Malware Prediction Using Neural Networks

Bharath Ratna Karumudi, Sriganesh Chandrasekaran, Barry Armour, Izzat Alsmadi

College of Engineering and Computer Science
Syracuse University
Syracuse, NY USA - 13244
(bhkarumu,srchandr,blarmour,imalsmad)@syr.edu

Abstract — **Antivirus and malware detection software's primarily uses a signature based approach. These software's use a set of rules from a library, based on previously known malware and published vulnerabilities from the software manufacturer. The rules are very specific to the vulnerability and malware, thus making them brittle and very context and malware specific. So a new malware release still has the potential to take advantage of the same vulnerability that has already been addressed for a different malware.**

The Antivirus software provider releases updates to the library of rules on a periodic basis, or as critical vulnerabilities are discovered. One risk with the traditional Antivirus or malware detection software is zero-day vulnerabilities, where the hackers discover it before either the software manufacturer or the antivirus provider. In this implementation, we will be attempting to build such models using the Malware dataset.

Keywords—Malware, Neural Networks, MalConv, Machine Learning.

I. Goal/Hypothesis

The goal of our project is that we are given a data set[1] that contain malware data in bytes and asm files. We would like to predict the best possible match for the malware to a set of target based on the program characteristics (like Header, bytes file info, IDAT disassembled code features). The overall research hypothesis for this dataset is to parse the data, extract the features, model and then classify the malware into one of the nine categories: *Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY, Gatak.*

We will initially experiment with numerous algorithms and formally move further with the one that accomplishes the outcome with best accuracy. The main challenge with the dataset is the domain knowledge and we need to do domain research as well to do some feature analysis which will help us determine potential features to use.

There are two types [3] of Malware analysis:

- *Static analysis* analyses the malware attributes by executing a static parser and understanding underlying patterns on the suspect malware fingerprints.
- *Dynamic Analysis* is basically executing the suspect malware in a controlled environment and understanding the behaviour.

In our approach, we will be extending the static analysis with Machine Learning and model predictions through the end goal of mapping the dataset with high degree of accuracy to the provided fingerprints. We also intend to understand and learn on the static analysis of malwares (pros and cons) as an additional item. If computing power is not sufficient and scalable with our limited resources to perform the analysis, we may end up limiting the data set.

The entire end to end Machine Learning processing involved:

- Data collection and Preprocessing.
- Feature Selection/reduction.
- Classification analysis.
- Modeling
- Ensembling - techniques to improve the overall techniques including stacking and voting based techniques for multivariate classes.

II. Dataset

The dataset is from Microsoft[1] and is about 500 Gigabytes, which includes:

- 10,868 malware files (Bytes file+ASM disassembled version) in training dataset.
- 10,873 malwares in test dataset.
- trainLabels.csv - which has the labels for training dataset.

Every malware has two related files:

- Hexadecimal byte contents.
- ASM files produced by a commercial Disassembler IDA pro.

Figure 1 and Figure 2 shows the samples of ASM and Byte files respectively:

