Malware Detection from Network Captures

Kevin Hannay

Table of contents

1 Introduction **1**
1.1 Assumptions 2

2 Feature Engineering **2**

3 Model Selection **4**

4 ML Ops Pipeline **9**

5 Conclusions **10**

1 Introduction

This report summrarizes the work done in creating a classification model for malware detection from network captures (PCAP) files. The project is organized into a set of jupyter notebooks located in the `nbs` directory, with the majority of the logic located in the `00_core.ipynb` notebook. Additionally, using the nbdev library, the code in the notebooks is exported to a python package located in the `mdetect` directory.

From the home directory of the project the python package can be installed using the following command:

```
#| eval: false
pip install -e .
```

The python package also defines three command line scripts which can be used to run the project from the command line. These scripts are defined in the `01_cli.ipynb` notebook and are as follows:

```
mtrain --help
```

```
mtransform --help
```

```
mpredict --help
```

These allow for the project to move through the ML Ops pipeline from the command line, and also provide some wrapper code for getting started quickly with the models.

1.1 Assumptions

For the purposes of this project I made the following assumptions about the customer need:

- This model would be deployed on networks that aren't in the training data, i.e. we don't have pcap files which can be used to build a baseline for benign traffic for the specific network. If this data is available then it could be used to improve the model performance.
- The cost of missing malware is higher than the cost of mistakenly classifying benign traffic as malware generated. However, this difference isn't extreme since mistakenly flagging benign traffic as malware will degrade trust in the ML pipeline and potentially annoy users.
- The ML pipeline should be efficient enough to run at scale.

2 Feature Engineering

The first step in the project is to transform the raw PCAP data into a set of features which can be used for classification. For this I made use of the scapy python library and created some wrappers and data structures to process and store the data in the raw PCAP files. My strategy for processing the PCAP files was to focus on TCP/UDP traffic and seperate the files into seperate flows or sessions. These flows are defined by unique 5-tuples of (src_ip, dst_ip, src_port, dst_port, protocol), and then these flows were further broken down along the time dimension by splitting up packets within a flow along the time axis. The logic behind this step was the need for this model to generalize to new PCAP files which may have more or less benign traffic overlayed on the malicious traffic. From examining the PCAP files in wireshark it looks like some of the malware files have been filtered already to highlight the malware traffic. The conversion from PCAP to flows should help reduce the impact of benign traffic on the model, and make the model more portable to new networks.

My overall technique focused on extracting features which should generalize well when applied to new network captures. Therefore, I tried to avoid using features that might be specific to the particular network used in the capture. For example, while I developed some features

based on the Inner Arrival times of the packets I avoided using these features in the final model as they would be specific to the network used in the capture. However, in the event that we have access to data collected on a single network, these features could be used to improve the model.

In the end I ended up constructing a set of 23 features which I used in the final model. These features are as follows:

```
from IPython.display import Markdown
from tabulate import tabulate
feature_names = ['duration', 'pkts_rate', 'bytes_rate', 'mean_size', 'std_sizes',
       'q1_sizes', 'q2_sizes', 'q3_sizes', 'min_sizes', 'max_sizes',
       'num_pkts', 'num_bytes', 'flags_FIN', 'flags_SYN', 'flags_RST', 'flags_PSH', 'flags
       'flags_URG', 'flags_ECE', 'flags_CWR', 'src_port', 'dst_port', 'protocol',]
feature_type = ['numeric' for _ in range(20)]  + ['categorical' for _ in range(3)]
descriptions = ["Duration of the flow", "Rate of packet flow", "Rate of bytes transmitted"
feature_table = list(zip(feature_names, feature_type, descriptions))

Markdown(tabulate(
  feature_table,
  headers=["Name","Type", "Description"]
))
```

Table 1: The features used in the classification algorithms

Name	Type	Description
duration	numeric	Duration of the flow
pkts_rate	numeric	Rate of packet flow
bytes_rate	numeric	Rate of bytes transmitted
mean_size	numeric	Mean size of the packets
std_sizes	numeric	Standard Devation in Packet size
q1_sizes	numeric	First quartile in sizes
q2_sizes	numeric	second quartile (median) in sizes
q3_sizes	numeric	Third quartile in sizes
min_sizes	numeric	minimum size
max_sizes	numeric	maximum size
num_pkts	numeric	Total number of packets
num_bytes	numeric	Total bytes
flags_FIN	numeric	Total FIN flags
flags_SYN	numeric	Total SYN flags
flags_RST	numeric	Total RST flags
flags_PSH	numeric	total PSH flags

