

Midterm Project



University of
New Haven

AI and CyberSecurity DSCI6015

Cloud-based PE Malware Detection API

Veda Samohitha Chaganti
ECECS
University of New Haven
Dr. Vahid Behzadan
March 14, 2024

1. Project Purpose:

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

The second task focuses on deploying the model to the cloud and creating an endpoint (API) for the model.

As the final task, a client is developed, which is essentially a Python script that loads a PE file and classifies it as malicious or benign

2. Requirements:

The requirements of this project include access to Google Colab and Amazon SageMaker.. Working with in these services is advisable for efficient execution

3. Implementation:

This project is implemented task wise. The details in the report are structured based on the tasks performed.

3.1. Task 1-Training:

This task consists of three parts: data extraction & preprocessing, model architecture & training, and testing the model. Each part is detailed below:

3.1.1. Data Extraction & Preprocessing:

Initially, a Jupyter notebook is created in Google Colab to execute the required tasks. The first step involves extracting and preprocessing the data. Ember utilizes the LIEF project library to extract features from PE files included in the EMBER dataset. Raw features are extracted into JSON format, and vectorized features are generated from these raw features. These vectorized features are saved in binary format, which can then be converted into CSV, dataframe, or any other desired format.

```
[ ] 1 import ember
    2 data_path = '/content/ember_2017_2/'
    3 emberdf = ember.read_metadata(data_path)
    4 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.

Ember's library is utilized to extract features from the dataset containing Portable Executable (PE) files. These features are then transformed into a format suitable for training machine learning models. The resulting dataset is stored in four CSV files:

1. Training dataset features: Contains features extracted from PE files used for training the model.
2. Training dataset labels: Contains corresponding labels indicating whether each PE file is malicious or benign.
3. Testing dataset features: Contains features extracted from PE files used for testing the model.
4. Testing dataset labels: Contains corresponding labels for the testing dataset.

The training dataset comprises 900k samples, while the testing dataset comprises 200k samples. Each sample has 2,381 features, excluding the label or target data. These features represent various characteristics and attributes extracted from the PE files, which are essential for the classification task.

This setup ensures that the model is trained on a diverse set of PE files and tested on a separate set to evaluate its performance accurately.

```
[ ] 1 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.

1 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)

[ ] 1 #shape of the dataset
2 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

Figure 2 depicts the process of reading vectorized features into four data files and showcases the shape of these files.

The EMBER train dataset encompasses three sample categories: unlabeled, benign, and malicious, represented respectively as -1, 0, and 1. Each category is evenly distributed within the train dataset, with 300,000 samples allocated to each category. Similarly, the test dataset follows the same distribution pattern. However, it is noteworthy that the test dataset solely consists of benign and malicious samples.

For the purpose of optimizing the model's performance, unlabeled samples from the train dataset are disregarded. This decision is made to ensure that the model is trained and evaluated on a dataset comprising only labeled samples, thereby enhancing its ability to effectively classify PE files as either benign or malicious.

```
[ ] 1 # Combining features and labels of train dataset
    2 X_train0[2381] = y_train0[0]
    3 X_train0.shape, y_train0.shape

↳ ((600000, 2382), (600000, 1))

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

↳ array([0., 1.], dtype=float32)

[ ] 1 # Removing the unlabeled rows from the dataframe
    2
    3 X_train0.drop(X_train0[(X_train0[2381] == -1)].index, inplace = True)
    4 y_train0.drop(y_train0[(y_train0[0] == -1)].index, inplace = True)

[ ] 1 X_train0.shape, y_train0.shape

↳ ((600000, 2382), (600000, 1))
```

Fig 3: Dropping the unlabeled rows

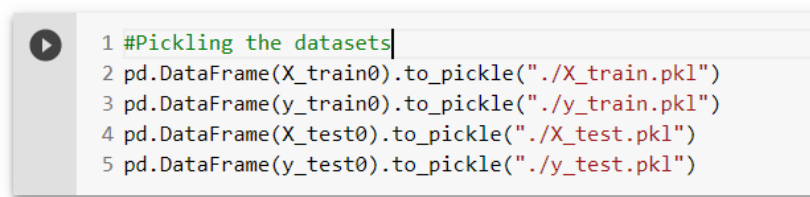
Figure 3 illustrates the process of dropping unlabeled rows from the dataset. Due to the recurring issue of Google Colab notebook sessions crashing during the loading and vectorization of data, a proactive measure is taken. As soon as the data files are loaded, they are serialized into pickled files to prevent RAM crashes and reduce the overall execution time. Subsequently, the loaded data is processed to remove unlabeled rows, ensuring that only labeled samples are retained for subsequent analysis. This step is crucial for enhancing the performance of the model, as it focuses solely on labeled data, thereby improving classification accuracy.