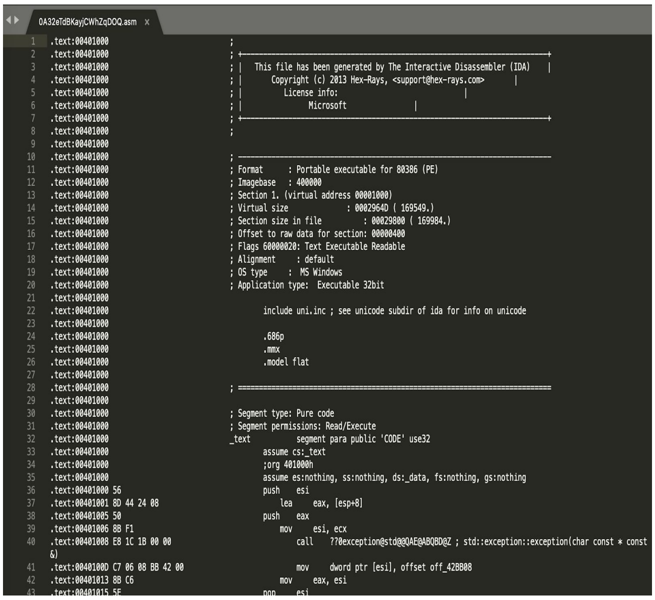


Fig.1. Sample asm file

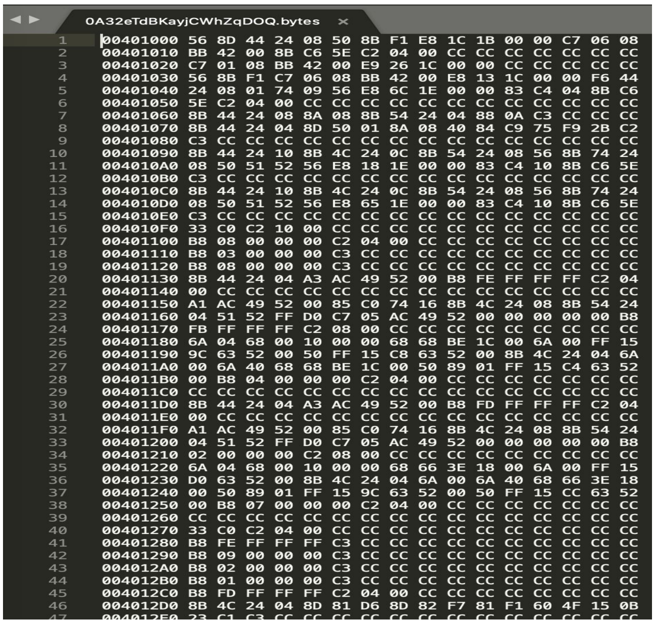


Fig.2. Sample bytes file

III. Related Contribution

We have identified several implementations on this Kaggle competition dataset [1]. This section will summarize their implementation and our understanding of their approaches. We referred to some of the solution approaches from Kaggle and how they had proceeded from thought process to give us some ideas. Almost all major approaches used some sort of xgboost/gradient boosting techniques to improve performance as the dataset is humongous to train and validate on.

Out of them, one is by the first place winner [2] whose approach is on intensive feature engineering and used the Gradient boosting (Xgboost), Ensembling, semi-supervised learning. The main model implemented is Gradient Boosting and also cross validation helped in avoiding the overfitting. They extracted three kinds of features: opcode N-gram count, segment line count and asm file pixel intensity features and extracted the features by writing the code in Python. For supervised modeling, Xgboost and ensemble are used. The below image [2] can depict the high level overview of their approach.

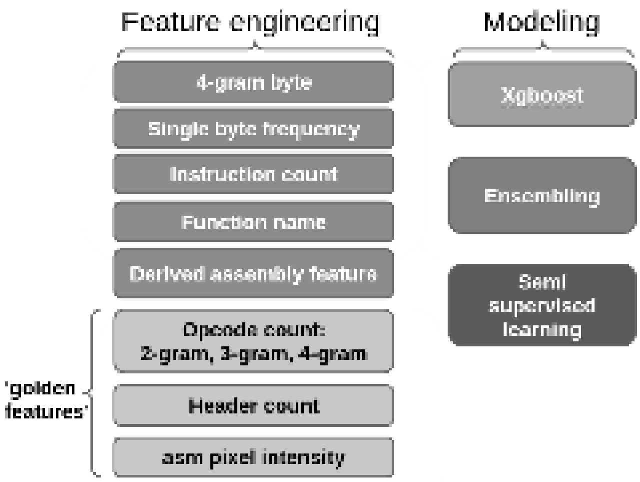


Fig.3. Overview of the approach

Also an interesting implementation was ASM file pixel intensity feature, where malwares can be visualized as gray-scale images. So they extracted the gray-scale image from asm files. The model was built on Xgboost to perform a multiclass classification. The implementer also tried Random Forest, Naive Bayes, but Xgboost performance gave better results.

Code Execution:

The implementer's code[2] is available in GitHub repository, but it is almost five year old and a lot of libraries are obsolete and running on Python 2.7. We tried our best to refactor the code to rerun and below is the code execution to create a single model.

