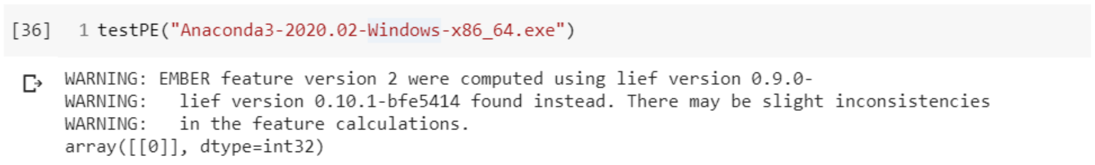
Fig 9: Model trained with 1 epoch

Then the model is trained with 30 epochs which resulted in the fluctuating train accuracy of 52% and validation accuracy of 46%. The hyperparameter choice of the model is made by testing different combination of hyperparameters and finalized them based on the better model performance accuracy.

3.1.3. Model Testing:

The model is tested on the test data and the obtained model performance accuracy is 44%. The trained mode is saved and uploaded to Google Drive for future use. As a part of testing the model, create a function that takes the PE files as an argument, runs it through the trained model and returns the nature of the PE files, Malware (1) or Benign (0).

To do this testing, I downloaded Anaconda PE file using wget command and passed it to the function created to test the file. The result is as shown below:



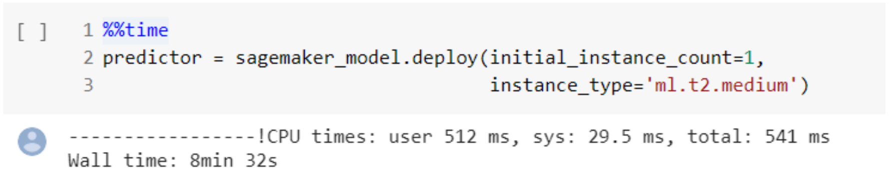
```
[36]  testPE("Anaconda3-2020.02-Windows-x86_64.exe")

WARNING: EMBER feature version 2 were computed using lief version 0.9.0-
WARNING:   lief version 0.10.1-bfe5414 found instead. There may be slight inconsistencies
WARNING:   in the feature calculations.
array([[0]], dtype=int32)
```

Fig 10: Post-training PE file Model testing

Note: The detailed code for this task is in the file "AISec_Task 1_Model Building.ipynb"

3.2. Task 2-Deploy model on the cloud:

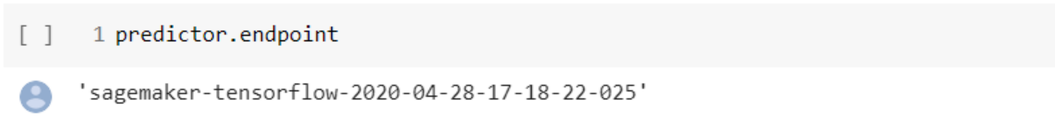
To deploy the model to cloud (AWS), create a notebook instance in AWS Sagemaker and create a notebook where all the executions are done. Then import the required libraries for the creation of the endpoint for the model. Upload the saved model and model weights to the notebook instance. The creation of endpoint took nearly 9 min.



```
[ ]  %%time
     predictor = sagemaker_model.deploy(initial_instance_count=1,
                                        instance_type='ml.t2.medium')

-----------------!CPU times: user 512 ms, sys: 29.5 ms, total: 541 ms
Wall time: 8min 32s
```

Fig 11. Endpoint creation

The endpoint name needs to be noted to use it further. The endpoint name can be obtained as shown:



```
[ ]  predictor.endpoint

'sagemaker-tensorflow-2020-04-28-17-18-22-025'
```

Fig 12: Endpoint name

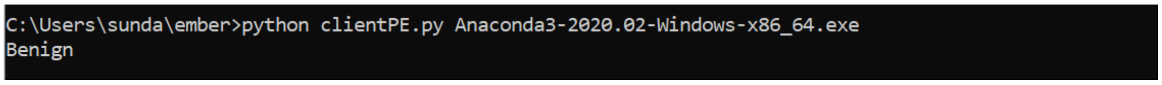
This concludes the task 2 of this project.

Note: The detailed code for this task is in the file "AISec_Task 2_Model_Deployment.ipynb"

3.3. Task 3-Create a client:

In this task, a python code is created that takes PF file as argument and returns the nature of the file. All the within operations requires to reach the result are done in cloud API created in the previous task. Connection to the AWS Sagemaker API can be done by using boto3 library and specifying required keys and token ids of the AWS CLI.

The PE file is parsed and features are extracted from it using the Ember's feature extractor class and the data is dumped into the endpoint using the obtained endpoint from the previous set. Before executing this file, one should install all the requirements needed for the execution of Ember libraries. Then download any PE file, I choose Anaconda's PE file and execute the python file as shown below:

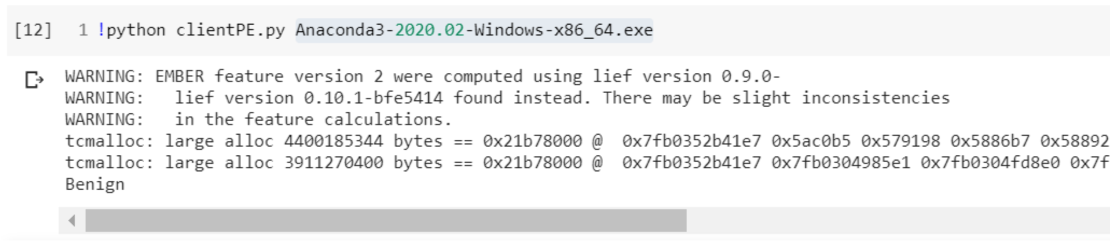


```
C:\Users\sunda\ember>python clientPE.py Anaconda3-2020.02-Windows-x86_64.exe
Benign
```

Fig 13: Client Execution

Note: The detailed code for this task is in the file "clientPE.py"

While I was installing the required libraries and packages in my local computer, I faces lot of issues and error. If that is the case, one can execute this even in Jupyter Notebook. Open a new notebook, install all the required libraries and PE file and then execute the following shown command in the notebook:



```
[12]  !python clientPE.py Anaconda3-2020.02-Windows-x86_64.exe

WARNING: EMBER feature version 2 were computed using lief version 0.9.0-
WARNING:   lief version 0.10.1-bfe5414 found instead. There may be slight inconsistencies
WARNING:   in the feature calculations.
tcmalloc: large alloc 4400185344 bytes == 0x21b78000 @  0x7fb0352b41e7 0x5ac0b5 0x579198 0x5886b7 0x58892
tcmalloc: large alloc 3911270400 bytes == 0x21b78000 @  0x7fb0352b41e7 0x7fb0304985e1 0x7fb0304fd8e0 0x7f
Benign
```

Fig 13: Client Execution in Notebook

Note: The detailed code for the above is in the file "AISec_Task 3_Client Execution.ipynb"

4. Conclusion:

The building of this project is entirely based on the versions of the libraries and packaged and even during their installations. During this project, I learnt about so many services and packages. Overall experience of building this project is knowledgeable and informative.