Name	Type	Description
flags_ACK	numeric	total ACH flags
flags_URG	numeric	total ACK flags
flags_ECE	numeric	Total URG flags
flags_CWR	numeric	Total ECE flags
src_port	categorical	total CWR flags
dst_port	categorical	Source port
protocol	categorical	Destimnation ports

The PCAP files were processed into a set of these flows where each flow was labeled as being from a malicious capture (1) or a benign capture (0). I supplemented the benign PCAP files with two additional PCAP files since the benign data set provided has significantly fewer flows that the aggregate of the malware samples (these supplemental data sets are included in the data directory of the repo).

The data was divided into a training and validation set randomly with 30% being allocated to validation.

3 Model Selection

I tested a set of machine learning pipelines on this processed data set, as can be seen in the `00_core.ipynb`. The confusion matrices for the model pipelines applied to the validation data flows are shown in figure Figure 1. The random forest and XGBoost classifiers perform the best achieving accuracies in the high 90's.

The confusion matrices error rates can be adjusted by adjusting the thresholds used to make a binary classification. To account for this ambiguity we can examine the ROC curves which show the true positive rate (TPR) vs the false positive rate (FPR) for the different models. The ROC curves for the different models are shown in figure Figure 2. The area under these curves is the AUC metric which is a measure of the overall accuracy of the model.

We can also examine the model calibration which is a measure of how well the model probabilities match the true probabilities. The calibration curves for the different models are shown in figure Figure 3. The calibration curves show that each of the models are reasonably well calibrated.

I applied a 5-fold cross validation procedure for each of the model pipelines as well. The results of this cross validation are shown in figure Figure 4. The cross validation results show that the random forest and XGBoost classifiers perform the best overall, and each of these models have a small degree of variance in their performance. This indicates they haven't been overfit to the training data.