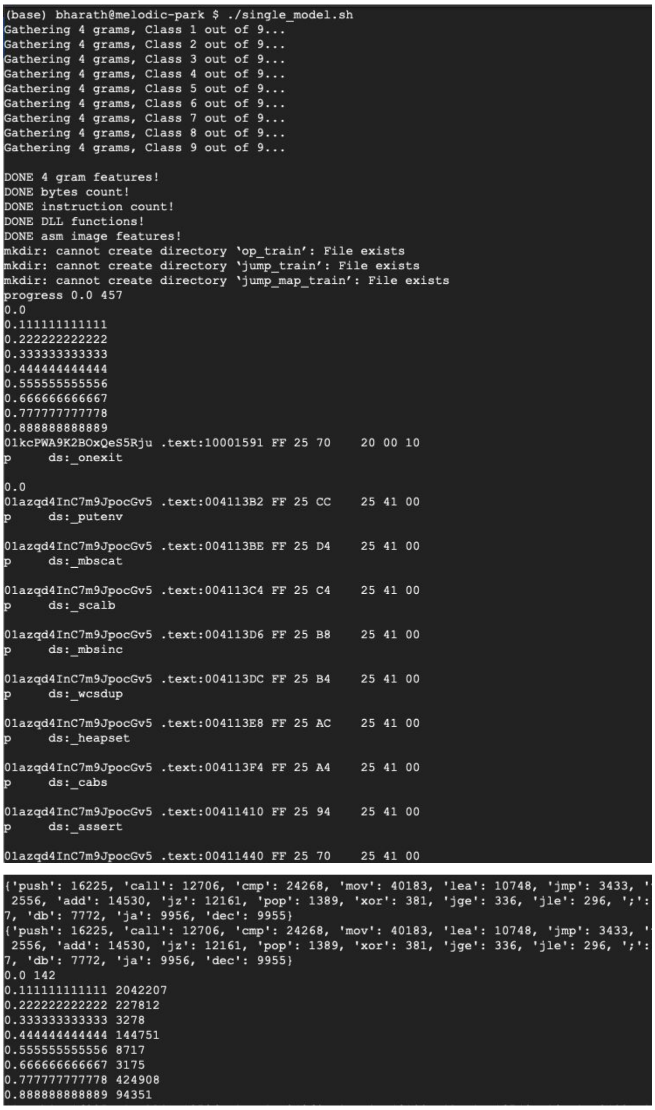


Fig.4. Code execution after refactoring

The second approach, who was the sixth place winner [4], used iterative method of feature extraction and model training. In this approach, the researchers had vetted both bytes file and the ASM file for certain unique characteristics that can be considered as features.

Bytes file: One interesting thing to note was that the approach actually took the physical file size properties (in terms of the bytes file and the ratio with corresponding ASM files) which is pretty interesting. This was also done for the ASM file to extract as much features as possible.

There is also the usage of finding out the numeric values of 1, 2 and 4 byte n-grams and calculate ratios of these. The way N-grams were calculated is as below:

Let's consider the *OA32eTdBKayCWhzqDOQ.bytes* file as an example:

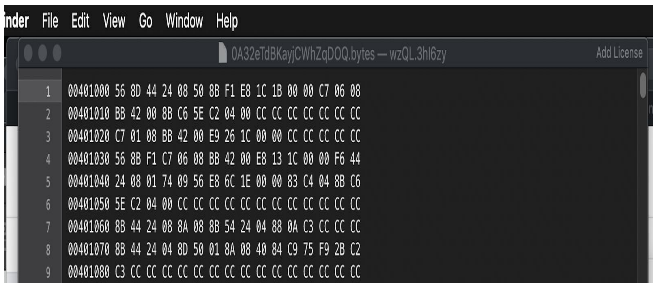


Fig.5. <Desc>

The way 1 gram, 2 gram and 4 grams are calculated are as below -

00401000 56 - 1 gram
00401000 44 24 - 2 gram
00401000 F1 E8 1C 1B - 4 gram

The research then goes into collating the weightage of each of these n-grams and consolidating based on popular or repeatable n grams that can be used as feature for prediction.

ASM file

In addition to the file size and compression size characteristics, within the ASM file interpunction characters like '','?', '.', '+' , '-' etc. were also looked at to understand the density of these characters. They also looked at the proportion of lines or characters in each section namely the HEADER,.data, .rdata. This approach though does not need a lot of domain knowledge behind the assembler coding itself. The model had used Xgboost and gradient boosting trees.

The whole analysis is done in two parts -

1. First, evaluate the set of new features using untuned modelling
2. Predictions from untuned model is then used as features in the meta bagging model for the final accuracy evaluation.

The other approach [5] combined an examination of the software fingerprint of each ASM file to look at the type of code used in the ASM file as well as the bytes file ngrams /characteristics. This helped to come up with a pool of features and categorize them into relevant feature sets. The total number of features pulled in were 28000, which were grouped into 9 feature sets based on unigrams, bigrams in bytes files and opcodes frequency on n-grams in ASM files at a high level. The method to classify used Gradient Boosting with a depth of 3 and a K-fold validation where K=6.

