Classifying Malware with Extreme Gradient Boosting via content-based features

MSc Data Science 2018 Final Project

Geoffrey Owden ID13031392

This page is intentionally blank

Contents

List of Tables, Charts, Figures, Images, Diagrams and Files

Files contained on USB

Python Files
- GUI – Main file to open to classify malware
- MASTER_UTILITY_FUNCTIONS – Utility functions for GUI.py
- PCA40 – PCA for finding the top 40 PC's
- DataCleaner– for cleaning the dat set

Pickle Files
- XGB_Model – Trained model used in prediction

CSV Files
- TrainingDataRaw – 1st cleaned version of the dataset with full features
- Training Data PCA 39 – Dimensionally reduced dataset
- Trainlabels – class labels of the data set for training
- x86_Opcodes – master reference file of opcodes and feature names

PDF
- Final report – PDF version of final report

INTRODUCTION

This project explores how malicious software can be classified into different families via content-based features e.g. file size, opcodes. It explores the development of a lightweight software tool for that classification via features which require little domain expertise or knowledge of the inner working of the family class of Malware sample. The method allows non-practitioners the ability to participate and offer scalability to future implementations.

The aims of the research are

1. Identify a dataset which contains a diverse range and healthy volume of suitable and safe malware samples.
2. Accurately clean and transfer a raw dataset to a robust format for use in a practical classification exercise.
3. Research recent malware classification methods based on content-based features.
4. Identify and isolate the key features of the dataset through statistical measurement to optimise the classification and computational processes.
5. Produce an easy to use and robust piece of software which is significantly less susceptible to user error or malicious use.
6. Attempt to accurately classify malware samples by the defined features.

The aims for my personal development are

7. Improve upon software development skills in Python with libraries and methods which have **not** been covered in my current set of modules.
8. Develop the skills of training and then implementing a classification model in Python.
9. Work with state-of-the-art/industry classification models to develop experience which is up-to-date.
10. Develop the skills of building a classification framework which can implement various classifiers.
11. Emulate the work of a professional software project following a prescribed and identified methodology.
12. Work in a clear and efficient fashion which does not obstruct the full exercise of a software development project.
13. Contribute to a portfolio of work directed towards a career in cyber security and data science.

Definitions of terms used

An **.asm** file refers to a malware file which has been reverse-engineered through a disassembler.

Opcodes refer to the computer instructions which have been extracted from the an .asm file.

The term **gram** means a segment of words of any language and will generally refer to Opcodes.

A **sample** will mostly refer to a malware sample file which is either training a model or being classified.

Background

Weaponised software for malicious application is a serious threat affecting economies, enterprise operations and the lives of many. Digital networks and architectures are increasingly vulnerable via the expanding attack surface. This is due to the proliferation of digital devices and their interconnected networks.

malware as a class is evolving. The traditional model of a single software working alone is developing to one of fragmented software converging on a target, thereby avoiding detection [19]. Remedial methods and implementations for this problem are vital. Stakeholders within this problem's space are mainly Network Security professionals, but informally anyone with a digital device could be included. From the research taken in the proposal stage, two current malware classification approaches are mainly used. The first and riskier approach is the **static** "fingerprinting" of a known malware sample for classification. This means that an infection must have happened, either in a contained space or "in the wild" before a fingerprint can be taken. The second is the **dynamic** analysis of live samples to understand the heuristics in their behaviour, which is slow and costly due to the domain expertise needed. Exploration into a faster and safer method of statistical classification via static content-based features means a sample can be initially screened by a machine before being run, and if suspicious, flagged for further inspection by a domain expert. This will save resources.

A successful statistical classification model which has had wide success and acclaim, regardless of the subject domain, is Extreme Gradient Boosting (XGBoost). XGBoost has evolved from traditional Gradient Boosting (GBM). Both will learn from their mistakes and reapply the lessons learnt to the algorithm's next iteration. XGBoost has developed upon this approach as well as implementing processing optimisations. XGBoost will be the starting point for the classification model.

The dataset being used is Microsoft’s BIG15 Malware Dataset which was used in a malware classification competition in 2015 hosted by Kaggle [20]. The data is a 9 class, unbalanced, labelled set of 10k+ samples.

Table 1 – dataset fundamental statistics				**Opcode Related stats**					
Malware class No.	**Name**	**Type**	**No. of samples**	**Mean No. of Opcode**	**Median**	**Min**	**Max**	**Range**	**Non-statistical Outliers**
1	Ramnit	WORM	1542	45944	18449	66	475615	475549	8 files with 1 opcode
2	Lollipop	ADWARE	2478	23757	6059	612	221360	220748	2 files with 0 opcodes
3	Kelihos_ ver3	BOTNET	2943	1277	1263	390	4938	4548	4 files with 0 opcodes
4	Vundo	TROJAN	439	2395	2303	45	13935	13890	20 files with 0 opcodes
5	Simda	KEYLOGGE R	43	2644	2388	608	7234	6626	0
6	Tracur	TROJAN ROOT KIT	751	5916	3802	59	34197	34138	0
7	Kelihos ver1	BOTNET	390	3682	741	5	273448	273443	6 files with 0 opcodes
8	Obfuscat or.ACY	SPYWARE	1228	6326	4595	172	190696	190524	9 files with 0 opcodes
9	Gatak	TROJAN	1013	7422	4610	18	78050	78032	2 files with 0 opcodes

Constraints on this project are

- Time, due to the submission deadline of 12th September.
- The complexity of the problem and the need to limit the scope of the project to be viable within the time constraints.

Research question: *Can .asm malware files be classified by some of their content-based features using an Extreme Gradient Boosting model in order to circumvent the need for domain expertise?*

A software implementation of this question should be able to take an .asm file, document features from it and run those through a classification model displaying the output to the user.

Reading this report

An appendix is attached at the end of this report containing technical notes on classifiers mentioned, PCA charts and definitions to clarify project methodologies. A glossary of key terms which may need more elaboration is also attached. References to other works are made by use of square brackets “[ ]” containing a number which can be linked to the reference section at the end of the report. Software is attached in a USB.

RESEARCH

This section looks at how content-based features and Opcodes have been used in malware classification from single occurrences to n-gram sequences. We will also look at derived features which have been used such as file size, the conversion of malware into Grey scale images, the proven benefits of single word semantic derivation in natural language, and the classification model chosen, XGBoost .

Opcodes as a predictor for malware

Bilar (2007) [14] discusses the detection for malicious code through statistical analysis of opcode distributions via 67 malware executable samples, statically disassembled. Their opcode frequency distribution compared aggregate statistics of 20 non-malicious samples. Bilar found that malware opcode distributions differed significantly from non-malicious software. Moreover, rare opcodes seem to be a stronger predictor, explaining 12–63% of the frequency variation, gathering random samples of malicious and non-malicious 'Goodware' binaries. As an example, he found that Root kits make heavy use of software interrupts which involved a certain sub-group of opcodes.

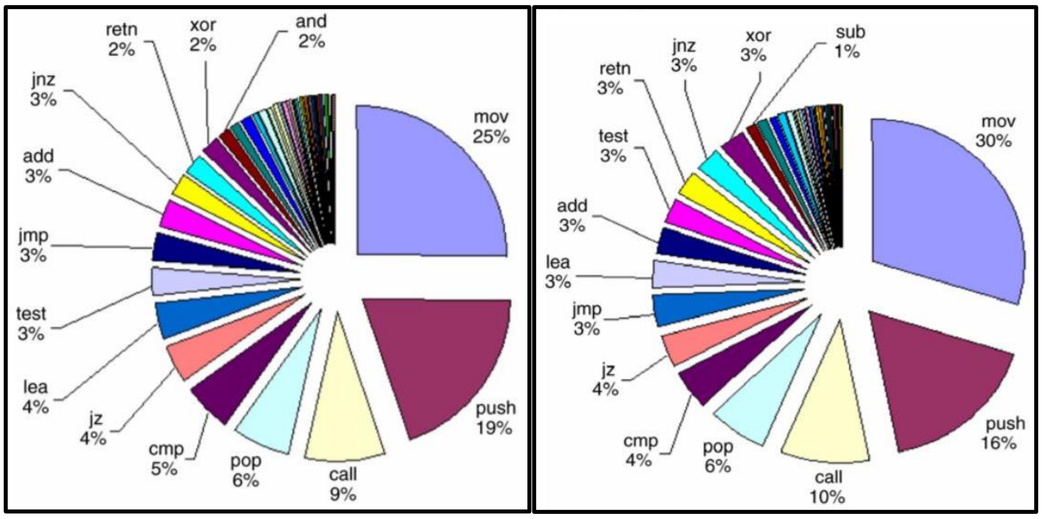


Chart 1 - Bilar's results of malware and goodware opcode distributions

Of his dataset 72 opcodes accounted for >99.8% of opcodes found, 14 opcodes accounted for 90% and the top 5 opcodes accounted for 64% of extracted opcodes. The simplicity of this approach is its merit. The limitations of Bilar's work is the use of only a reference set of 398 opcodes, which is a lot less than the 600+ that are available.

Kaggle Competition winners’ approach

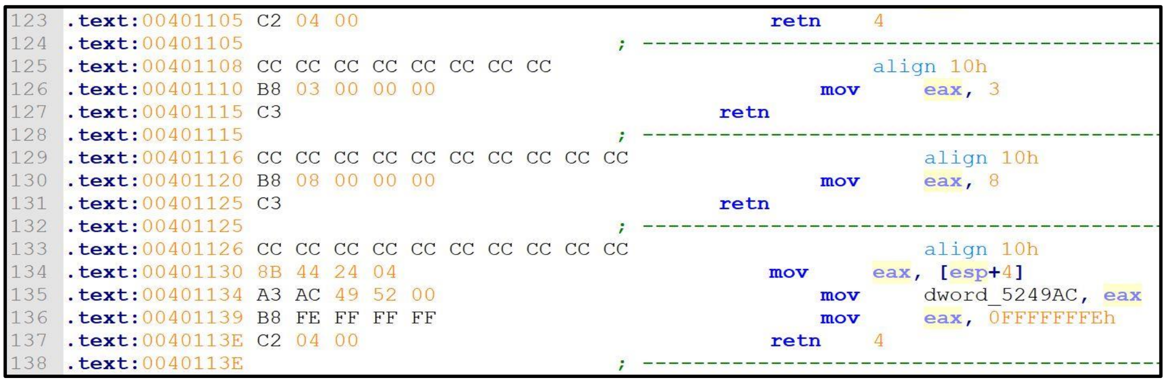
From their presentations (2015) [5] [6] the winners of the kaggle competition had no prior experience with the domain of Malware. They worked purely on the results from their models. The approach they chose identified 8 features, 3 of which being “Golden Features” were more important.

The features of their approach were

Opcode count - Opcodes were counted by frequency of appearance in each file. Candidates would be the first tokens in each line of the .asm file which is not a byte.

Segment counts - Within each .asm file the segment (or sections) are the instruction types within each line e.g. “.text:”. Liu et al (2015) recognised 448 different types of segment in the training. See in figure 1.1.

.asm pixel intensity - This process was based on the work by Nataraj et al (2011) .They found the intensity of the first 800 pixels to be a good indicator. On its own, they claim it was not very impressive; however in combination with other features, it made a significant positive impact, but exactly how, they are unsure.



```
.text:00401105 C2 04 00                                          retn    4
.text:00401105                              ; ---------------------------------------
.text:00401108 CC CC CC CC CC CC CC CC                                   align 10h
.text:00401110 B8 03 00 00 00                                  mov     eax, 3
.text:00401115 C3                                      retn
.text:00401115                              ; ---------------------------------------
.text:00401116 CC CC CC CC CC CC CC CC CC CC                             align 10h
.text:00401120 B8 08 00 00 00                                  mov     eax, 8
.text:00401125 C3                                      retn
.text:00401125                              ; ---------------------------------------
.text:00401126 CC CC CC CC CC CC CC CC CC CC                             align 10h
.text:00401130 8B 44 24 04                               mov     eax, [esp+4]
.text:00401134 A3 AC 49 52 00                                  mov     dword_5249AC, eax
.text:00401139 B8 FE FF FF FF                                  mov     eax, 0FFFFFFFEh
.text:0040113E C2 04 00                                  retn    4
.text:0040113E                              ; ---------------------------------------
```

Figure 1.1- Asm code snippet

Chart 2 – Kaggle winners final feature set

Feature type	**Amount**
Opcode n-gram	4416
Segment count	19
.asm file pixel intensity	800
“Other features”	2193

They settled on a 3-model ensemble, classified with a XGboost classifier. The ensemble was able to predict with 100% accuracy Class 1,3,4 and 7 of the Malware. This approach was very effective but was very complicated requiring the work of three PhD students with great expertise.

Opcodes as Genes

Drew et al (2016) [8] entered the Kaggle competition with STRAND (*Super Threaded Reference-Free Alignment-Free N-sequence Decoder),* a gene sequence classifier which could accommodate unstructured data with any alphabet including source code or compiled machine code. They showed their method achieves accuracy levels above 95% while needing much fewer resources of the training time used by the winning team.