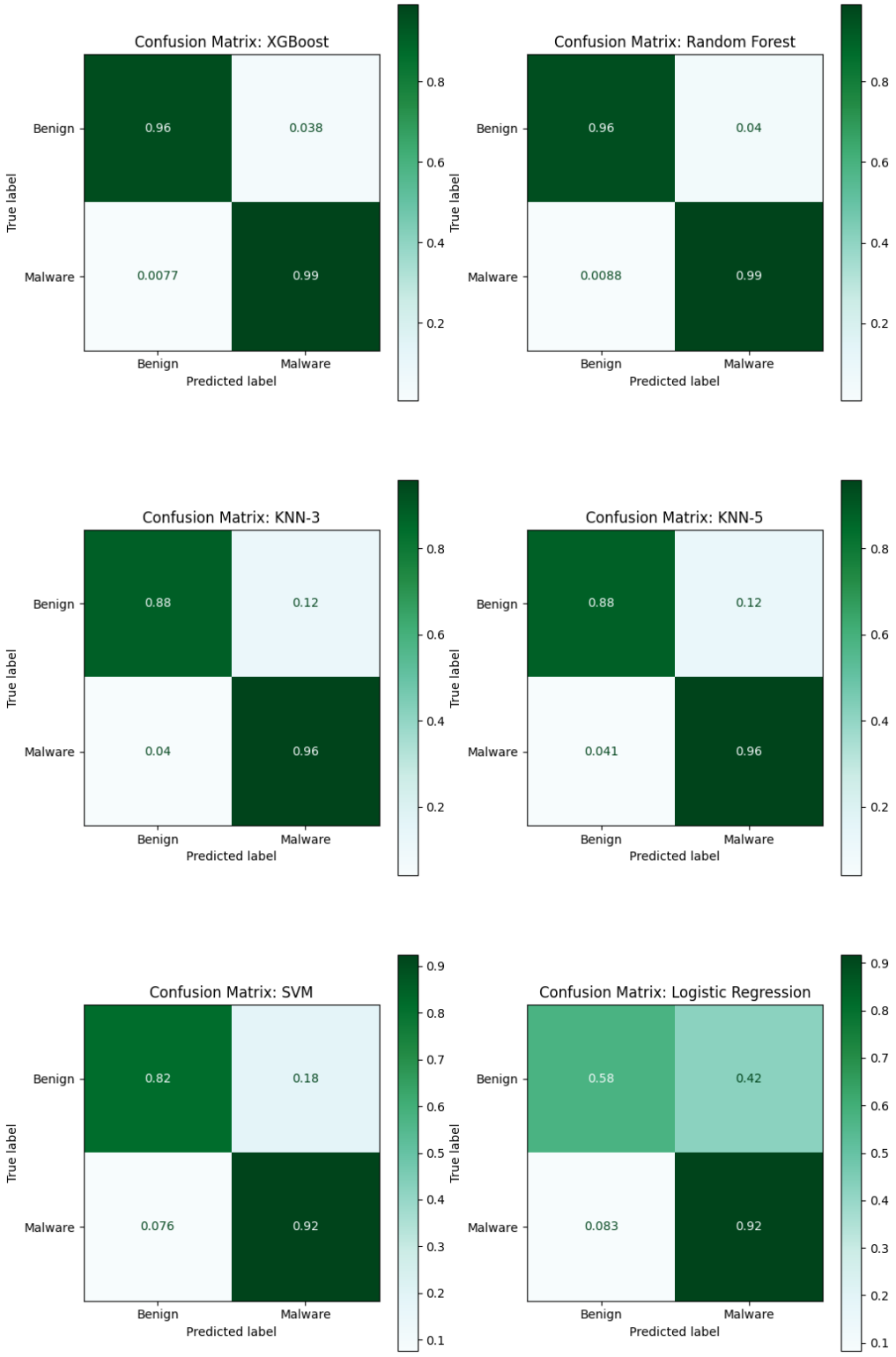

Figure 1: Confusion Matrices: For ML pipelines showing the classification accuracy by true label