One interesting thing that we found out is that there will be a huge challenge in parsing the ASM files and properly classifying them into sections like .data, .rdata etc. We are analysing multiple ways of coding this parser tool and are trying out few examples to ensure we can select the right features, correlation between the features and then doing the predictive modelling tests.

One common theme used across multiple solutions is boosting and bagging methods which improves the performances of the models and help in scalability.

IV. Feature Selection/Engineering

Each malware file has an identifier, a 20 character hash value uniquely identifying the file, and a class label, which is an integer representing one of the nine family names to which the malware may belong. There are multiple ways that we are planning to approach the feature selection for both bytes file and the ASM files.

We used *RapidMiner* predominantly for Feature Selection and Engineering techniques. It is critical to gather the tokenized list of n-grams and opcodes for rich and varied subset of features, which we describe in the below section. In addition, since the volume of feature set was high, it was critical to do feature reduction using known techniques like PCA to get out of the 'Curse of Dimensionality' and avoid noise in the datasets.

Data collection and Preprocessing:
The data preprocessing included writing elaborate processes to get useful information out of the large training dataset provided, reading them as text, tokenizing and generating n-grams/filtering tokens as shown in the process snapshot below.

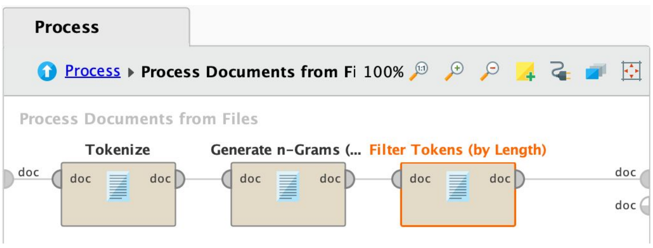


Fig.6. Collection of n-grams from Bytes files

Bytes file - Extract the features of 1 gram, 2 gram and n-gram hash codes. For our experiment, due to computational limitations we restricted our modelling to 1 gram and 2 gram on the bytes file which still yielded the necessary accuracy. During the selection of attributes we used the below steps to process documents from files in loop, generate and select attributes.

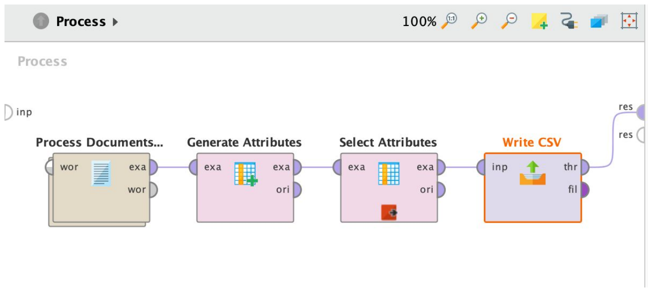
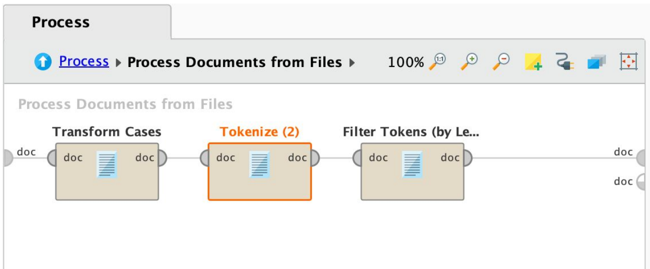


Fig.7. Final extraction/selection of bigram attributes to n-gram files

The mode used for bi-gram collection from Hex bytes file was to process the entire set of bytes file as a Process Document loop in RapidMiner. Series of extraction steps involve tokenizing the entire hex file for the 2 byte grams(modifying the filter criteria to extract 2-byte grams), their TF-IDF frequencies, generating the attributes including the metadata file attributes and consolidating into a single CSV for further processing steps. The base n-grams were 2-byte opcode as explained by Zak. et.al[10]. This yielded close to 65,000+ attributes which were further reduced to the most relevant attributes using the variance based selection/correlation techniques to 1000+ n-gram features on the collection dataset.

ASM file - We looked at the base set of opcode operators with particular focus on opcodes that impact memory processing like arithmetic/branching instructions(sub, mov, jmp, jnz) for tokenization and filtering. We processed the ASM files as text files, tokenized based on the opcodes n-gram and processed as a CSV file for the initial step.