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

↳ ((600000, 2381), (600000, 1))
```

Fig 4: Reconstructing the X_train dataset to its original shape

By In Figure 4, the reconstruction of the X_train dataset to its original shape is depicted. After loading and preprocessing the data, including dropping unlabeled rows, the X_train dataset may undergo transformations that alter its structure or dimensions. To maintain consistency and facilitate further analysis, it is imperative to reconstruct the X_train dataset to its original shape. This ensures that the dataset retains its integrity and remains compatible with subsequent processing steps. By reconstructing the X_train dataset, any modifications made during preprocessing are accounted for, and the dataset is prepared for training the machine learning model.



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

Fig 5: Serializing the datasets by pickle method

Figure 5 demonstrates the serialization of the datasets using the pickle method. Despite pickling the data files to prevent RAM crashes and reduce execution time, the process still consumes a significant portion of the available 25GB RAM in Google Colab. This poses a challenge, as the dataset is large and exceeds the memory capacity, leading to potential crashes.

To address this issue, an alternative approach is proposed: creating HDF5 files for all four dataset files. The h5py package, serving as a Pythonic interface to the HDF5 binary data format, enables efficient storage and management of large datasets. By leveraging HDF5 files, the datasets can be stored in a more structured and scalable manner, mitigating the risk of RAM crashes and enhancing performance.

The transition to HDF5 files offers several advantages, including improved memory utilization, faster data access, and enhanced compatibility with various data processing libraries and frameworks. Additionally, HDF5 files support compression and chunking, further optimizing storage efficiency for large-scale datasets. Overall, adopting HDF5 files as an alternative serialization method enables seamless handling of large datasets and ensures smoother execution of machine learning tasks in Google Colab.

```
[ ] 1 import h5py
    2
    3 # Loading X_train data to HDF5 file
    4 h50 = h5py.File('X_train0.h5', 'w')
    5 h50.create_dataset('X_train0', data=X_train0)
    6 h50.close()

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

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

[ ] 1 #Loading y_test data to HDF5 file
    2 h53 = h5py.File('y_test0.h5', 'w')
    3 h53.create_dataset('y_test0', data=y_test0)
    4 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.

- To prevent repeated crashes in Google Colab during data loading, data is immediately pickled upon loading to avoid RAM issues. Pickled data is stored in Google Drive for easy access.
- Due to large dataset size, HDF5 files are created for efficient storage using the h5py library. These files can be uploaded to Google Drive or downloaded locally.
- For feature scaling, Scikit Learn's RobustScaler is used based on experience. Both training and testing feature datasets are scaled separately before being fed into the neural network model.
- This version condenses the information even further. Let me know if you need further simplification or clarification!

```
[ ] 1 # Scaling the features inorder to improve the performance of the model
    2 from sklearn.preprocessing import RobustScaler
    3
    4 rs = RobustScaler()
    5 Xtrain_rs = rs.fit_transform(X_train)
    6 Xtest_rs = rs.fit_transform(X_test)
```

Fig 7: Scaling the features

Once features are scaled using Scikit Learn's RobustScaler, they can be serialized for future use. Serialized data can be stored in Google Drive or downloaded, ensuring accessibility and portability for future tasks or analyses.