They viewed the gradual changes in polymorphic malware computer code as analogous to the mutation of biological sequences which occur over successive generations of an organism. Genes which make up an organism are made of DNA which are long paired strands of a 4-letter language (TAGC's). Gene sequence words or n-gram*s* are commonly searched for in various data. They chose STRAND because unlike the other gene sequence classifiers, it can process sequences of arbitrary alphabets. STRAND addresses scaling requirements by utilizing a form of lossy compression called Minhashing [28] which still supports sequence comparison, but with a much-reduced memory footprint. Each Minhashing hash function is associated with a permutation of the rows of the data matrix in question, not a fully permutated version of the matrix, the computational saving. Alignment-free methods like STRAND search for seed words first and then expand matches. Once the word length k is defined, the fixed size sliding window moves from left to right across the sequence data producing each word by capturing k consecutive bases from the sequence.

The final features used for their model required only 4GB, and both feature engineering and generating the top performing model took around 48 hours. They also used a single model of XGboost, ensembled with a Random Forest as opposed to the 3+XGboost models for the competition winners. The winners of the Kaggle also took 72 hours to complete their classification and required 500GB of disk space for the original training data and an additional 200GB for engineered features. Therefore, this approach using STRAND is much more efficient and could be used with other data sequences, such as the network traces of attacks. It does however need the understanding of complex technology which is beyond the scope of this instance of the project.

Opcode-sequence-based Malware Detection

Santos et al (2010) [4] proposed classification of malware based on the frequency of appearance of opcode sequences in a vector representation of the executables. They describe a method to mine the relevance of each opcode, thereby weighing each opcode sequence frequency in benign and malicious samples and calculating a discrimination ratio based on statistics.

They view a program *P* as a sequence of instructions *I* where *P* = (*I1, I2, ..., In−1, In*). Each instruction is composed with an opcode and a parameter or list of parameters. Gathering opcodes into several blocks which they call *opcode sequences*, they chose a length of *opcode sequence length* n, of either 1 or 2. According to Liu et al [5] longer sequences do not make much impact. They then calculated weighted *term frequency* of each opcode into a vector and calculated the cosine similarity between the two vectors. They found that the similarity degree of a single type of n was insignificant but a combination of both 1 and 2 lengths of n obtained high malware variant similarity degrees, whilst the benign similarity degrees remained low. This meant their system was able to identify malware variants and distinguish benign executables using low frequency opcode sequences as standalone features, which is encouraging. However, this method is elaborate as it requires measurement of sequence permutations which increases the size of the data and processing requirements exponentially.

Content-based features: Lines of code, size of file, text tokens, API calls

Ahmadi et al [9] kaggle competition entry focused on easier to compute content-based features from the .bytes and .asm files. They incorporated an algorithm for feature fusion which outputs the most effective concatenation of features categories, mainly: Lines of code, file size, frequency of code symbols often used in evasion (-, +, *,], [, ?, @,) , API calls to malicious addresses, Grey scale Image representation [1] and Programme Executable code sections. They used a single XGboost model ensembled with bagging rather than a more complex approach such as that by the competition winners, and achieved 99.8% accuracy. An advantage with their approach was that large-scale malware categorization tasks did not need domain expertise.

Mining frequent opcode sub-graphs with G3MD

In 2018 Khalilian et al [13] proposed a Graph Mining for Metamorphic malware Detection (G3MD), a system for static detection of metamorphic malwares on the opcode graphs of a metamorphic malware family. It extracts the frequent sub-graphs, termed *micro-signatures*. Based on these sub-graphs, a classifier was trained to distinguish between a benign file and a metamorphic malware. Khalilian et al conducted experiments on four families of metamorphic malwares common in previous studies, namely Next Generation Virus Generation Kit (NGVCK), Second Generation Virus Generator (G2), Mass-Produced Code Generation Kit (MPCGEN) viruses and Metamorphic Worm (MWOR) worms. Although the precision claimed was over 99% in most cases, with just 4 classes, the odds of false positives are high.

Classification of ransomware families based on n-gram of opcodes

Zhang et al (2018) [15] proposed an approach based on static analysis to classify ransomware. The team first transformed opcode sequences from ransomware samples into n-gram sequences. *Term Frequency-Inverse Document Frequency* (TF-IDF) was calculated for n-grams to select feature n-grams reducing the dimensions. Finally, they treated the vectors composed of the TF values of the feature n-grams as the feature vectors and subsequently fed them to five machine-learning methods to perform ransomware classification. Through their experiments, they demonstrated their approach can achieve an accuracy of 91.43%. By contrast, the average F1 test accuracy measure of the Wannacry ransomware class is up to 99%, with the Accuracy of binary classification up to 99.3%. In addition, Zhang et al (2018) discovered different feature dimensions are required for achieving similar classifier performance with feature n-grams of diverse lengths.

Single Word Counts in natural Languages

Coming from the real world, Pennebaker et al proved that 1-gram frequencies of words in natural language have shown the behaviour of humans [7]. Insights to the sentiment can be contained in the style of a text—in such elements as how often certain words and word categories occur, irrespective of context. The pronouns, articles and especially prepositions matter and are titled “Function words”. In the English language there are approximately 500 of these Function words, making up 55-60% of all the words that we use. They are the shortest words in a language and are the generally received nonconsciously by the brain in less than 0.2 seconds. Pennebaker has implemented his research into software "*Linguistic Inquiry and Word Count" [16]* and has proven to detect Personality, Status, Honesty, Mental health and Intent in individual people and groups.

Although this approach applies to natural languages , Percentage of total words is the key statistical interpretation metric, and this simple approach may hold promise with unnatural languages.

Malware Images: Visualization and Automatic Classification

Nataraj et al [1][2] classified malware by converting the binary files into grey-scale images. Each malware sample Binary file was read as a vector of 8-bit unsigned integers. Each 8-bit represented a pixel's 256 degrees of grey-scale between 0(Black) and 1(white). Nataraj et al (2011) achieved a 98% classification accuracy on 9,458 malware samples with 25 different malware families at a significantly lower computational cost than other methods. Nataraj went on to use packers to obfuscate the samples and reported that even code with sections which have encryption layered over deeper layers were still classifiable. This approach has strength in that it requires no domain knowledge, is based on an unconnected feature space, resource efficient and could be automated. Its limitations are that it is based on global image features and so circumvention can be obtained by adding redundant data or moving sections around in the code. Nataraj et al suggest research into a more localised image-based version of this model on the code. They also did not mention how many types of packer they used which is important as the packers may contribute their own identifiable signatures.

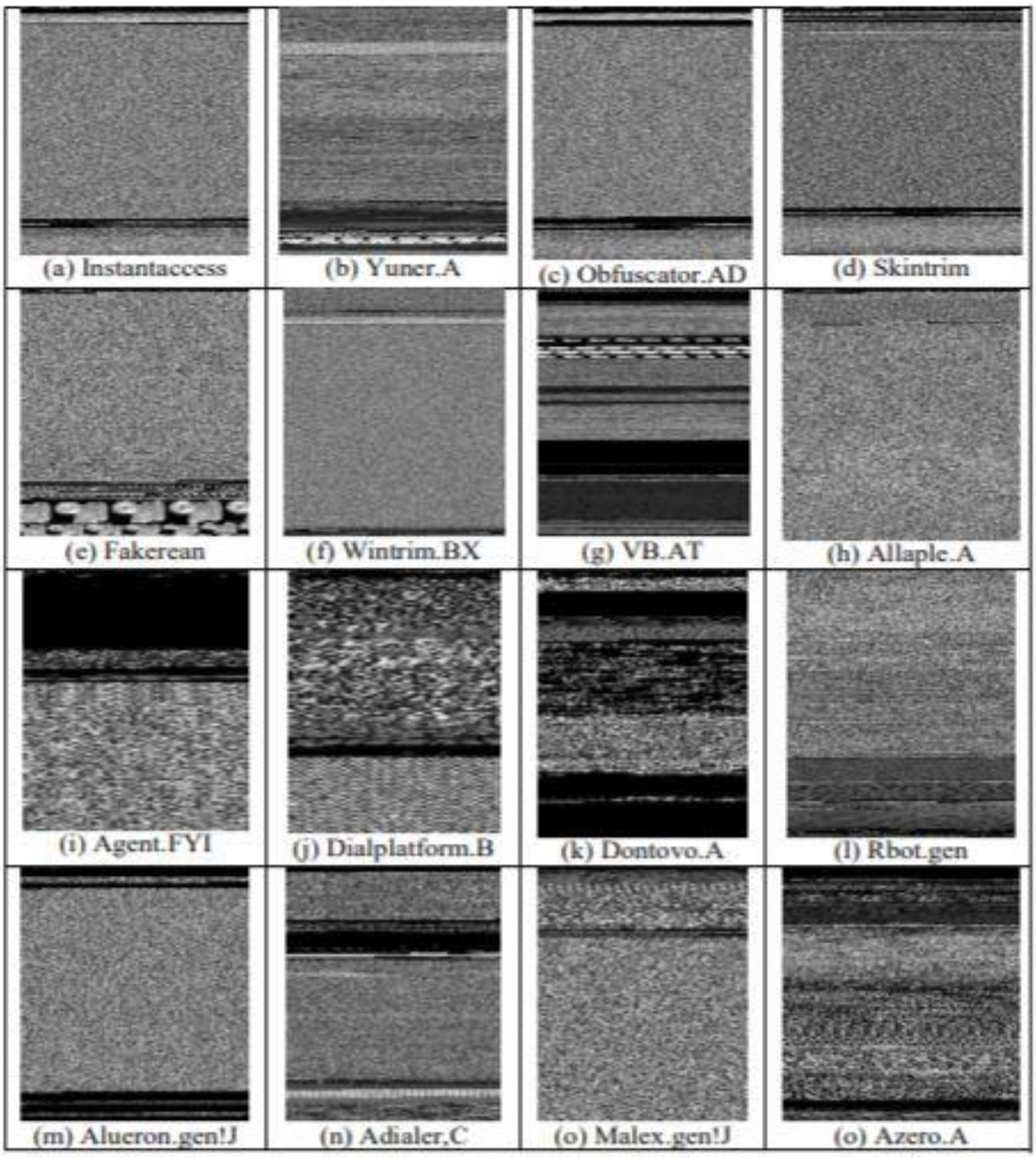


Image 1 - 16 different malware Families represented as grey scale images

Luo e tal [3] expanded on the work of Nataraj et al with an image based on extracting local binary pattern (LBP) to classify malware samples. Combining these through Convolutional Neural Networks they achieved an accuracy of 93.17% on 12000 malware images of 32 families. The two main image classification techniques the authors used are Local Binary Patterns (LBP) and MaxPooling (See glossary). Their best performing model (Chart 3) used LBP features for training SVM and KNN. The authors reduced the dimensions through layers of Convolutional Neural Networks (CNN)

Input image	→	CNN	→	2x2 MaxPooling	→	CNN	→	2x2 MaxPooling	→	LBP
← Section in the Tensor Flow platform →										

Chart 3 – Final model combination

Although this approach requires little domain knowledge, it is complex and needs much in the way of resources and understanding of CNN.

MODEL SELECTION

"If linear regression was a Toyota Camry, then gradient boosting would be a UH-60 Blackhawk Helicopter. A particular implementation of gradient boosting, XGBoost, is consistently used to win machine learning competitions on Kaggle." - Ben Gorman [22]

Because of its well-documented effectiveness and growing ubiquity, XGBoost was the model type chosen as the first classifier to be trained on the dataset. They and their Hyperparameters can be reviewed in the appendix.
Before we explore the "Extreme" version, we shall look at the standard Gradient Boosting Machines (GBMs). GBMs are an ensemble method which iteratively build upon previous estimates, summing complex models by fitting them on top of sub-models which have been previously fit. This fitting is on residual errors to improve the previous model's fit on the samples which are still incorrectly classified.

GBMs use a "pseudo gradient" or direction of quickest improvement. This pseudo gradient is the derivative of a general loss function. A sub-learner (usually decision tree) which is as close to the pseudo-gradient as possible is added to the next iteration of the model. This new addition can often have varying weighting on different sub-portions of the trees. GBMs have strong *shrinkage* and *regularisation* properties which simplify complexity and reduce overfitting respectively. Regularisation can be represented mathematically as

$$f_t(x) = w_{q(x)}, w \in R^t, q : R^d \longrightarrow \{1, 2, \ldots T\}$$

where w is the vector of scores on leaves, q is a function assigning each data point to the corresponding leaf, and T is the number of leaves. This means it is more likely that a good model in training will be good model in testing, A point we will come back to.

Extreme Gradient Boosting is a development upon gradient boosting.
The reasons XGboost is a very popular package for machine learning competitions are

- It can approximate non-linear transformations and subtle interactions in the feature space.
- XGBoost is good at the learning tree structure
- XGBoost is optimised to reduce overfitting
- It is optimised for multi class classification.
- It automatically ascribes missing values.

Like GBM's, XGboost's algorithms take the best score from each branch of the previous classification and incorporate it into the next iteration to improve the model's overall performance. XGboost will calculate the complexity of a tree's structure to understand the Regularization, therefore defining the tree's complexity. Once XGboost has calculated the complexity of a tree it then enumerates its structure to have an addressable index of leaves.

To find the best objective function of a model, two aspects must be combined, *the training loss function* (measure of *predictability*) and a *regularization term* (measure of complexity) to help with overfitting and create a model with simplicity and improved predictability. Mathematically this is where XGBoost improves on GBMs as it allows for the choice L1 and L2 regularizations .

L1 *regularization*- **Ridge regression** adds "*squared magnitude*" of the coefficient as the penalty term to the loss function.

