Cloud-based PE Malware Detection API

AI and Cybersecurity - Spring 2024

Jafar Vohra

Project Purpose:

The purpose of this project is to deploy a machine-learning model for malware classification. This project is comprised of three tasks. The initial task is to train a deep neural network to classify PE files as malware or benign using the Ember open-source dataset, EMBER-2017 v2. The second task deals with deploying the model to the cloud and creating an endpoint, or an API, for the model. The final task is to create a web application using Streamlit where the application user can upload a PE file that is then classified as malicious or benign.

Requirements:

One must use Google Colab, Amazon Sagemaker, and Streamlit to complete this project. Google Colab is utilized for the training of the model. Amazon Sagemaker Notebook and Endpoint is leveraged for the model deployment. Streamlit is employed for the Client creation and user interaction portion of the project. Each of these tools is essential to the success of the project. Their use will be described in detail in this report.

Training Implementation:

To complete the model training portion of the project, a notebook is created in Google Colab. The training and testing data as well as the metadata is acquired from an Amazon S3 bucket and copied to Google Drive to be accessed. All of the code from the Ember repository is

installed before using one of the methods from there to read the vectorized features and metadata and save them to usable objects in the notebook. Rows in the training dataset with no labels are then discarded. At this point, the preprocessing steps are nearly complete.

To effectively work on the model, train more quickly, and improve model performance, a subset of the entire dataset is used. A 5% sample of the original 600,000 training data records and 200,000 test data records is taken via the Pandas DataFrame sample() method, ensuring the class weights are maintained. This sampled data is not used in the final model, however, it is essential to the process of completing the project. Once this is complete, the datasets are scaled via StandardScaler from scikit-learn. The data is then reshaped to be easily consumed by the model before being tensorized and split into training and validation sets. A batch size of 64 is set before the model is then set to be trained.

The MalConv Neural Network model is built using PyTorch. It starts with an embedding layer that utilizes 8 channels over the 256 unique bytes. It is followed by a permutation of the shape of the data to be consumed by the convolution layers of the model. There are three convolution layers implemented in the model with a dropout rate of 25% used between each after passing through the ReLU activation function. After the convolution and dropout layers have been passed through, a global max pooling phase is completed. Next, two fully connected layers are implemented and another two rounds of dropout are completed between each layer before data passes through the final sigmoid activation function to create the classifications.

This model is then trained with the Binary Cross-Entropy Loss Function and the Adam Optimizer. Trained for 15 total Epochs, the model is passed forward with zeroed parameter gradients to get the outputs and loss value before being passed backward and enforcing a step in

the optimizer. Training and validation loss is printed for each Epoch while checkpoints for the model are saved at every 5 epochs.

Finally, the model is evaluated on the separate test dataset after also being tensorized and loaded. After the model is set to evaluation mode, the test data is passed through the model. During this process, logit values are converted to probabilities via the sigmoid function before predictions and labels are made and flattened to be evaluated by the functions to compute the evaluation metrics.

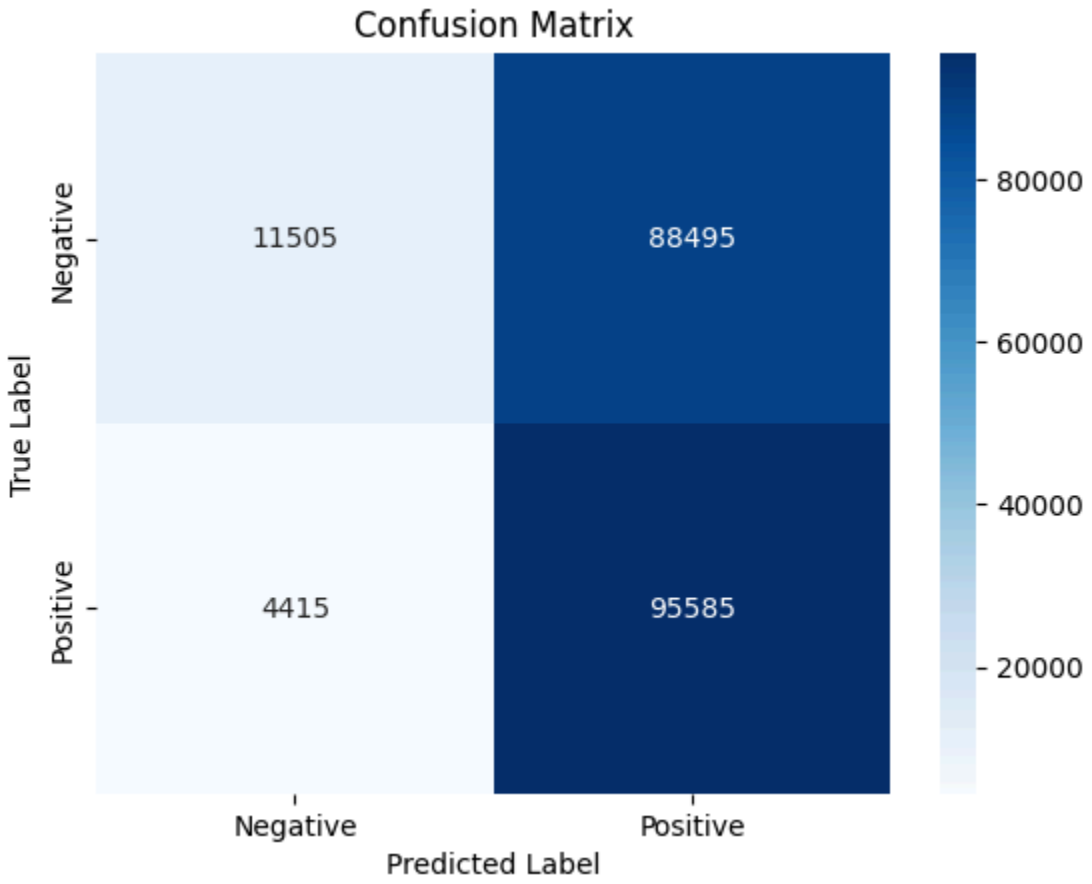
Performance Analysis:

The performance metrics provided indicate several important aspects of the model's performance, but they also raise some concerns. The test accuracy represents the proportion of correctly classified samples out of the total test set. In this case, the accuracy is approximately 53.54%, which means the model is slightly better than random guessing. However, this accuracy level might not be satisfactory, especially in the case of malware detection.