The process step for ASM files was constructed similar to that of the bytes file and shown in the below figure:



The parsing was done through RapidMiner RegEx based tokenization and extraction which yielded good results in the data collection step. This is similar to using a custom dictionary /wordlist to pull the relevant tokens from the ASM files as the opcodes may not substitute for english words directly in the dictionary.

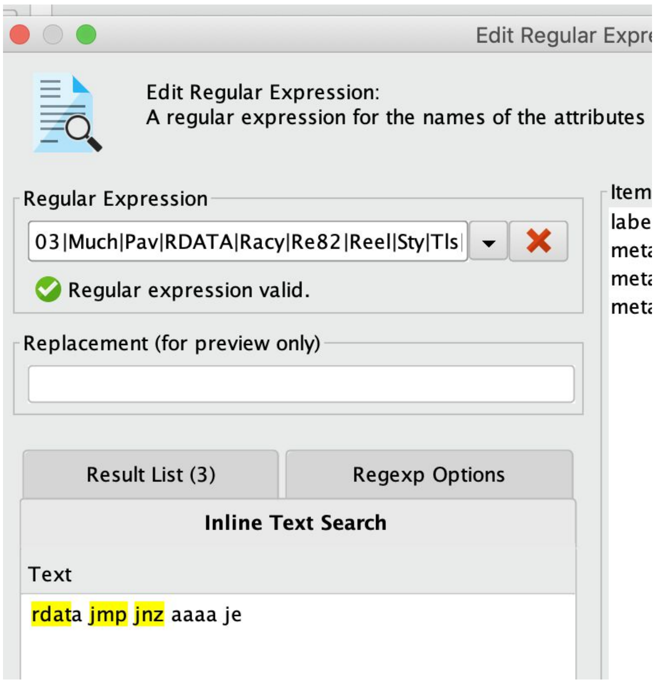


Fig: List of Opcodes in RapidMiner RegEx operator

This ended up with 150+ core opcode based attributes to be processed on the ASM files.

VISUALIZATION :
A quick visualization on the bytes n-gram as shown below clearly indicates there are outliers that can cause additional noise for modelling. We may have to remove them further using the normalization/feature reduction step before running through the modelling techniques.

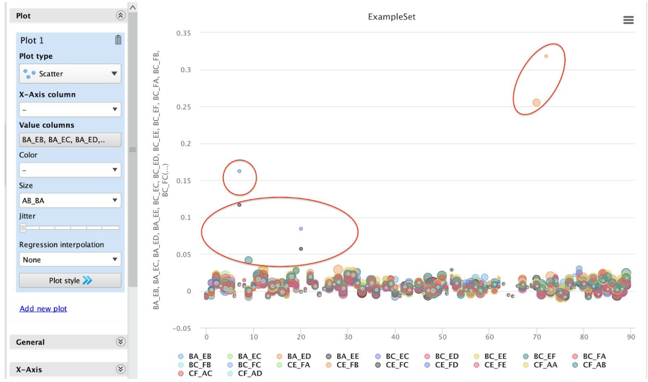


Fig: Outliers in bigrams extracted from bytes files

On a similar approach, ASM file features were also extracted and below is a quick visualization of the extracted dataset for further processing. On close analysis of the extracted features, we found a dense scatter on some of the memory handling/branching opcodes like mov, jnz etc as can be seen below. These seem to add more information to the feature dataset and hence higher weightage during the modelling phase.

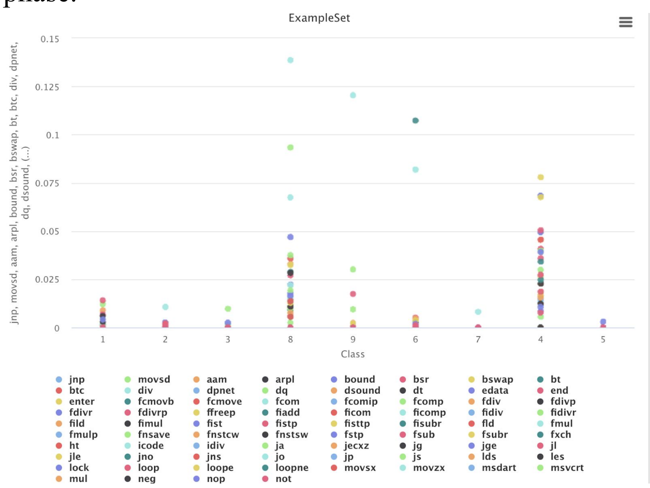


Fig: Opcode based distribution across classes

Feature Selection / Reduction:
From the fig.7 the distribution of Training Labels can be clearly seen that the training data set is not evenly distributed dataset. So, we may have to look for some level of normalization on the bytes and ASM datasets when we extract the features to see if there are even distributions.