L2 *regularization* - **Lasso Regression** (Least Absolute Shrinkage and Selection Operator) adds "*absolute value of magnitude*" of the coefficient as a penalty term to the loss function.

The key difference between these techniques is that Lasso shrinks the less important feature's coefficient to zero thereby, removing some features altogether. This works well for feature selection in case we have a huge number of features.

XGboost contains three main algorithms

1) Basic approximation of the global splitting points

2) exact greedy for local splitting points

3) identify splitting points within sparse data

XGboost system design also contributes to its efficiency with a few non-traditional optimisations:

out-of-core processing,

Parallelized learning of data blocks,

Cache Aware access

Data Sparsity Awareness in the algorithms

Out-of core processing is an optimisation technique to take advantage of the computer system's resources. Data is divided into multiple blocks and stored on the disk. To reduce the cost of sorting through the data, blocks are structured in the compressed column format (CSC), requiring the data layout to only be computed once before training. It can then be used in later iterations.

Parallelized learning with the computer's core processors can then be automatically exercised over the resulting statistics of each processed block of data. This block structure also helps in the approximation algorithms.

Cache aware access model is to make read/write to disk faster. The cache access awareness is a prefetching approach to reduce the read/write dependency overhead. In figure 1 we can see how the cache aware access model improves the time with an insurance claim dataset and the Higgs Bosun Collider dataset, both with 1 million and 10 million data samples respectively.

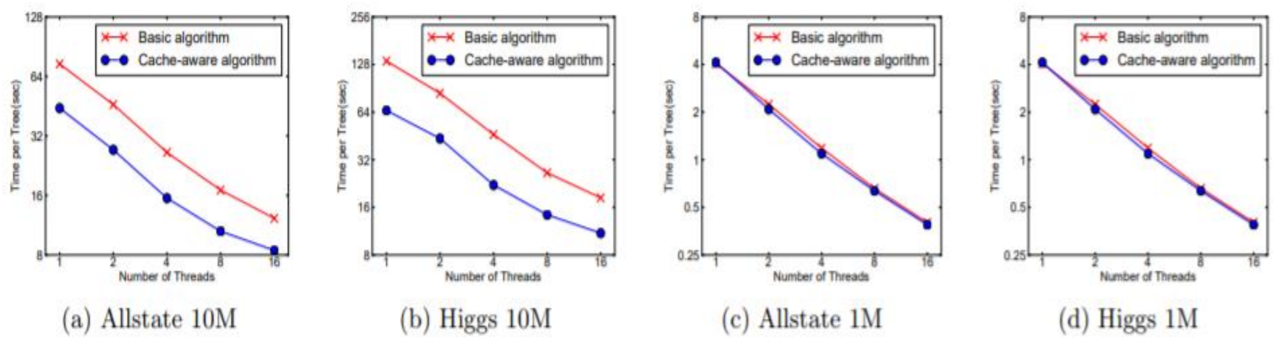
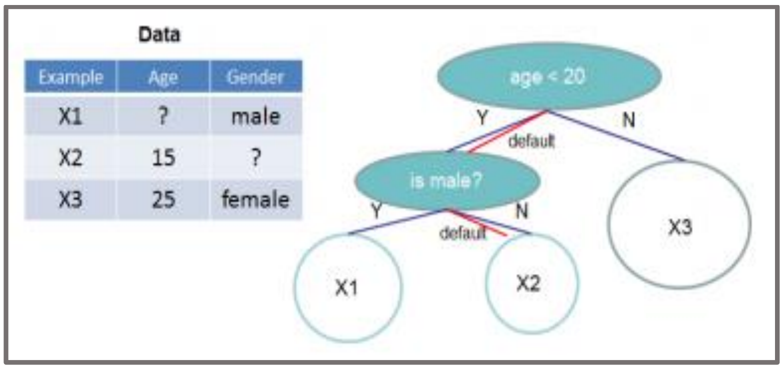


Figure 2– Cache-Aware Access model results [2]

Sparsity awareness

Sparsity in data could be caused by things such as missing or null values. To make the algorithm aware of sparsity in a dataset XGboost will add a default direction in each tree node such as illustrated in figure 2 which improves efficiency.



Data

Example	Age	Gender
X1	?	male
X2	15	?
X3	25	female

Figure 3: Default Decision Making with sparse data [1]

Feature selection

To derive meaning from the .asm files, building a parser to understand the grammatical structure of the samples would be preferable. This was explored; however, it was realised that to build a parser would not be practical considering time and skill level required, therefore more accessible and immediate features were chosen.

Feature name	Definition
Opcode counts	The number of times an opcode appears in a .asm file sample
File size	The size of the file in bytes appears in a .asm file sample
Character counts	The count of chars in a .asm file sample
Whitespaces	Count of total white spaces in a .asm file sample

Opcode counts were chosen as their occurrence in a file may allude to a malware family's certain style of behaviour. malware of the same class will have the same behaviour and so need similar instruction. The similarities of the opcode counts could be picked up by a classifier model. Santos et al [4], Liu et al [5], Drew et al [8] all used opcodes of differing n-grams being the motivation for this feature. Pennebaker et al [7] also used single occurrences of words in his classifications

File size was chosen as the size of the instructions it takes to execute a class of malware would have some similarity and so would allude to a similarity between malware families. Ahmadi et al [9] used this feature within his experiments.

Character counts in the .asm file were chosen as a non-domain feature which may be linked to obfuscation from polymorphic additions to the functioning of the malware, adding extra unnecessary data. The assumption was that the obfuscation through polymorphism will carry a uniformity within a class.

Whitespaces were measured as a feature of obfuscation as based in the elementary assumption that a packer would add complexity and therefore density to a malware sample. The white spaces could be used to signify the density of the code.

File name was taken as a unique identifier for comparison to the class labels for matching later.

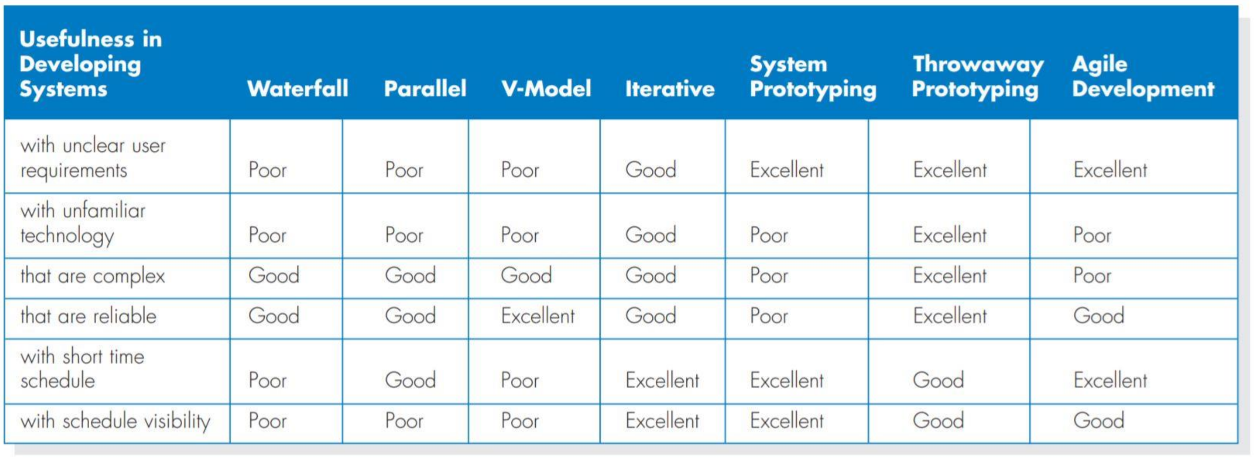
Based on the research and for use in this project the hypothesis is *:*

Is Extreme Gradient Boosting able to positively classify malware samples based on content-based features and 1-gram opcode counts.

This will be measured by the accuracy of the classification to a p-value of 0.5

DEVELOPMENT METHODOLOGY

The chart below taken from Dennis [11] highlights the criteria which were considered for selecting this project's methodology.



Usefulness in Developing Systems	Waterfall	Parallel	V-Model	Iterative	System Prototyping	Throwaway Prototyping	Agile Development
with unclear user requirements	Poor	Poor	Poor	Good	Excellent	Excellent	Excellent
with unfamiliar technology	Poor	Poor	Poor	Good	Poor	Excellent	Poor
that are complex	Good	Good	Good	Good	Poor	Excellent	Poor
that are reliable	Good	Good	Excellent	Good	Poor	Excellent	Good
with short time schedule	Poor	Good	Poor	Excellent	Excellent	Good	Excellent
with schedule visibility	Poor	Poor	Poor	Excellent	Excellent	Good	Good

Chart 4 – Development Methodology criteria by Dennis

The original development methodology, Agile, was changed to that of Prototyping, with the additional aspects of the Agile class owing to the time frame , level of familiarity with the technology, the realised complexity of the problem and the need for reliability to meet the deadline .

In Agile development, a **stand up** is a daily brief by all 6 members of a team informing all on their tasks for the day. Although I was a single person team, a planning meeting each day was held to focus on the day's tasks ahead.

A **sprint** is a distinctive step, usually 30 days of a project, which ends with a short review. The project aimed to take 2 sprints: Planning and Research, and then Implementation and Report writing.

Planning

From the Initial proposal, the planned stages of this project were to spend 1.5 weeks on research, 0.5 weeks on design, 3.5 weeks on implementation and testing, and 2 weeks on writing the report. The assumption was that having a clear structure would help keep the project on track for the submission date; additional time was added to the implementation stage based on current experience. This time frame was generally adhered to. The Scope of a project is an aspect which can "creep away" quite easily. It was therefore important that the requirements of the project were kept tight and focused. It was also important that objectives were simple and realistic to achieve the aims of the exercise.

Normally, staffing a project can determine the number of people to be assigned, matching skills with the needs of the objectives and managing potential conflicts. A normal deliverable is a staffing plan which would describe the number and kinds of people working on the project and their skills. However, in this circumstance of one person, it was irrelevant. To keep the project's time well managed, notes were written in a log. This listed obstacles, or developments for the next day, which kept the focus of the project and momentum through the separate stages as planned from research to development.

Commercial research taken in the proposal stage of the project showed there was no standalone tool which would classify a decompiled malware sample based on the proposed feature set. Some tools would compare the hash signatures of known malware samples via an SVM model, or via behavioural Heuristics. None were found based on the opcode occurrences and static features.

In this iteration as a standalone tool, the scope of Use-case is narrow. The software would need to be able to identify an .asm file only. That file will be deconstructed in to the feature space and run through a pre-trained model for classification. Results would need to be presented on the screen for the user to read. Abuse-cases were also important to consider so as the software would remain stable under mis-use.

Architecture

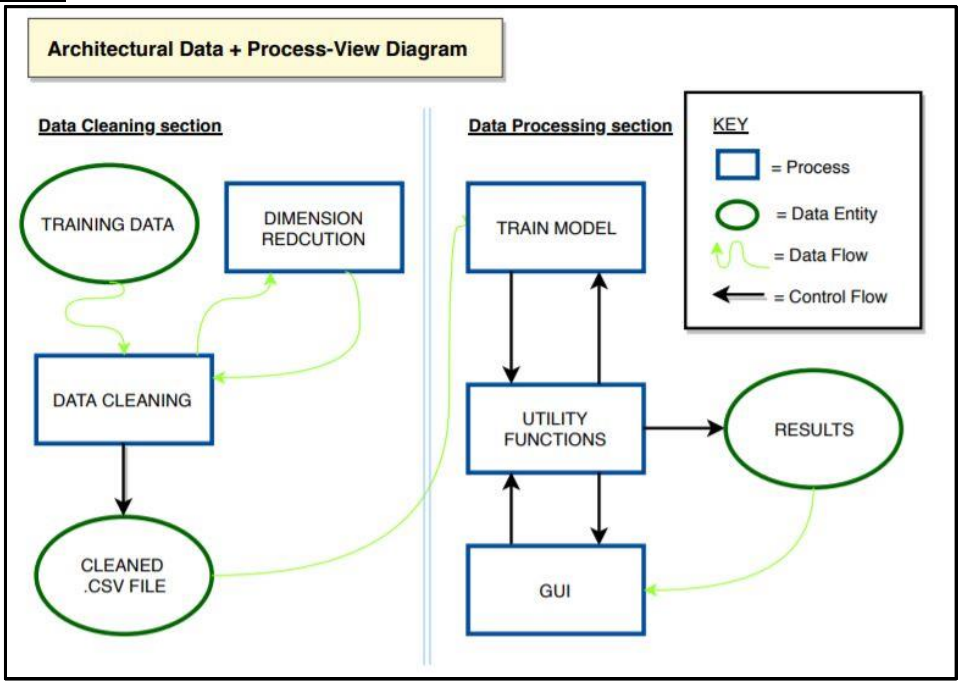


Diagram 1 - Data and Process view diagram.

The architecture consists of two sections on one level: Data Cleaning and the Data Processing. The two are only connected by their use of the cleaned data file. Maintainability and robustness of the software mainly influenced the architecture. Over time, as feature selections evolve, the models which would be trained on that feature style would need to be retrained and replaced. Therefore, a modular architecture was used to be robust with low exception handling, convenient for software testing and editing in the future.