Precision measures the proportion of true positive predictions out of all positive predictions made by the model. A precision rate of 51.93% suggests that just over half of the samples predicted as positive are truly positive. This indicates that the model's ability to avoid false positives is relatively low.

Recall, also known as sensitivity, measures the proportion of true positive predictions out of all actual positive samples. A recall rate of 95.58% indicates that the model is capturing almost all positive samples in the dataset. While high recall is desirable, such a high value, especially when combined with lower precision, could indicate potential issues, such as the model being overly sensitive and possibly prone to false positives.

In addition to the performance metrics discussed, a confusion matrix was also able to be created to see the breakdown of the Type I and II errors. It can be seen below:



This confusion matrix depicts was was previously discussed. The model was successful at classifying PE files as malicious, however is overly safe and predicts that many of the files that pass through it are as such. Many of the errors occur from predicting a file is malicious when it is truly benign. Very few errors occur where the model predicts the file is benign when it is truly malicious, which is a strong quality of the model.

Overall, the model's performance has a significant trade-off between precision and recall, where it tends to classify many samples as malicious but struggles with precision, leading to a high false positive rate. This imbalance might be due to class imbalance in the dataset or model complexity. Further investigation into the model's behavior on different subsets of the data,

analysis of misclassifications, and potential adjustments to the model architecture or training process could help improve its performance. Additionally, considering alternative evaluation metrics like the F1-score, which combines precision and recall, might provide a more comprehensive understanding of the model's performance. With more time and resources allocated to this portion of the project, the model's performance can be significantly improved.

Deployment Implementation:

To complete the deployment portion of the project, it is necessary to import the libraries that will be used throughout the process. Once complete, a compressed archive file, or a ".tar.gz" file is created with the file of the completed model in the root directory. It must be placed in the root directory as the Amazon Sagemaker endpoint will not be able to find the model otherwise. After that, the compressed archive file with the model is placed in an S3 bucket in AWS. It is important to note that the notebook for this portion of the project was created as a Sagemaker Notebook, thus the AWS Academy Lab credentials did not need to be added to the code. Once the file is added to the default S3 bucket, the PyTorch model is created with the data and the entry point of the inference.py file that was created to complement the model deployment.

The entry point Python script begins by redefining the neural network that was created in the training portion of the project. Both the initial and forward propagation functions were included in the class that houses the model. Following the MalConv model class, the PyTorch model is loaded and set to evaluation mode in the model_fn method. Once the data is deserialized and transformed into a tensor in the input_fn method, it is passed into the model and a value for its prediction is returned for the user of the sagemaker endpoint to visualize in the predict_fn and output_fn methods, respectively. The entry point script is directly referenced in

the model deployment as the Sagemaker endpoint is created. It takes approximately three to four minutes for the Sagemaker endpoint to be created, but once complete, it is able to be viewed in the AWS console and invoked by a client.

Client Implementation:

Now that the endpoint is created and in service in Amazon Sagemaker, it is time to reference it through a remote client. After creating functions to extract the features from the original bytes with imported Ember methods and formatting the bytes into a tensor, the endpoint is ready to be invoked. Using the boto3 library, a Sagemaker runtime is able to be started with the credentials from the AWS Academy Lab to access the endpoint. The body of the response from the endpoint are passed back to the client before being deserialized and formatted into the final prediction classification of malicious or benign. In the example shown in the code, the PE file for a Windows calculator application, calc.exe, was passed into the model and was correctly predicted to be benign.

After the remote client was created, an attempt was made to create a Streamlit application to improve the user experience with the model. The Python script follows the same steps as the client notebook and has additional code for the user interface on the Streamlit app. Unfortunately, there were limited resources available for the app and it was unable to be deployed, but the best attempt at working Python script is included in the GitHub repository.

Conclusion:

Overall, this project proved to be an extremely difficult one. With many challenges faced in the model training portion of the project, the second and third tasks were unable to be

completed by the original deadline. With an extension to the deadline, the final two tasks were able to be completed and uploaded to a GitHub repository to be viewed and replicated. The project did prove extremely beneficial to the learning and growth process. Debugging numerous issues in each task and meeting with the professor to express difficulties and work through mistakes, both simple and complex, allowed for a great amount of overall improvement as both a coder and a communicator. Many lessons, such as time and resource allocation and attention to detail were able to be taken away from the outcome of this project.

If more time was allotted towards completion of the tasks in this project, the goal would be to improve the performance of the model in the first task as well as improve the user interface for the interaction with the model in the third task. These improvements would not only level up the project, but also provide even more learning opportunites for the student and experience in areas that are not currently strong.

Bibliography:

Much of the credit is owed to Dr. Behzadan and his GitHub repository that assisted in the second and third tasks of the project. Without his code as well as his time and efforts in person, much of the progress would not be possible. Thank you so much and I appreciate your support!

https://github.com/UNHSAILLab/S24-AISec/tree/main/Midterm%20Tutorial