The output of the ASM file feature extraction and the bytes file extraction was consolidated with training labels provided in the competition in the RapidMiner process with class as the label. Here the class is multi-variate (0-9) and is polynominal which needs to be taken into account for further modelling.

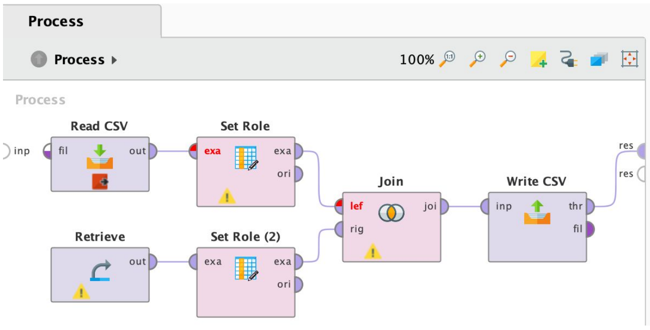


Fig: Creating a Labelled dataset for classification

As we can see the number of files provided across classes are not uniform. There is a significant bias against class 5 and class 7 which has a limited number of samples.

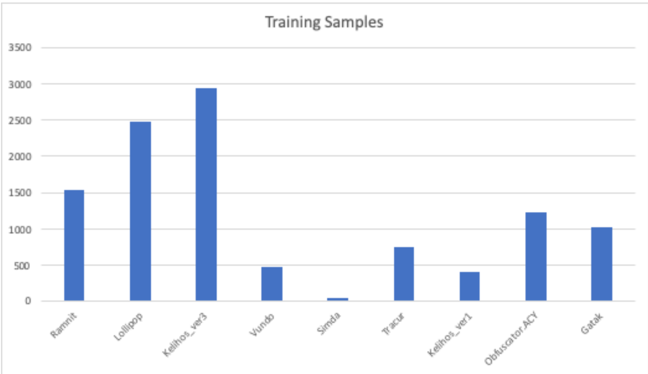


Fig.8. Label distribution from training dataset

V. Training Methods

We have built our models using:

- Naive Bayes
- Random Forest
- MalConv using Neural Networks
- Ensemble techniques using Stacking and Voting

Random forest is especially useful in large features specifically like in cases like ours where we have 1000+ n-gram features from bytes files and 110+ features from ASM files in our feature set.

The training model involved building three different variations and evaluating the performance measures. The first approach we took was to classify the malware samples by multi-class on the bytes bi-grams.

Parallelization: The main constraint was to handle the parallelization of the modelling run, which we accomplished by batching the feature selection steps and using the Append operator in the process to the output dataset.

The model itself is shown as below using Naive Bayes:

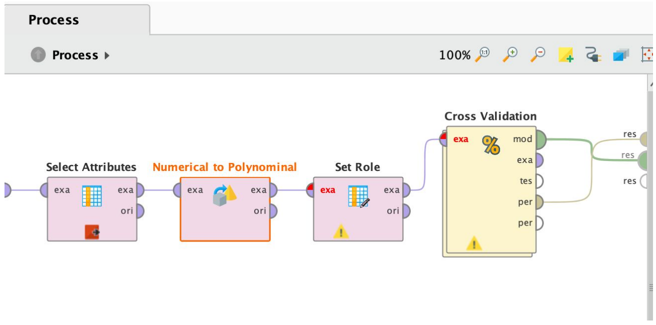


Fig: Overall Modelling process using cross-validation

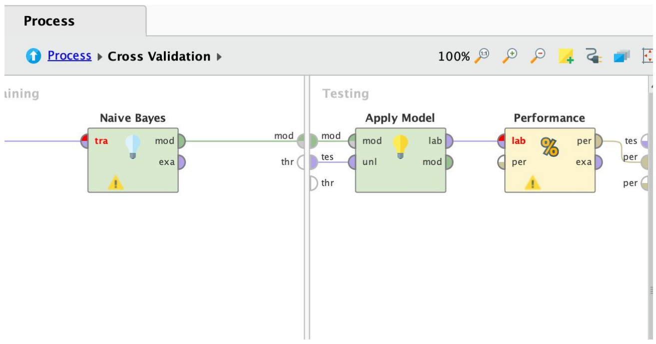


Fig: Initial Naive Bayes based classification on the dataset

While the initial Naive did yield some accurate results, we wanted to explore additional modelling using the ensemble techniques and result:

Bytes n-gram model
ASM opcode n-gram based model
Combined bytes+ASM model
Ensemble methods
ACCURACY IMPROVEMENTS

Malconv - A Neural Network based Architecture

There were many implementations on this dataset, but none with Malconv, so we have selected this architecture and built the model using Malconv. The Malconv model was designed by Nvidia[2] as shown in figure, specially for Malware Detection and the model aimed for three features:

1. The ability to scale efficiently in computation and memory usage with sequence length.
2. The ability to consider both local and global context while examining the entire file.
3. An explanatory ability to aid analysis of flagged malware.

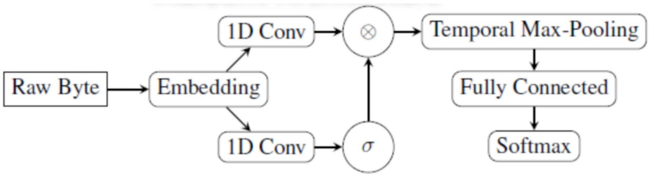
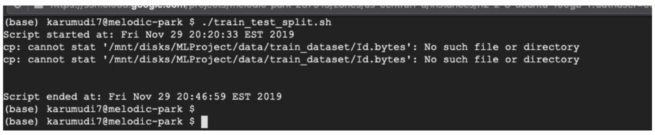


Fig: Malconv Architecture

The model does not need to perform any feature engineering,

Malconv Model Execution:

Splitting the dataset into Train and Test datasets - for this we have a shell script that will segregate the files into two different directories from where our program will read.



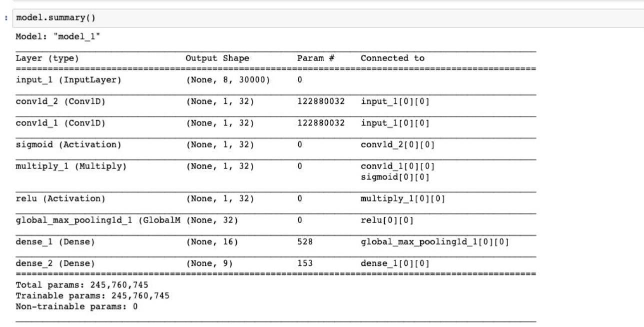
```
(base) karumudi7@melodic-park $ ./train_test_split.sh
Script started at: Fri Nov 29 20:20:33 EST 2019
cp: cannot stat '/mnt/disks/MLProject/data/train_dataset/Id.bytes': No such file or directory
cp: cannot stat '/mnt/disks/MLProject/data/train_dataset/Id.bytes': No such file or directory


Script ended at: Fri Nov 29 20:46:59 EST 2019
(base) karumudi7@melodic-park $
(base) karumudi7@melodic-park $
```

Fig: Train-Test split process

The program was written in Python 3 and using Keras and TensorFlow libraries. The approach starts by splitting the given Train dataset into two Train and Test by randomizing the given TrainLabels.csv - for this the above mentioned shell script will do the operations on the file system and prepares the dataset.

We clean the byte sequences for any unknown characters and do the embedding by converting the input to a numerical format - each byte into an eight dimensional vector. Using Keras functional API, we are capturing the Conv1D to conv1 and conv2 and then activate one of the convolutions, say conv2, sigmoid. We multiply the Activated convolution and non activated convolution and then activate the final result - relu. With these, we build the model and the built model summary is as shown in the figure.



```
model.summary()
```

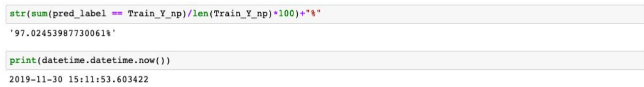
Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 8, 30000)	0	
conv1d_2 (Conv1D)	(None, 1, 32)	122880032	input_1[0][0]
conv1d_1 (Conv1D)	(None, 1, 32)	122880032	input_1[0][0]
sigmoid (Activation)	(None, 1, 32)	0	conv1d_2[0][0]
multiply_1 (Multiply)	(None, 1, 32)	0	conv1d_1[0][0] sigmoid[0][0]
relu (Activation)	(None, 1, 32)	0	multiply_1[0][0]
global_max_pooling1d_1 (GlobalM	(None, 32)	0	relu[0][0]
dense_1 (Dense)	(None, 16)	528	global_max_pooling1d_1[0][0]
dense_2 (Dense)	(None, 9)	153	dense_1[0][0]