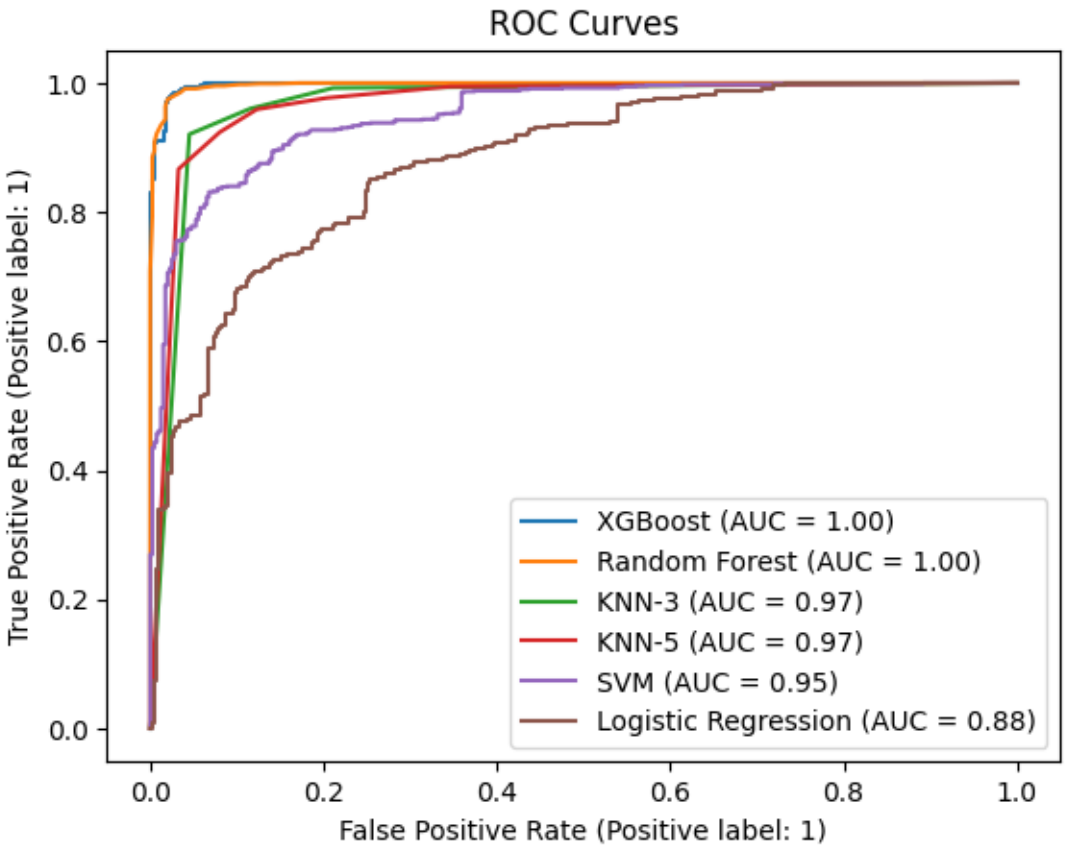

Figure 2: ROC Curves: For ML pipelines showing the classification accuracy by true label

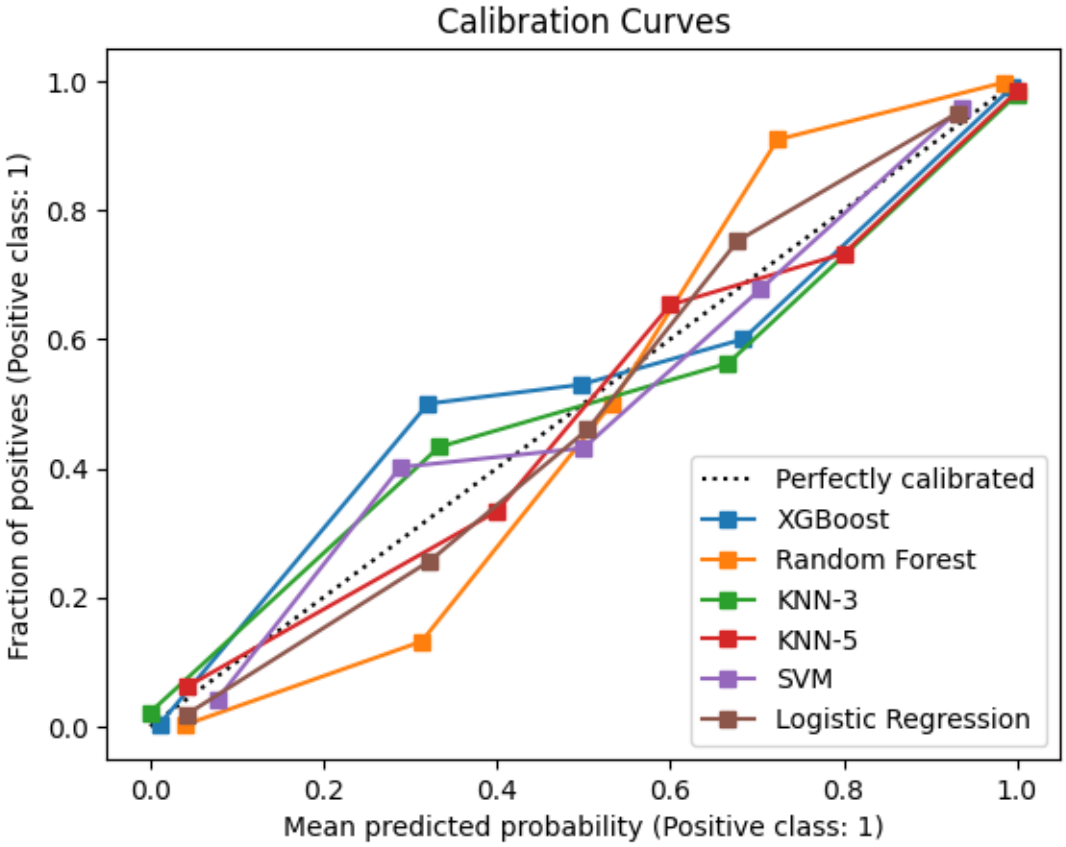

Figure 3: Calibration Curves for the models: Which examines how meaningful the probability predictions are made by the models