Software Requirements

Requirements were derived through creative reflection of the problem. The module is required to read an .asm file in the right encoding. It would then pass through each line of the assembly code, splitting the encoded string into sub-strings to identify features. If the module identifies a substring which is a positive match it will add that to the count of that type of feature. The 2nd step is to run these results through a pre-trained model to attempt a prediction of the sample's class. The User out-put is a percentage likelihood of the sample's family, displayed upon a GUI.

Requirements definitions data cleaning for model training stage				
Number	**Description**	**Type**	**Nice-to-have OR Must-have**	**Was met (Y/N)**
1a	The module will be able to run through a folder and open .asm file types until exhaustion.	Functional	Must-have	Yes
1b	The module will be able to read through a .asm file and split the code on each line into strings for further processing.	Functional	Must-have	Yes
1c	The module will be able to identify a substring as an opcode from a predefined set of Opcodes.	Functional	Must-have	Yes
1d	The module will calculate a ratio of Opcode-to-Lines-of-code.	Functional	Must-have	No
1e	The module will write training statistic results for malware scores to a data file to a standalone output.	Functional	Must-have	Yes
1f	The user will be able to identify either a file or folder for processing.	Non-Functional	Nice-to-have	No
1g	The module will be able to recognise a folder from a file.	Functional	Nice-to-have	Yes
1h	The data cleaner will only recognise .asm files.	Functional	Nice-to-have	Yes
	Classification stage			
2a	The module will apply an XGBoost classification model to the Results File.	Functional	Must-have	Yes
2b	The module will output an Error rate value of the prediction accuracy	Functional	Must have	Yes
2c	The module will apply an SVM classification	Functional	Nice-to-have	no

	model to the Results File.			
2d	The module will apply an KNN classification model to the Results File.	Functional	Nice-to-have	no
2e	The module will apply a TREE classification model to the Results File.	Functional	Nice-to-have	no
2f	The module will apply an GBM classification model to the Results File.	Functional	Nice-to-have	no
2g	The classifier cleaner will only recognise .asm files.	Functional	Nice-to-have	Yes
2h	The file selection will be presented in a graphical user interface.	Functional	Must have	Yes
2i	The output statistics will be presented to the user via a graphical user interface.	Functional	Must have	Yes
2j	The user can load up and select trained models to the software.	Non-Functional	Nice-to-have	No
GUI Requirements				
3a	Only .asm files can be input.	Functional	Must have	Yes
3b	The Run button can only be pressed a single time per classification attempt to avoid error.	Non-Functional	Nice-to-have	Yes
3c	The open button will be made inactive once an .asm file is loaded.	Non-Functional	Nice-to-have	Yes
3d	File name being classified will be displayed on the GUI.	Non-Functional	Nice-to-have	Yes
3e	The clear button will reset the GUI display and its objects back to a default.	Non-Functional	Nice-to-have	Yes

Chart 5 – Requirement definitions, type, Must-have or Nice-to-have and if satisfied

Use and Abuse Cases

The main use case is that the user would

a) Open the software
b) Select a file
c) Run the file on the classifier model
d) Read the output on the display
e) Clear the screen to try another file.

Abuses Cases

Abuses Cases for the software were considered from the following questions:

1) *How can the system protect against wrong inputs?*

   Proposal: Restrict the file input to .asm files only.

2) *How can the system's buttons be regulated against abusive over running?*

   Proposal: In the first instance of a button being pressed and its function being called, the button is disabled until the function has finished and re-activates the button.

3) H*ow can the Software prevent wrong model files being uploaded to the software?*

   Proposal: Verify the input type against a list of known extensions in advance of running the model.

4) *How can System Ports be secured when sending automated email notifications?*

   Proposal: Close the port after each email notification.

As well as the GUI, the Software combined a group of functions to execute. These are listed in the chart below with Name, Action and where it is used.

Function name	Action	Used by
trainLabels	Returns a dictionary of the training labels of the Training data.	Data Cleaning
MalwareStatsGenerator	Returns a Dictionary of all features and their counts from a sample file as a key value pair.	Data Cleaning
Data cleaner	Builds a Csv file of all the feature Stats of a chosen training dataset.	Data Cleaning
*Notify**	Sends an email containing a user message	Model training
OpCodeReference	Extracts the features from a .csv list.	Data Cleaning/ Data Processing
GetFileName	Returns the name of a file.	Data Cleaning/ Data Processing
GetFileSize	Returns the size of a file in bytes.	Data Cleaning/ Data Processing
GetWhiteSpaces	Returns the count of white spaces in a file.	Data Cleaning/ Data Processing
GetTotalChars	Returns the count of total chars in a file.	Data Cleaning/ Data Processing
OpCodeReferenceDICT	Builds and returns a Dictionary of Features with	Data Cleaning/ Data

	0 counts	Processing
ASM_SampleToPredict	Takes a csv file and trained XGboost model and outputs a class prediction.	Data Processing
reportWriter	Writes a txt report on the best parameters of a Grid search CV of a model.	Data Cleaning/Model training
Time measurments	Returns time measurements from a "start" to a "finish".	Data Cleaning/ Data Processing /Model training

*Chart 6 – Functions, actions and used by – page 24 (**more details can be found in the appendix.)

DATA PREPERATION

Cleaning the dataset

The Data and control flow Diagram of the *Data Cleaner* below shows the steps of the process.

Step 1 – Initiate 2 functions, *TrainLables* and *OpcodeREF*.

Step 2 - Generate 3 Dictionaries: Training labels, Opcode Reference, Sample Statistics

Step 3 – Create an Output .csv file to write features

Step 4 – Write header to the Output file from OpcodeREF

Step 5 – Loop round the Training data deriving each file's features

Step 6 – Close Output file

Step 7 – Send email notification when done

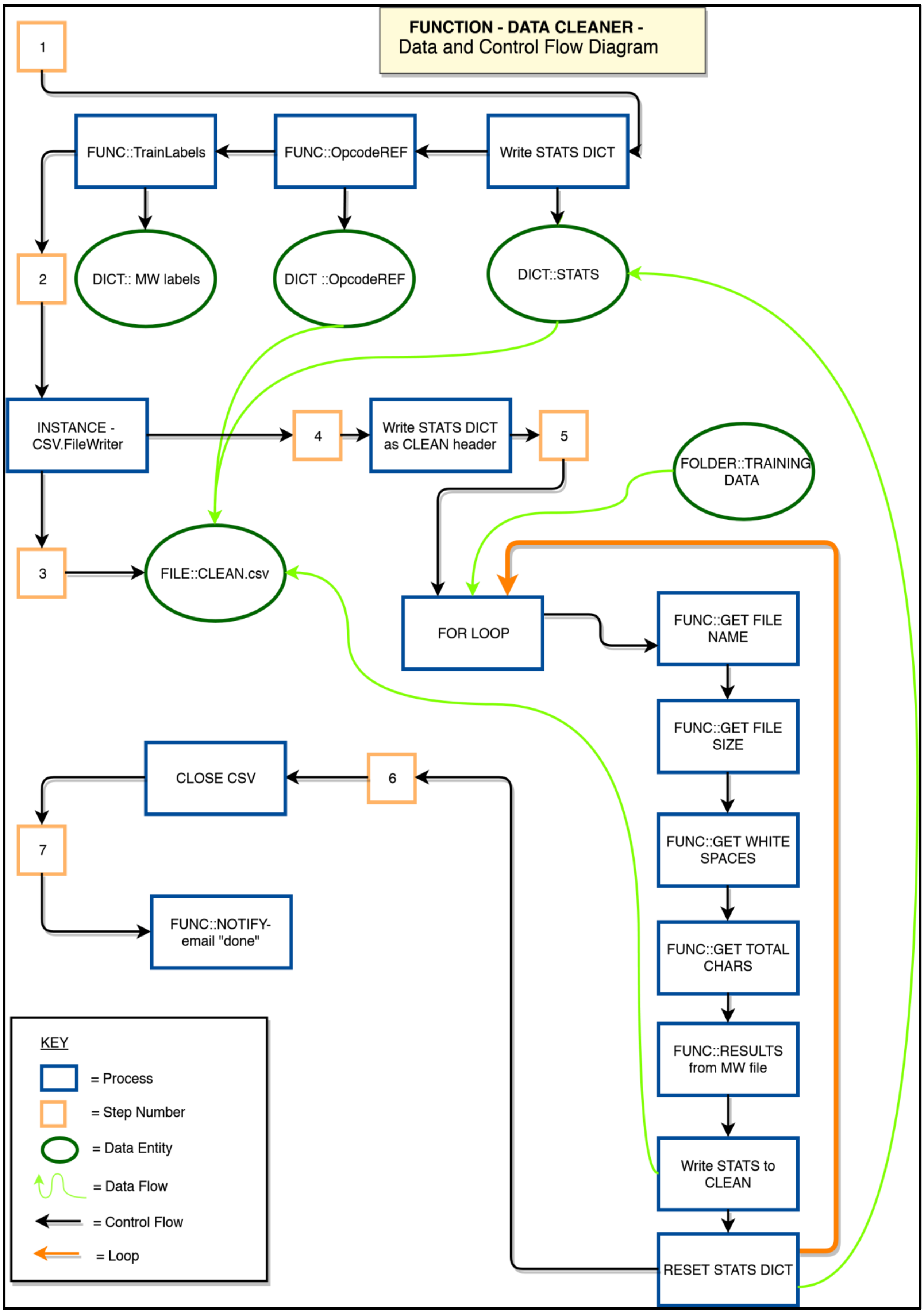


Diagram 2 - Data and Control Flow diagram

Building the training data file

The most comprehensive list of X86 Opcodes found [17] was used as the master reference set containing 657 different opcodes. These, along with the afore mentioned features and the malware class numbers, were included in the x86_OpCodes.csv file as the feature set of 663 (Opcodes, file size etc).

The Python *os* module was used in the GetFileName function to create a Python list of sample names as strings for iteration. Then features would be extracted from each sample and written to a Python Dictionary called *Reference Dictionary*. A Dictionary was chosen for its speed of access and processing, and order, as opposed to other data types such as a Set or List which can be slower. Each key would be a feature derived from the features.csv file, and values would be counts associated with each file (opcode frequency, file size, etc).

Malware Statistics Generation

The *MalwareStatsGenerator* function would take a *Target File* name, *Reference Dictionary, White Space Count and File Size* as parameter inputs and return a dictionary of features [Keys], and their integer counts [values]. It would also contain a *filename* as a text string key value pair.

To not corrupt the reference dictionary a copy *Return Dictionary* would be initialised by going through the R*eturn Dictionary* and copying every Key from the reference with a default value of 0. However, it is now apparent that this could be more elegant with a single duplication of the *Return Dictionary* at each calling of the function.

The initial step was to open the sample to simply read out via a print statement. This proved problematic as the encoding format was not recognised. After some searching, (ANSI) encoding was used. Using this as the encoding parameter on the Open function allowed the file to be to be read in a text format. As each line was read by the first level of *for loop,* it was split by its blank spaces into separate substrings. In the nested *for loop,* if a substring matched a feature key from the Reference Dictionary, its value in the reference dictionary would be incremented by 1. Once all lines had been read from the file, it would be closed, and the Return Dictionary would be returned as the output.

Principle Component Analysis (PCA)

Many sample file feature counts had zeros. In fact, the training dataset consisted of 88.5% zero values which is quite sparse. To be able to know the principle components of each malware class in a legible way, a standalone programme was developed to conduct Principle Component Analysis (PCA) on each malware family. This would indicate any features which are the most significant to that class, those that were not and any which were significant across the entire dataset, allowing for the removal of features which have little statistical significance. In this project scaling the data is necessary as some features were counted in the thousands, where as some had counts of 1. The maxabs_scale value was chosen as this sciKlearn pre-processing method is designed to perform well on sparse data such as this.

Exploration was taken to find the features which could explain as close to 100% of the variance in the dataset as possible. This is to reduce its features (dimensions) down from 663 features to a smaller amount which would increase the efficiency of computational processing, model training and prediction later. This stage would have two steps; first, to sum all of the explained variance from the top score features to as close as 100% of the sample set; and secondly, to plot the data on a scree plot [see appendix] to look for the point where the amount of variance explained reduces severely forming an elbow in the line plotted.

Starting from 2 features, the number was incremented, in some cases up to 40 features until the output came close to our variance explained target. Higher than this was not considered a reduction in the dimensions and so a limit was set there. The least amount of variance which could be explained by any one class was 73%; all other cases were over 80% with more than half over 90%. Moreover, as variance is a square value of the true deviation from the mean, the features would be smaller still and potentially statistically insignificant for model training.

From the scree plots an elbow can be derived in some cases, meaning that some features do explain more variance than others. However, overall a small amount is explained by the wide angle of the elbows and a maximum recorded of just 2.7. At this stage it would be prudent to consider that the feature selection could have statistical insignificance as in nearly half of families, the P-value is kept below 0.05. Nevertheless, features with the higher value variance above the elbows were identified and labelled as the principle features, allowing for a subset of training data to be created so that model training and classification could still be explored. Statistical models may still be able to create a statistically significant classification accuracy based on this feature set. After removing duplicate PCA's and features which explained no variance, training could take place on the new dimensionally reduced dataset. The dataset was reduced from 663 to 39 features. Below is a chart of the resulting features for each family.

It was interesting to see that both families 5 and 7 included features which were not opcodes, possibly aligning with Ahmadi et al [9] . Although their sample size as a class was smaller than many others, this may point to this style of feature having statistical importance in these families. It was also encouraging that few opcodes occurred twice between families indicating a difference between families' use of opcodes and may align with Bilar's [14] discovery of rare opcodes having importance in distinguishing samples.