Total params: 245,760,745
Trainable params: 245,760,745
Non-trainable params: 0

Fig: Model Summary

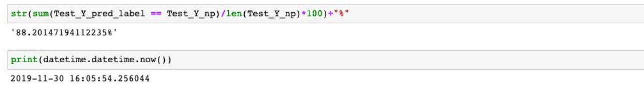
The built model is trained with our training dataset and predicted with an accuracy of 97%.



```
str(sum(pred_label == Train_Y_np)/len(Train_Y_np)*100)+"%"
'97.02453987730061%'
print(datetime.datetime.now())
2019-11-30 15:11:53.603422
```

Fig: Training Dataset Accuracy

The model is also tested with our test dataset and predicted accuracy of 88.2%.



```
str(sum(Test_Y_pred_label == Test_Y_np)/len(Test_Y_np)*100)+"%"
'88.20147194112235%'
print(datetime.datetime.now())
2019-11-30 16:05:54.256044
```

Fig: Test Dataset Accuracy

The given test dataset don’t have any labels, so we performed a prediction on the given test dataset and model classified the malwares as below:

VI. Model and Execution Time

We have built our machine learning model using multiple models and below table shows the accuracy and execution time for each.

Model	Accuracy	Execution Time

VII. Artifacts and Technology used

GitHub: URL
Technology: Python 3.7, RapidMiner 9.4
Environment:

- MalConv: 8 vCPU, 64GB RAM and 600 GB
- RapidMiner: 2 CPU, 8 GB RAM

VIII. Conclusion

The traditional antivirus and malware detection software's approach is insufficient in growing malwares every day. Because every environment is unique and has specific binaries and these detection softwares might never seen those before and with millions of new malware samples every day it will be difficult. So there is a need to develop a detection system that can that can adapt to the rapidly changing malware ecosystem is seemingly a perfect fit for Machine Learning. As implemented in this project with Malconv[2] architecture we could build an anti-virus system without feature engineering and that would save efforts and allows to detect the malware across a variety of operating systems and hardware.

IX. References

[1] Microsoft Corporation. (2015). Microsoft Malware Classification Challenge (BIG 2015). Retrieved from https://www.kaggle.com/c/malware-classification/data.

[2] Xiaozhou Wang, Jiwei Liu, Xueer Chen, "Say No to Overfitting" Winner of Microsoft Malware Classification Challenge (BIG 2015).

[3] Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., & Nicholas, C.K. (2017). Malware Detection by Eating a Whole EXE. ArXiv, abs/1710.09435.

[4] Eureka: A Framework for Enabling Static Malware Analysis, Monirul Sharif1, Vinod Yegneswaran2, Hassen Saidi2, Phillip Porras2, and Wenke Lee1, College of Computing, Georgia Institute of Technology {monirul,wenke}@cc.gatech.edu, Computer Science Laboratory, SRI International {vinod,saidi,porras}@csl.sri.com

[5] Microsoft Malware Classification Challenge 2nd place solution documentation , Marios Michailidis (London, United Kingdom), mimarios1@hotmail.com, Gert Jacobusse (Goes, The Netherlands), gert.jacobusse@rogatio.nl

[6] Microsoft Malware Classification Challenge 6th place solution 21.04.2015, Oleksandr Lysenko

[7] Metamorphic Virus Detection in Portable Executables Using Opcodes Statistical Feature, Babak Bashari Rad, Maslin Masrom.

[8] Detecting unknown malicious code by applying classification techniques on OpCode patterns. Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev and Yuval Elovic

[9] Malware Images: Visualization and Automatic Classification. L. Nataraj, S. Karthikeyan, G. Jacob, B. S. Manjunath.

[10] Zak, R., Raff, E., & Nicholas, C. (2017). What can N-grams learn for malware detection? 2017 12th International Conference on Malicious and Unwanted Software (MALWARE). doi: 10.1109/malware.2017.8323963

[11] Mitchell Mays, Noah Drabinsky, and Stephan Brandle. 2017. Feature Selection for Malware Classification.. In MAICS. 165–170.