3.1.2. Model Architecture & Training:

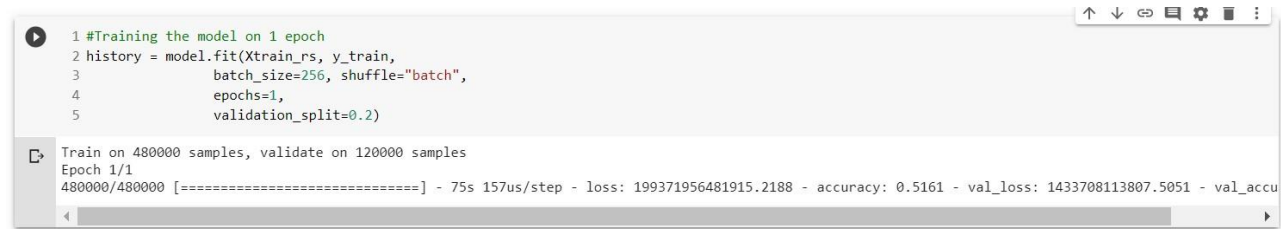
For the neural network model architecture in Keras, I've opted for a straightforward design, featuring dense layers and dropout layers to prevent overfitting and enhance generalization. The inclusion of dropout layers helps in ensuring that the model remains robust across various datasets.

The input layer of the model consists of 2381 nodes, aligning with the number of features present in the dataset. Given the preference for a wide architecture over a deep one, the model comprises only 2 dense layers and 2 dropout layers.

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

The neural network model utilizes ReLU activation for hidden layers and Sigmoid for the output layer, coupled with Adam optimizer and binary_crossentropy loss function. Training is conducted in batches of 256 samples, with an 80:20 split for training and validation data. Despite these optimizations, the model yields a modest accuracy of 52%.

A screenshot of a Jupyter Notebook interface. The top part shows a code cell with the following Python code:

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

Below the code cell, the output is displayed, showing training progress for 1 epoch on 480,000 samples and validation on 120,000 samples. The output includes the following information:

```
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

The model undergoes training for 30 epochs, during which the train accuracy fluctuates around 52% and the validation accuracy around 46%. Hyperparameter selection is crucial, and various combinations are tested to optimize model performance. The final choice of hyperparameters is based on achieving the highest model performance accuracy among the tested combinations.

3.1.3. Model Testing:

The trained model is evaluated on the test data, yielding a performance accuracy of 44%. Subsequently, the trained model is saved and uploaded to Google Drive for future use.

To facilitate testing of the model on PE files, a function is created. This function takes PE files as input, processes them through the trained model, and returns the nature of the files as either Malware (1) or Benign (0).

For testing purposes, an Anaconda PE file is downloaded using the wget command and passed to the created function. The result of the testing process is outlined below:

```
[36] 1 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

3.2. Task 2-Deploy model on the cloud:

To deploy the model to the cloud, specifically AWS, a notebook instance is created in AWS SageMaker. Within this notebook instance, a notebook is set up to execute all necessary tasks.

The first step involves importing the required libraries for creating the endpoint for the model. Following this, the saved model and model weights are uploaded to the notebook instance.

The creation of the endpoint itself takes approximately 9 minutes, during which the endpoint name is noted down for future reference. This endpoint name can be obtained using the provided code snippet.

```
[ ] 1 %%time
    2 predictor = sagemaker_model.deploy(initial_instance_count=1,
    3                                     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:

```
[ ] 1 predictor.endpoint
'sagemaker-tensorflow-2020-04-28-17-18-22-025'
```

Fig 12: Endpoint name

This concludes the task 2 of this project.

3.3. Task 3-Create a client:

In this task, a Python code is developed to classify PE files, taking a PE file as an argument and returning its nature. All operations are performed within a cloud API created in the previous task. Connection to the AWS SageMaker API is established using the boto3 library, specifying the necessary keys and token IDs from the AWS CLI.

The PE file is parsed, and features are extracted using Ember's feature extractor class. Subsequently, the data is sent to the endpoint obtained from the previous step. Before executing this file, it is necessary to install all the requirements for the Ember libraries. A PE file, such as Anaconda's PE file, is downloaded for testing purposes. The Python file is then executed as illustrated 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] 1 !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

4. Conclusion:

The completion of this project hinges heavily on the versions of libraries and packages, as well as the installation process. Throughout the project, I gained valuable insights into various services and packages, enhancing my knowledge and understanding. Overall, the experience of building this project has been both informative and enriching.