Name	1.Ramnit	2. Lollipop	3.Kelihos_ver3	4. Vundo	5. Simda	6. Tracur	7.Kelihos_ver1	8.Obfuscator.ACY	9. Gatak
Type	(WORM)	(ADWARE)	(BOTNET)	(TROJAN)	(KEYLOGGER)	(TROJAN ROOT KIT)	(BOTNET)	(SPYWARE)	(TROJAN)
OPCODES	**'cvttsd2si'**	**'setnle'**	**'popa'**	**'bswap'**	**'movsb'**	**'setl'**	**'retf'**	**'stos'**	**'ja'**
	'fnstenv'	**'fxam'**	**'rep'**	**'lea'**	**'lock'**	**'fcmovnu'**	**'bsr'**	**'fs'**	**'leave'**
	'fldenv'	**'setl'**	**'xchg'**	**'mul'**	**'totalchars'**	**'movsw'**	**'lodsd'**	**'fninit'**	**'adc'**
	'fxam'	**'setle'**	**'jmp'**	**'movsx'**	**'file size'**	**'rol'**	**'das'**	**'ffreep'**	**'sbb'**
	'fs'	**'cmovbe'**	**'es'**	**'ror'**	**'ds'**	**'or'**	**'totalchars'**	**'fstp9'**	**'imul'**

Chart 8 - Features after PCA - Blue is not an opcode - Red occurs in < 1 Malware type

TRAINING MODELS

A Python programme template (Diagram 3) was created which could be used to train models additional to XGboost, such as SVM and GBM models which were experimented with (see appendix) to satisfy aim 8 from the introduction. Implementation would be straightforward in an improved version, ideally to satisfy requirement 2j (loading up models). In this section we walk through how the model training was executed in this instance and how model training can be automated using the GridsearchCV library.

First steps of model training

The performance of the main resource (the computer) being used was of a modest specification (2.4 GHz 12 GB RAM). The efficiency of speed, size and data conversion were important considerations for tasks which involved high processing. Now that the training data had been cleaned and its dimensions reduced, the first step was to change the working format of the input data for simpler, faster processing via a NumPy ndarray. This was more compatible with the various library functions used in training statistical models in this section as oppose to working with a .csv file.

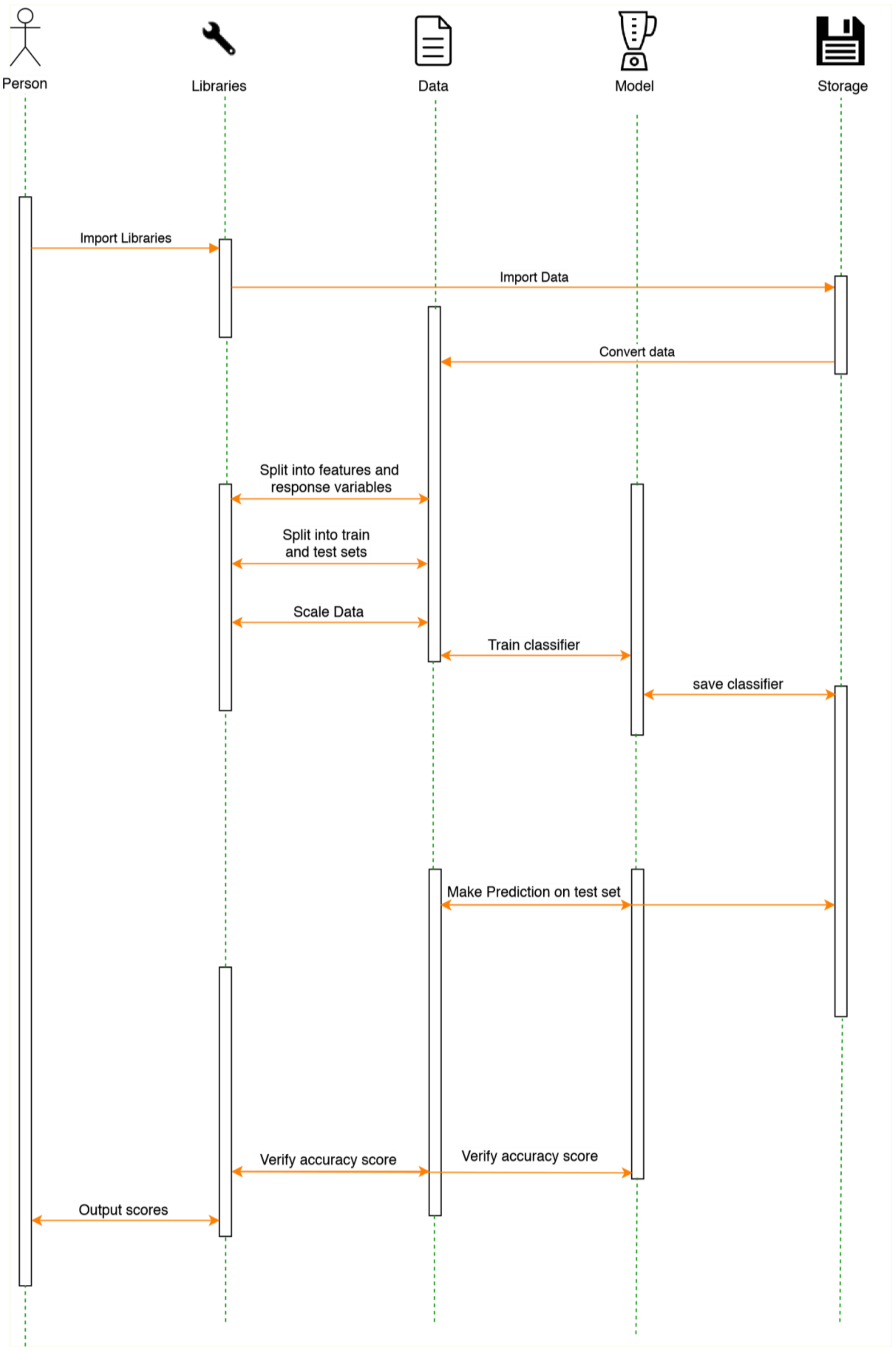
The dataset used for training was still at scale even though features had been removed during dimensionality reduction. This meant that the first step in the training model was to scale down the dataset to normalise the data values.

With hindsight it became apparent that not just training models in Python could be stored with the library Pickle but also dataframes. Storing data structures of great size could come in useful in further versions, when the feature selection is broader with multi-gram opcode counts.

The target variable was the labelled column for each data sample’s class number in the training data. To proceed with the target variable, data had to be reduced to a 1-dimensional matrix. Simple column selection was not enough and failed to process. This challenge was overcome by flattening the matrix using the ndarry method Flatten to convert to a matrix of 1 dimension.

To find the best performing selection of parameters for each statistical model *GridSearchCV* from the Sklearn model selection library was used. This removed the

Diagram 3 – Model training sequence diagram



need of manual searching for the optimum parameter set of a model by automating the parameter testing along all permutations. This process was still intensive for the computer (in some cases over 36 hours to explore its options), however it could be left running throughout the night, with an email notification sent when compete.

GridSearchCV's arguments in this instance were firstly the *Classifier (*XGboost *etc)*. Second a Python dictionary of parameters with *keys* as parameter string names and *values* as parameter values to create the *Grid*. Thirdly a cross validation or *CV* integer value used to set the split amount (CV= 5 for a train test split of 80/20) to test which parameter set maximises the accuracy score of the current grid set of parameters. The main output of *GridSearchCV* would be the best performing trained model which could be saved as a classifier object based on the values entered in the grid. The best_params_ method could then be called to output on the trained classifier model to produce a dictionary of those best parameters [keys] and their settings [values]. With this output of the best performing parameters, further Iterations of *GridsearchCV* parameter training could take place on a more refined selection. As an arbitrary example, values of Cost could be selected initially at (0,10,100,1000), once 10 has been identified as the best value, (9,10,11) could then been chosen for the next training iteration, continuing to (10.1,10.2,10.3….) in the third. The resultant XGboost parameters can be seen in the appendix.

Software testing

The software was tested on three levels throughout development, always after a piece of code's development was finished.

Unit tests refers to Function/Object/Methods and were important to prevent errors later in the process, especially at higher levels of the software. They would be carried out by running the Unit on each new edit, even after name changes to be confident that

a) Units were functioning without a fatal error

b) the correct output format, and messaging - e.g. indexing in a data type.

For example, the *notify* function would be tested to make sure the email was received with the correct title of the model it was training. This would also test that the function opens a port, logs in to an email account, sends a message and logs out.

Component testing – Once a unit had been completed and the output was satisfactory it would be tested by incorporation with its related units to make sure that no unwanted emergent behaviour appeared, and the outcomes were as expected from the combination. This was less frequent but equally important, again to prevent greater errors later in the system. As an example, the *ASM_SampleToPredict* was tested in this way. It needs to call multiple functions to operate and create several data structures which were needed to be robust and accurate in different processes.

System testing – Once the system had been completed, it was tested in unison with a small dummy data fragment. This is a scale up from the Component testing to make sure that unwanted behaviour or outputs would not appear. As an example, the GUI would be tested for Abuse Case:2 the run button being repeatedly pressed.

Prediction

With the best parameters selected and the model trained to its optimum, the model is stored for use later. The *Pickle* library offers the functionality for this. *Pickle* implements binary protocols for serializing and de-serializing a Python object. *"Pickling"* is the process of a Python object being converted into a byte stream, and *"unpickling"* is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back and loaded up into an object for use.

To check that the model's predictions were accurate within themselves, the CV accuracy score of the model CV was found by using the *cross_val_score* function from the sklearn.model_selection library. Its arguments were: the trained and pickled model; the training data; the target variable; the CV's K value and finally the *scoring* value (in this case 'accuracy'). Different K values were tried between 5 and 10. In most models K = 8 gave the highest prediction accuracy on a data sample which was made at the last step. The output from calling this function would be a list of accuracy scores. This list's *mean* method would then be called.

The *ASM_ Sample_to_predict* function overleaf implements the prediction. It follows the steps of:

1. Building a default feature reference dictionary statsDICT
2. Calls non-opcode feature functions
3. Writing non-opcode features to the statsDICT
4. Writing the malware opcode ResultsDICT to the StatsDICT
5. Converts the statsDICT to a data frame
6. Makes a prediction of the Stats Dataframe through the trained model
7. Outputs the prediction

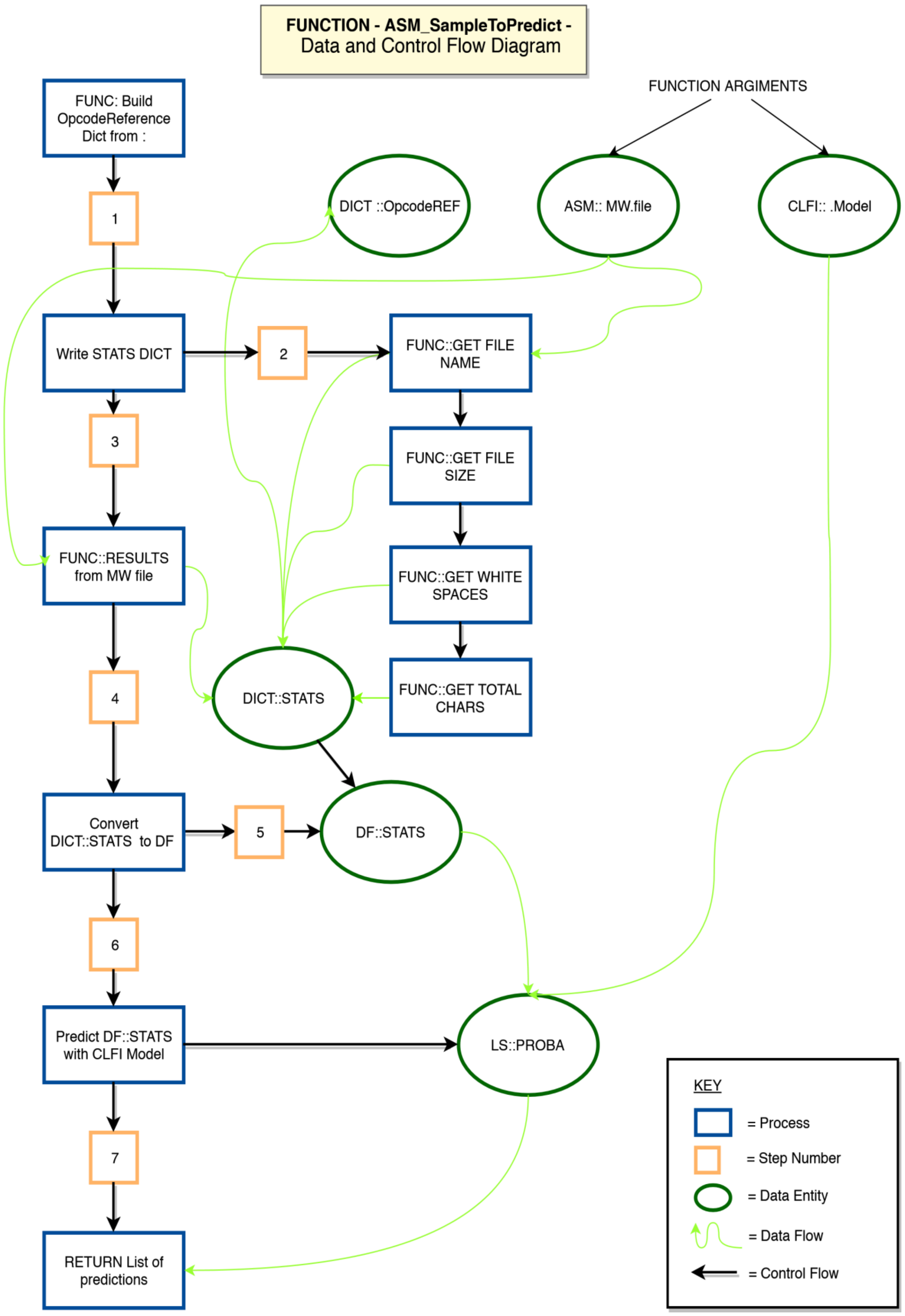
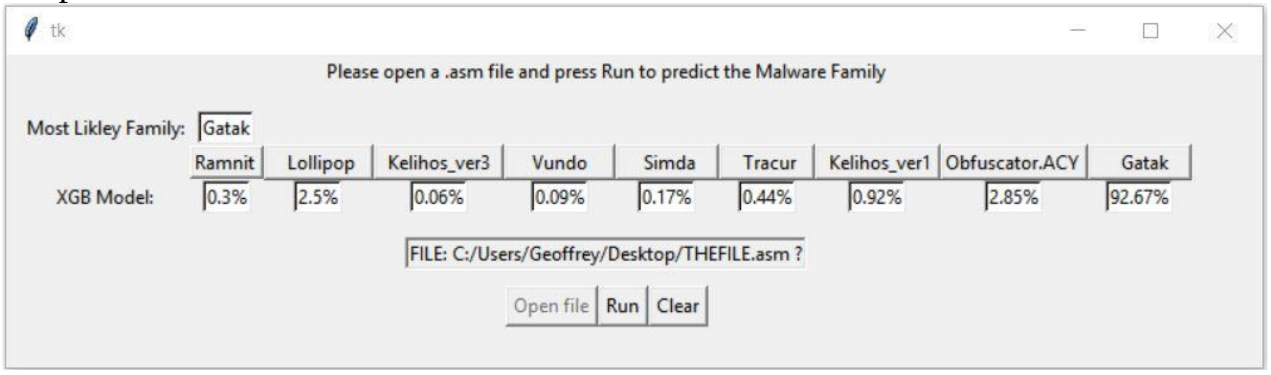