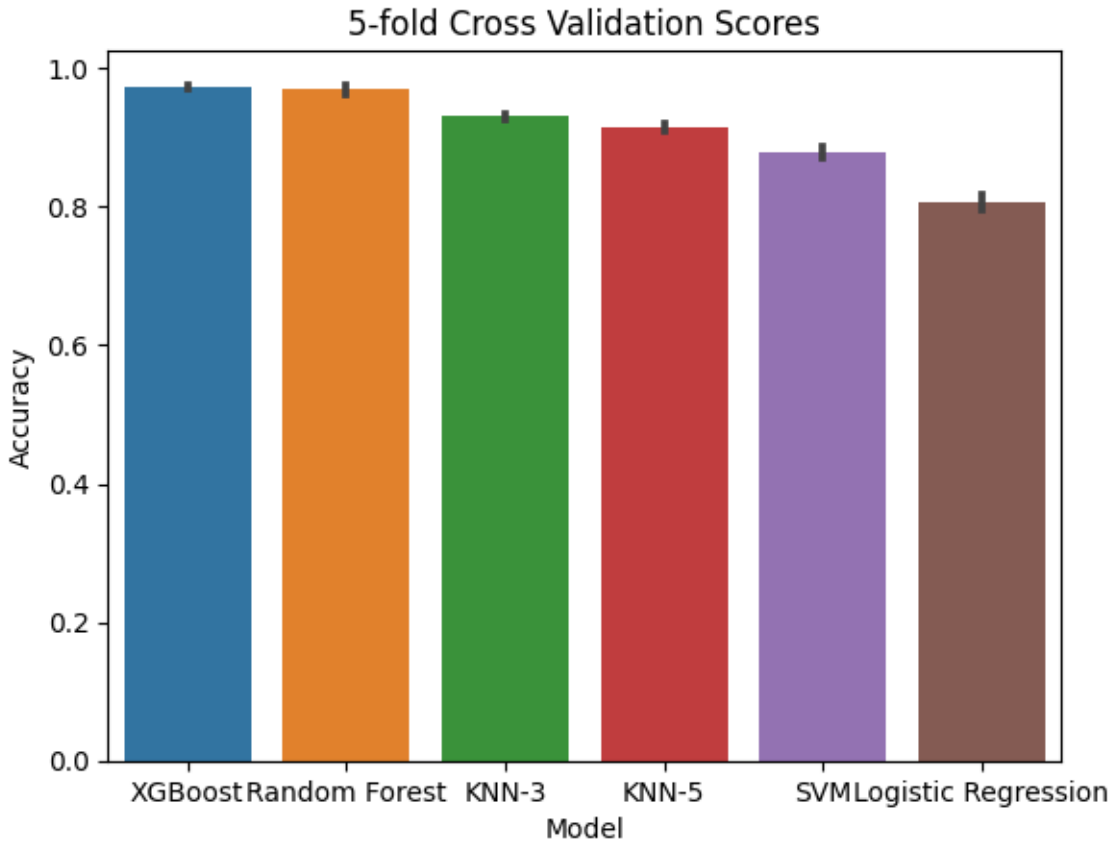

Figure 4: Cross Validation Results: Showing the accuracy of the models on the training data

The principal metric I used in comparing models was the (AUC: Area under the Curve) based on ROC Curves for the models as seen in figure Figure 2. The random forest and XGBoost classifiers performed the best overall. This isn't surprising since these models are the standard for tabular data classification problems like the one considered here. Overall I would recommend the XGBoost classifier because of its ease in implementation on the elastic stack (although random forests are almost as easy as well), speed of training and inference.

The poor performance for the SVM/Logistic Regression models is likely due to the fact that the data is not linearly separable. A neural network model will likely perform worse than these models (and indeed they do in my experimentation) since they assume some smoothness in the data which is not present here. The details for the models are outlined in the `nbs/00_core.ipynb` notebook.

4 ML Ops Pipeline

Since the underlying model is a XGBoost classifier this whole stack can actually be configured to run on elastic cloud as a classification task. The model training can also be done on an elastic ML node. Integrating with the elastic stack takes care of a lot of the ML Ops pipeline including feature registry, model versioning, and model deployment. Testing and monitoring can also be done within the elastic stack using the Kibana interface.

From a large scale the pipeline steps for inference are: * Transform a PCAP file into a set of flows and compute the summary features for each flow. * Call the XGBoost model to make a prediction on each flow. * Index the flows into elastic search with the prediction (malware probability) as a field. * Visualize the predictions and log key metrics.