Diagram 4 - ASM_Sample to predict data and control flow diagram

Graphical User Interface – GUI



Please open a .asm file and press Run to predict the Malware Family

Most Likley Family: Gatak

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
XGB Model:	0.3%	2.5%	0.06%	0.09%	0.17%	0.44%	0.92%	2.85%	92.67%

FILE: C:/Users/Geoffrey/Desktop/THEFILE.asm ?

Image 2 - Graphical User Interface

A working view of the software can be seen above and is represented in the component diagram below. The Tkinter based GUI is a class object which initialises a control panel and display panel objects which contain buttons, labels and boxes. The Class also initialises with two data objects (file name and results). The control panel Buttons each have separate functions corresponding to their title and allow a user to Open a file, Run the sample through a trained model and clear all data objects to start over again. The Run button calls the *ASM_ Sample_to_predict* function seen in the diagram above. The buttons are coordinated to satisfy *Abuse Case 2* from the requirements, so that irrelevant buttons are deactivated when exciting a function to avoid causing error and reactivated when needed. With only three buttons and clear explanation of the outputs on the interface, writing a manual would be unnecessary.

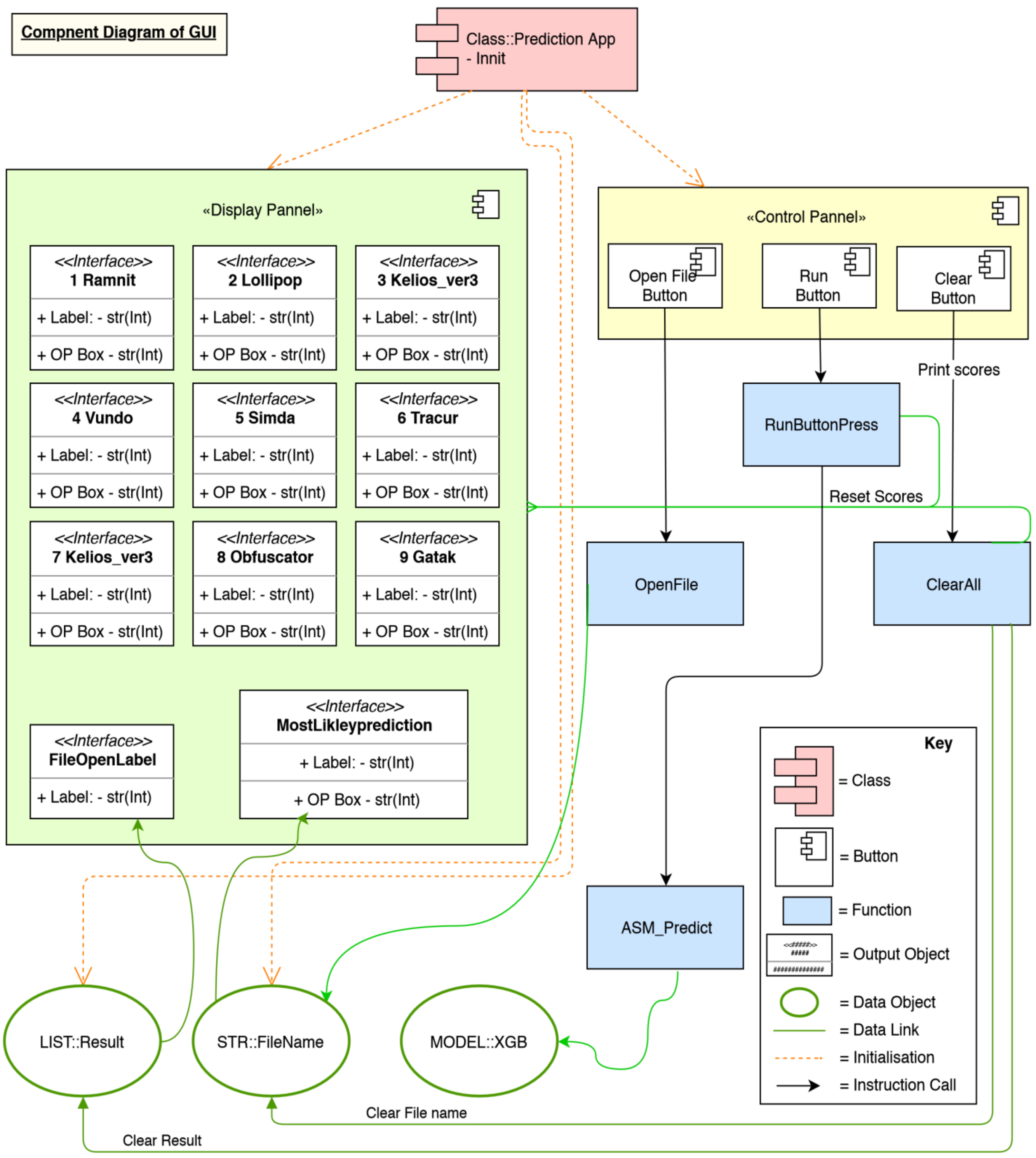


Diagram 5 – GUI components

EVALUATION

Project aims	How it was satisfied /not satisfied
Identify a dataset which contains a diverse range and healthy volume of suitable and safe malware samples.	Using the BIG15 MALWARE DATA SET of 9 unbalanced classes of Malware sample
Accurately clean and transfer a raw dataset to a robust format for use in a practical classification exercise.	The dataset was successfully cleaned to specification for later processing.
To research recent classification methods which **are via content-based features.**	This is contained in the research section
Identify and isolate the key features of the dataset through statistical measurement to optimise the classification and computational processes.	A module was written to perform PCA [PCA40.py] explaining approximately 100% variance within 40 features.
Improve upon software development skills in Python with libraries and methods that have not been covered in my current set of modules.	Production of working software and the use of the training models, Tkinter, SMTPlib, GridsearchCV, Scikitlearn, etc
Develop the skills of training and then implementing classification models in Python.	Training and tuning the XGBoost classifier and experiments with SVM,GBM,KNN, TREE. Using the Python pickel module
Work with state of the art/industry classification models to develop experience which is up to date.	Using XGBoost as the main training model
Develop the skills of building a classification framework which can implement various classifiers.	The classification file was built as a frame work **which** could use interchangeable models
Work within the spirit of a professional Project following a prescribed and identified methodology	Although initially “Agile” was chosen, it was identified that a “Prototype” would be more suitable.
Produce an easy to use and robust piece of software which is significantly less susceptible to user error or malicious use	This was satisfied as all Abuse cases were satisfied.
To work in a simple way which does not obstruct the full exercise of a software development project	This was satisfied with time frames being distinctly marked out and kept to strictly.
Contribute to a portfolio of work directed towards a career in cyber security and data science	This is evidenced by this project and report
Attempt to accurately classify malware samples by the defined features	This has still to be satisfied with further exploration below.

Table 2 – Project aims and achievements

Software Results and Classification Accuracy

Eleven of twelve of the “must have” functional requirements set out were met. The software is also defensible to all Abuse Cases. These were concluded by the testing of the software (Unit/Component/system) being carried out at each step of integration, before moving on. However, in a future revised time frame, and as software features grow, implementing a more rigorous testing framework could facilitate software improvements and evolved hypothesis testing.

The XGBoost model mis-classified all 20 malware samples from a holdout dataset. This best performing trained model which made these predictions had a CV-evaluation score of its predictive ability at 98%, viz, we can be confident that the null hypothesis cannot be rejected. Additional model training and testing were carried out on a 3-class balanced set (1000 malware samples each) of class 1,8,9. The null hypothesis could still not be rejected. The most likely contributing factor was the lack of explained variance in the feature space. Although there was a distinctive amount of variance explained in the PCA of 1-gram opcode counts [see appendix] , it suggests only enough variance to contribute within a larger group of features in the future, such as multi-gram opcodes or alongside grey-scale images. This is a flaw with the feature selection. Interestingly, PCA of each class defined a unique collection of features, a clear distinction between classes which requires further research.

If 1+ of the hold out samples were correctly classified, acceptance of the hypothesis could be considered within a < 5% p-value; however, this test set size was too small which potentially distorts the results. This is a flaw in the design of the experiment. A larger testing group of at least 2000 samples (20%) should be used in future. On the other hand, the binary “correct or incorrect” approach to testing was sound. As simulation, even 0.001% “infection” can be classified as an infection. This approach should be carried over to future work, in addition to other statistical testing, such as a multi-class logistical regression. Therefore, the approach using mainly classification via single opcodes is generally unreliable but has some merit.

Looking back to the literature review I can recognise that I could adapt the research from other results to get a better outcome. The first enquiry is to revise the model parameters and to research if underfitting has taken place. The second enquiry would then be to pickle a dataframe of 663^n gram permutations of existing features and rebuild the dataset. As well as changing the features, a dataframe can be transported around a system more elegantly than a csv file. The categorisation of opcodes would be a joint with the one above or third consideration. Both new features could give weighting because of the increased contextual richness, distinguishing samples from one another for a more accurate classification such as the methods of Drew et al (2016), Ahmadi et al (2015) and Pennebaker (2018). Categories include Arithmetic, Data copying, Logical operations, Program control and Special/Rare instructions.

Next the dimensionality reduction could be explored further with a different approach, such as Independent Component Analysis or Kernel-PCA. Benign goodware could also be incorporated as a 10th Class to create a more robust distinction between software classes. Exploration of training four comparative models (SVM, GBM, KNN and TREEs) has taken place in unmentioned stages of the development and could also be implemented to compare classification accuracies on a positive feature space.

The Conclusions

Regarding project management, on the small scale, days and hours were managed well, keeping the project on track. Of the larger stages (Table 2), time was deliberately divided in a "non-traditional" sense which could have had negative impacts. Adjustments in proportion of project timelines considered current levels of programming skill development in relation to the complexity of the research design.

	Planning	Analysis	Design	Implementation	Report
Actual proportion of work in weeks	12.5% (0.75 weeks)	12.5% (0.75 weeks)	8% (0.5 weeks)	57% (3.5 weeks)	2 weeks
Ideal proportion ccording to Dennis [11]	15% (0.9 weeks)	20% (1.2 weeks)	35% (2.1 weeks)	30% (1.8 weeks)	2 weeks

Table 3 – Project sections in time

More time reading module manuals before proceeding and beginning the research may have made improvements on the classification accuracy and software functionality, as well as the design of the experiments. However, this had to be balanced with practical application. With a bigger team, a log/diary/dashboard for "anytime" instant access would be useful so that all parties could coordinate based on where the team is in the project according to budget, week number etc.

Project reflection

"My only goal is to gradient boost over myself of yesterday. And to repeat this everyday with an unconquerable spirit." – Mike Kim, Kaggle

The perceived likelihood that malware classification through 1-gram opcodes occurrence and content-based features via an XGboost model would be achievable motivated me to proceed. This kept momentum going and morale high, which are both productive attributes for a project [26]. This focus towards success, as opposed to avoidance of the unknown, bolstered productivity. Many steps forward have been taken from the project aims as can been seen in table 2. In addition, possibilities for future research demonstrate a way forward for this line of malware identification.

The strong supposition was exercised to prevent overfitting in the model training and this may have led to underfitting instead. Future experiments will give more consideration to this supposition. Developing comprehension with the process of a Data Science classification programme is the tangible value here, and has positively impacted the practical skills which will be needed in solving similar problems in the future work place.

The assumption that managing time tightly had a positive impact on the project was correct as most aims have been met. Although the larger division of time stages was not traditional, the rigour of sticking to and managing time tightly did contribute value to timely delivery and some of the theoretical learnings. I was able to connect with an appropriate style of methodology and now feel more confident in my ability to run a project in the future.

The idea that the lack of formal training in mathematics required could be overcome by learning "on the job" was an over-confident assumption to make. It proved more time consuming than previously thought to understand concepts and to proceed. A remedial approach will be the completion of at least an A-level mathematics course in advance of further study to grasp deeply and underpin many of the mathematical, statistical and logical concepts involved in a project of this nature.

I have also recognised that a future approach should incorporate reflective practice more and define a ratio between the work/results obtained and the method of arriving there. The reason this is important is that as technology evolves, it will be a common occurrence to gather new information and review the value and effectiveness of current best practice in relation to innovation and new technology.

REFERENCES

[1] - Natraj Lakshman - 2011- **A Comparative Assessment of Malware Classification using Binary Texture Analysis and Dynamic Analysis**

[2] - Natraj Lakshman -2011 - **Malware Images: Visualization and Automatic Classification** - http://vizsec.org/files/2011/Nataraj.pdf -

[3] - Jhu-Sin Luo - 2017 - **Binary malware image classification using machine learning with local binary pattern**
- https://ieeexplore.ieee.org/document/8258512/

[4] Igor Santos - 2010 - **Opcode-sequence-based Malware Detection** - http://paginaspersonales.deusto.es/isantos/papers/2010/2010-Santos-Opcode.pdf - 2010

[5] - Jiwei Liu - 2015– **Live stream of Kaggle winners presentation** - https://www.youtube.com/watch?v=VLQTRlLGz5Y –

[6] - Jiwei Liu – 2015 - **BIG 15 winners Interview blog** - http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-place-no-to-overfitting/

[7] James W. Pennebaker - 2018 - **Linguistic Inquiry and Word Count software** - https://liwc.wpengine.com/ -

[8] Jake Drew- 2016 - **Polymorphic Malware Detection Using Sequence Classification Methods** -https://ieeexplore.ieee.org/document/7527757/

[9] Ahmadi - 2015 - **Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification** - https://arxiv.org/abs/1511.04317

[10] **Opcode Reference site** - 2018 - http://ref.x86asm.net/coder32.html

[11] A.Dennis - 2012 - **System Analysis and Design 5thed**

[12] Igor Polkovnikov 2016 - **Unified Control and Data Flow Diagrams Applied to Software Engineering and other Systems** - https://arxiv.org/ftp/arxiv/papers/1610/1610.02374.pdf

[13] A Khalilian - 2018 - **G3MD: Mining frequent opcode sub-graphs for metamorphic malware detection of existing families** - https://www.sciencedirect.com/science/article/pii/S0957417418303580

[14] D . Bilar - 2007 - **Opcodes as predictor for malware** - https://dl.acm.org/citation.cfm?id=1359312

[15] H.Zhang - 2018 - **Classification of ransomware families with machine learning based on N-gram of opcodes** - https://www.sciencedirect.com/science/article/pii/S0167739X18307325

[16] Pennebaker – 2018 - **L**inguistic **I**nquiry and **W**ord **C**ount software - https://liwc.wpengine.com/how-it-works/

[17] OPCODES list taken from - http://ref.x86asm.net/

[18] Andrew Ng - 2018 - **Machine Learning** - https://www.coursera.org/learn/machine-learning

[19] - Eric Filiol - 2018 - **Formalization and implementation aspects of *K*-ary (malicious) codes**

[20] – **Kaggle Malware Classification Competition** - https://www.kaggle.com/c/malware-classification

[21] - John McDermott and Chris Fox - 1999 - **Using Abuse Case Models for Security Requirements Analysis** - https://pdfs.semanticscholar.org/87f9/535260fee6f0f70429070f5472bffc7c35cf.pdf

[22] – Ben Gorman – 2017 - **A Kaggle Master Explains Gradient Boosting**
http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/

[23] XGBoost Parameters - 2018 - https://xgboost.readthedocs.io/en/latest/parameter.html

[24] - **AARSHAY** JAIN – 2016 - **Complete Guide to Parameter Tuning in XGBoost (with codes in Python)** - https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

[25] Crisis? What Crisis? - https://www.economist.com/science-and-technology/2018/03/17/are-research-papers-less-accurate-and-truthful-than-in-the-past

[26] Andrew Oswald – 2015 - **New study shows we work harder when we are happy**
- https://warwick.ac.uk/newsandevents/pressreleases/new_study_shows/

[27] **Learning By Thinking: How Reflection Improves Performance** - 2014 -
- https://hbswk.hbs.edu/item/learning-by-thinking-how-reflection-improves-performance

[28] Jeffry D Ullman - 2018 - **Minihashing** - https://www.youtube.com/watch?v=96WOGPUgMfw

APPENDIX

Gradient Boosting Machines **(GBMs)** – GBMs will produce an ensemble of weak predictors , that are combined based on their strongest predictions, or looked at another way, minimum Loss of error.

$$\hat{F} = {}^{min}_{F} E_{x,y}[L(y, F(x))]$$

Here $\hat{F}$ is an approximation function, E is the expected value of the loss L of the current functions F's attempt to predict y based on x.

The advantages of GBM's are:

handling of heterogeneous data

Predictive power

Resilient to outliers in the output space

GBM's do fall down with Scalability because the sequential nature of boosting prevents parallel learning .

The algorithm in a GBM will iterate on this equation when training

$$\boldsymbol{F_m(x)} = F_{m-1}(x) +^{minArg}_{h_{m\in H}} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h_m(x_i))$$

Where F_m is the current function output on that iteration, F_{m-1} is the previous function added to the minimum loss value L on the new predictor h, that is an element of the group class of learners H (those set by the parameters in Python).

To train the GBM's in Python a selection of parameters was choose to explore via GrdisearchCV. Some parameters were left out as they were not relevant for this classification problem. The main GBM Params used were -

n_estimators: number of boosting stages (trees) to perform. The more the better but 50 claimed 100%. 100 achieved 98%** Therefore a range between 35 -40 seemed best for GridsearchCV

learning rate: shrinks the contribution of each tree by learning_rate. There is a trade-off between learning_rate and n_estimators.

min_samples_split: Defines the minimum number of samples (or observations) which are required in a node to be considered for splitting. Default is 2 but 5 kept the accuracy inplace and will decrease over fitting. This should be ~0.5-1% of total values.

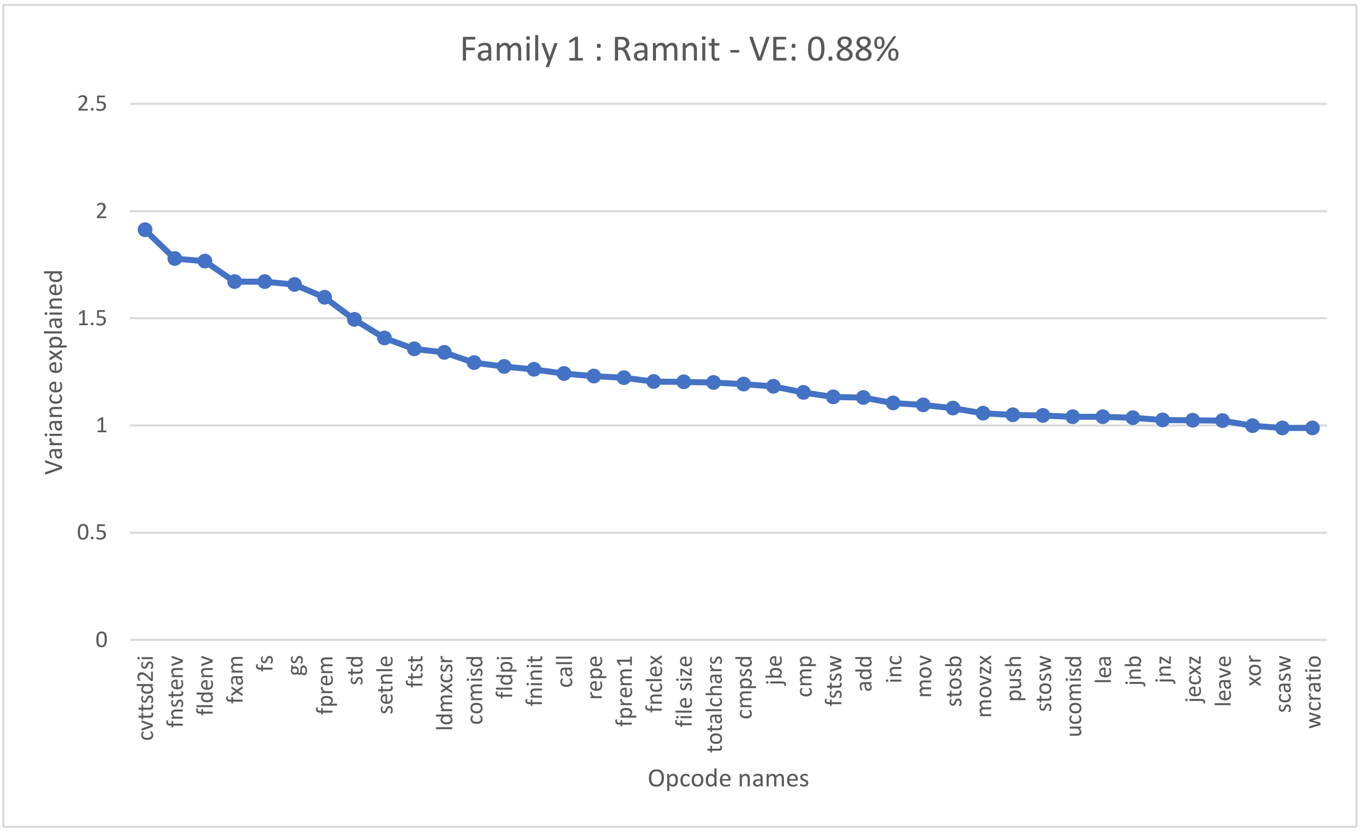
min_weight_fraction_leaf: The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. IF samples have equal weight (which they didn't) then sample_weight is not provided. 0.22 worked best between values of 0 and 0.5

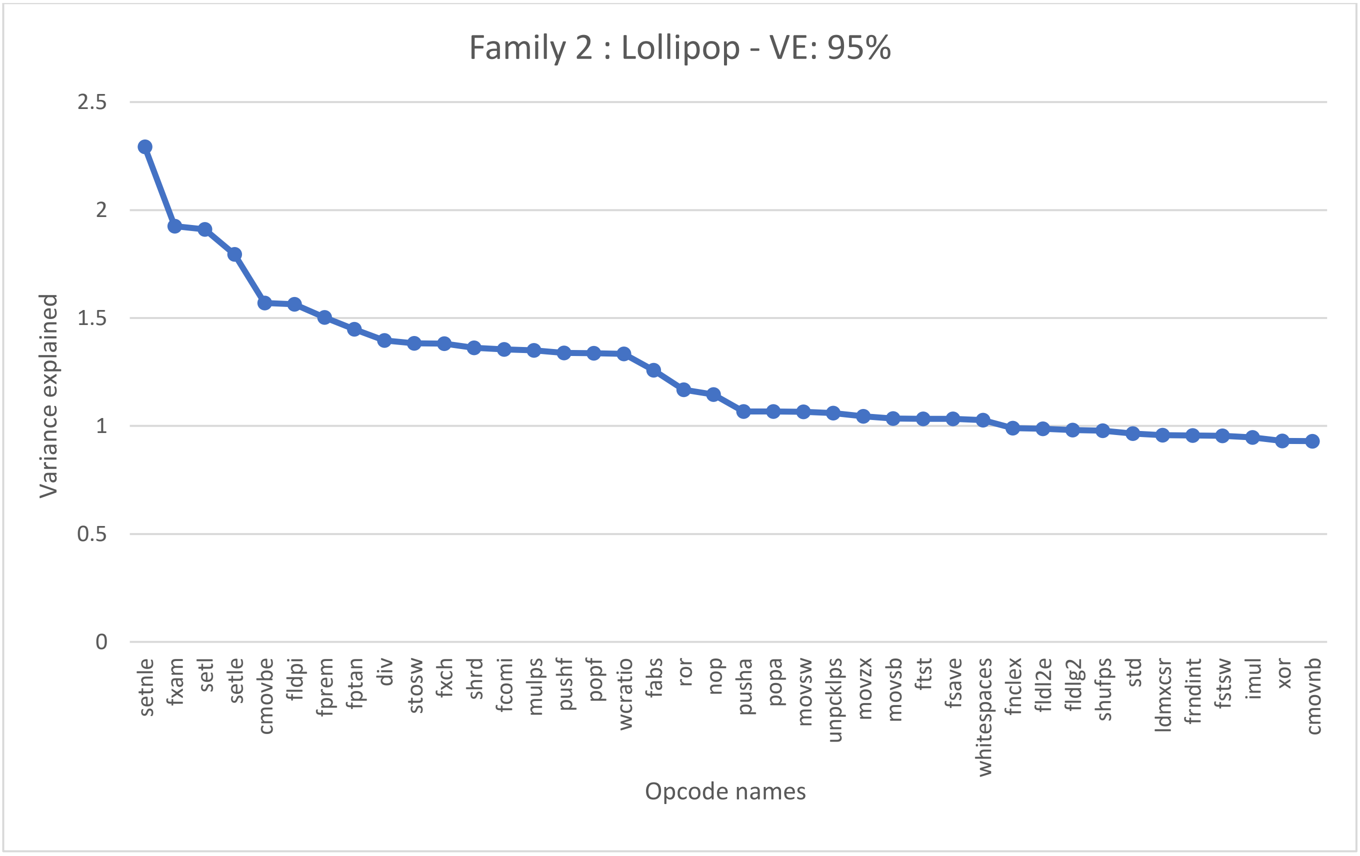
<u>Parameters excluded</u>
N/A - **Presort:** - Whether to presort the data to speed up the finding of best splits in fitting. = False
N/A - **subsample:** ideally 0.8 but adding this did not make improvements so its left out/default
N/A - **loss** . irrelevant as refers to deviance (re logistic regression)
How this looked in Python code