In the figure Figure 5 I have illustrated a flow where the initial PCAP data is stored in AWS S3 and then inserted into elastic search using an ingestion pipeline. This pipeline would effectively run the `malware-transform` cli function on the raw pcap data and insert the data into a elastic data frame. I would advocate that this process is put under a CI/CD pipeline linked to a source repository, so that any changes to the malware-transform function are automatically deployed to the elastic cloud from the main branch of the repository. The next step in the pipeline is to call the machine learning model on new data inserted into elastic. The XGBoost model could be deployed from the python code using the `eland` library. The model predictions and alerts can then be configured within kibana.

Model training could also be performed in elastic or externally and then deployed using the `eland` library. Training data could be loaded into elastic using the same versioned ingestion pipeline as inference with the additional label field specified. The model could then be trained from within the elastic stack using the eland library to version and track model metrics and

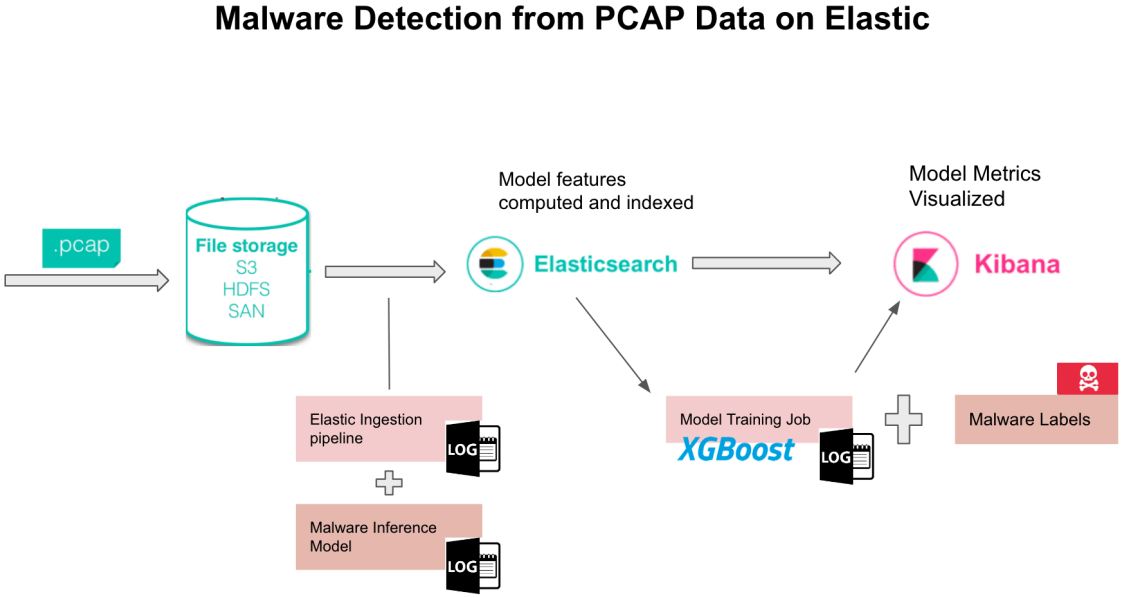

Figure 5: ML Ops Pipeline

5 Conclusions

In this project I have demonstrated the use of machine learning to classify network traffic as either malicious or benign. I have shown that the XGBoost/Random forest classifiers perform the best overall in this task. I have also outlined a ML Ops pipeline for the training, deployment and testing of a malware prediction model within the elastic stack. If I was to continue working on this project I would spend some time doing some hyper-parameter tuning on the XGBoost model to see if I could improve the performance of the model. I have used a bayesian optimization routine for this tuning in the past and expect it would work well under these circumstances.

However, the most important thing to do would be to collect more data. The high degree of variability in network traffic has hardly been captured in these demo data sets and more data is needed to do a complete assessment of the classifiers.

As a more long term project I would like to explore the use of techniques from Natural Language Processing to malware detection from network traffic. The idea would be to translate the flows into a sequence of symbolic language and then apply a transformer architecture to the problem. The ability of transformers to capture long range dependencies in sequences of symbols would be well suited to this problem. This model could be run alongside the XGBoost summary statistic approach as an ensemble model to detect malware signatures from the network traffic.