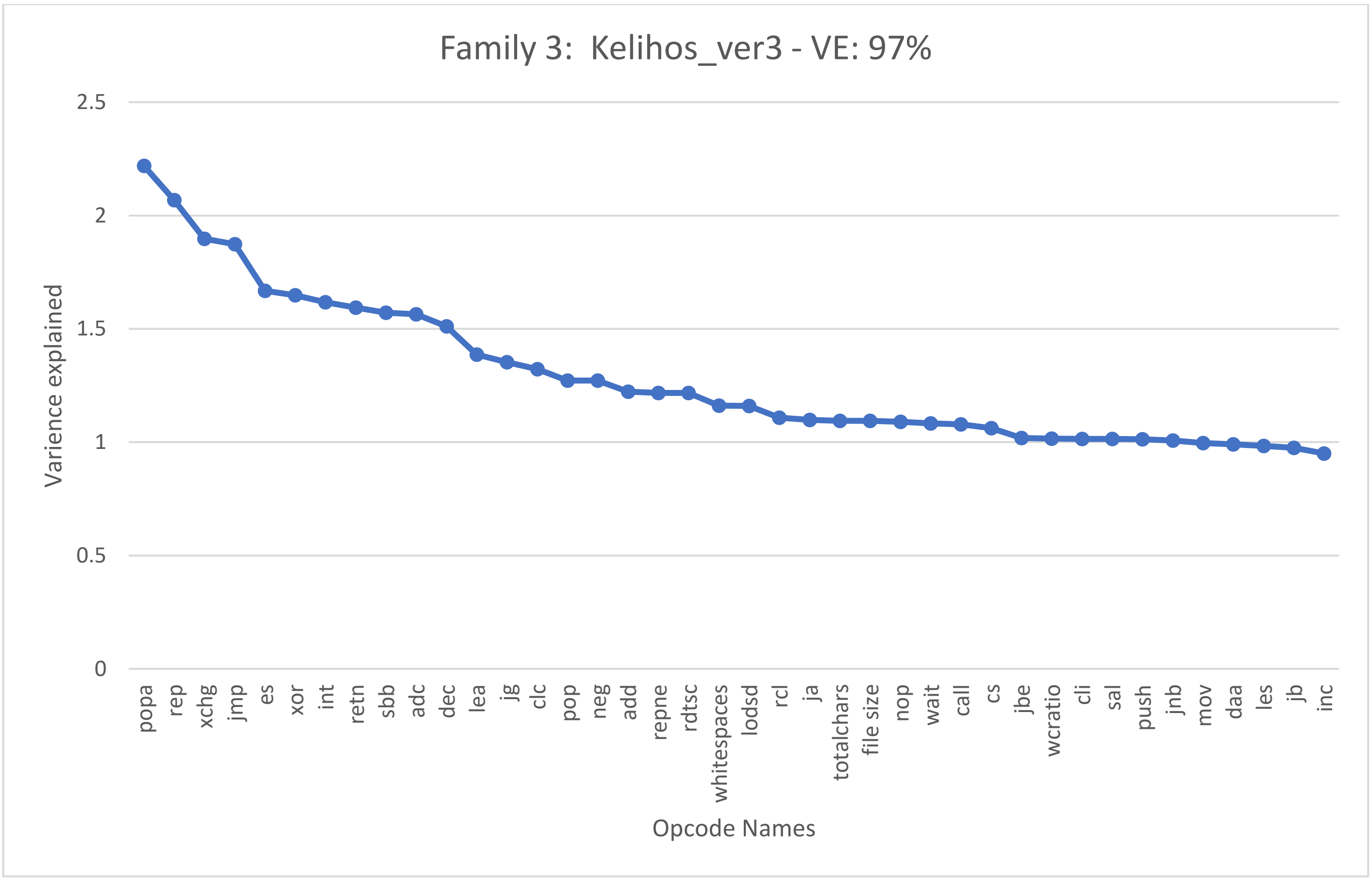
RESULTS

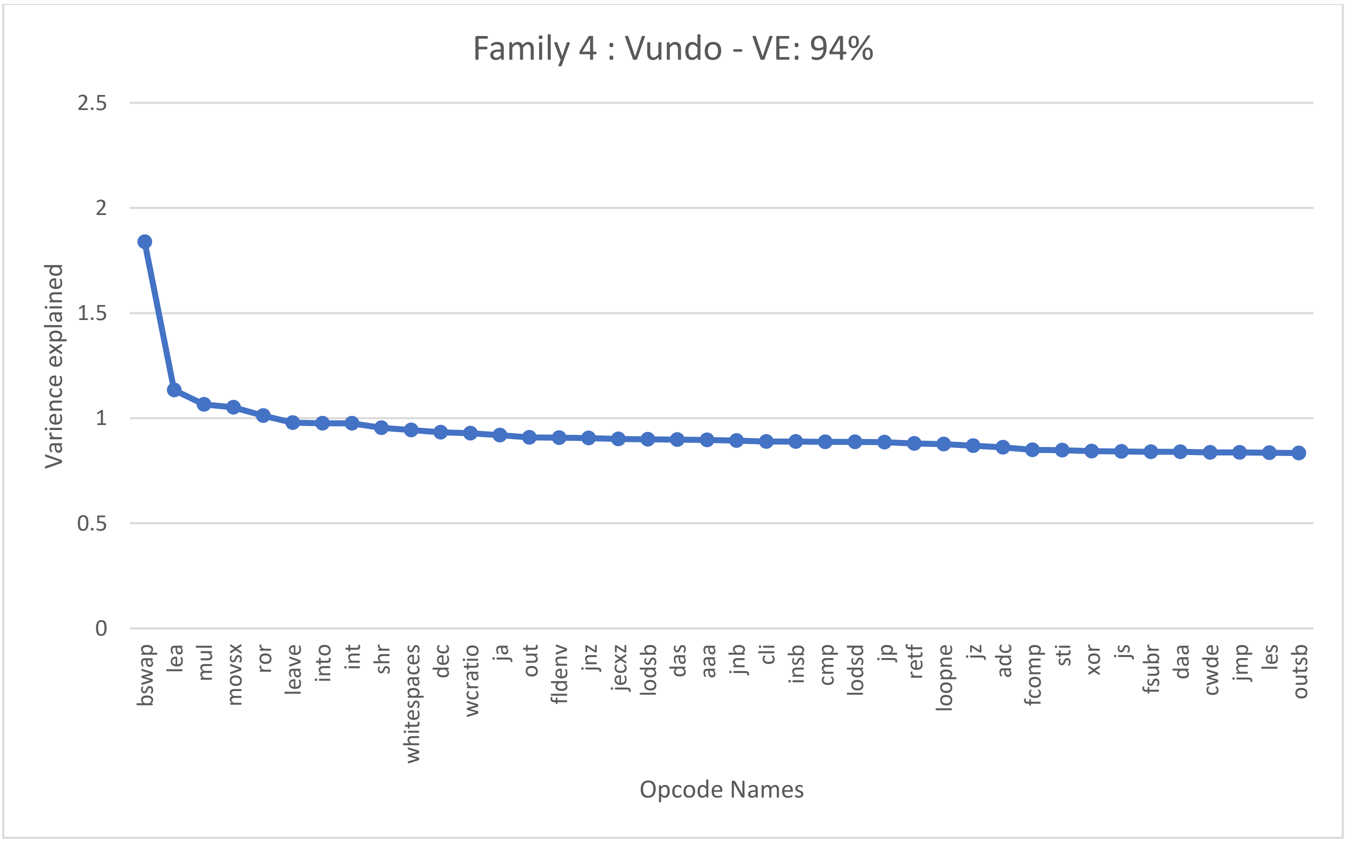
```
param_grid = {
   "n_estimators":[35,36,37,38,39,40],
   "learning_rate" :[0.1,0.2,0.3,0.4,0.5],
   "min_samples_split":[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20],
   "max_depth" : [2,3,4,5,6,7,8,9],
   "min_weight_fraction_leaf" : [0.1,0.2,0.3,0.4,0.5]
}
BEST PARAMS Params From CV Grid - {
                    'n_estimators': 38,
                       'learning_rate': 0.4,
                       'min_samples_split': 2,
                       'max_depth': 5,
                       'min_weight_fraction_leaf': 0.1,
                       }'''                                ** 98% value of the scores evaluation as
reliable, not prediction accuracy
```

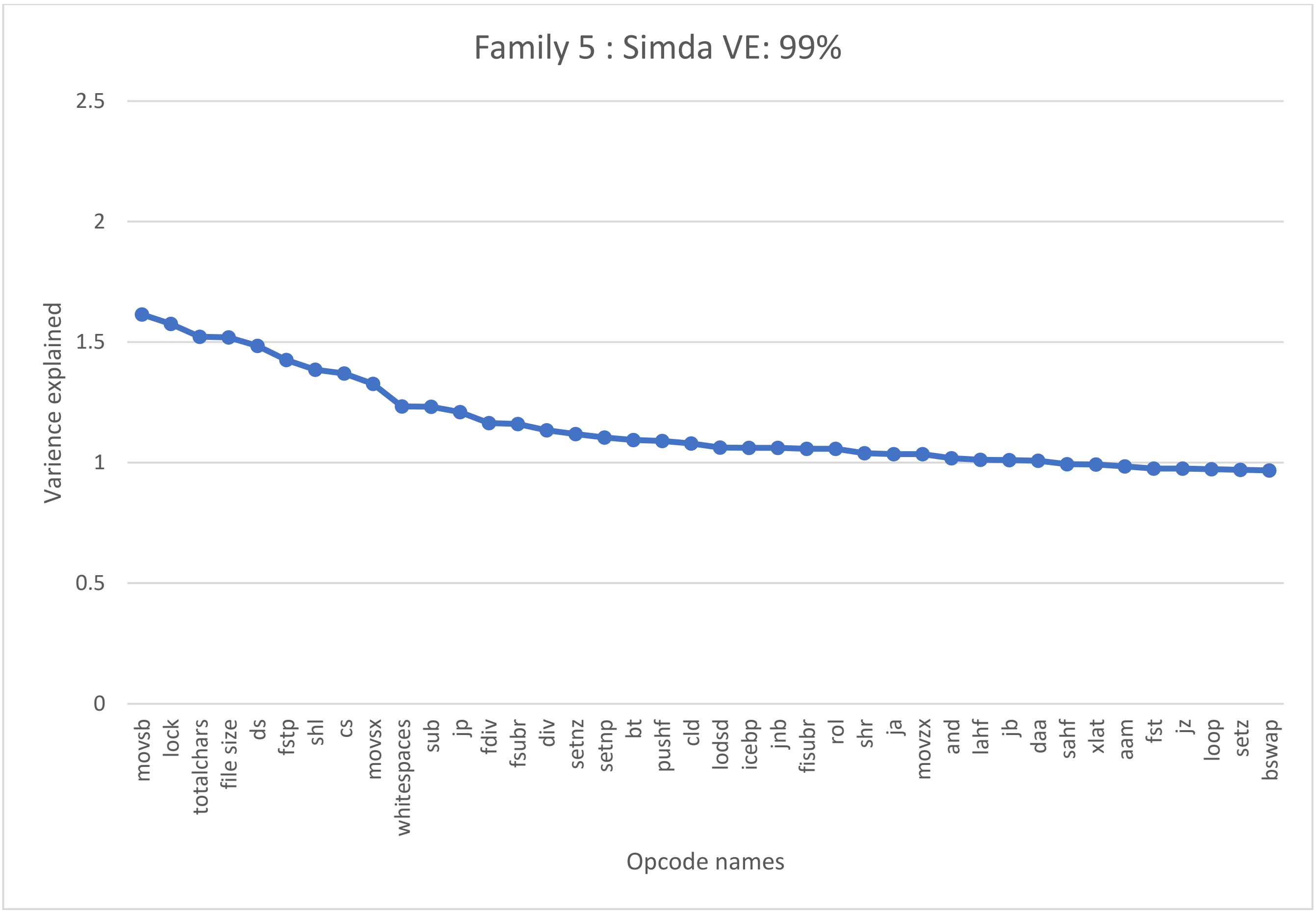
Scree Plots of the variance explained (VE) by the opcodes for each Family

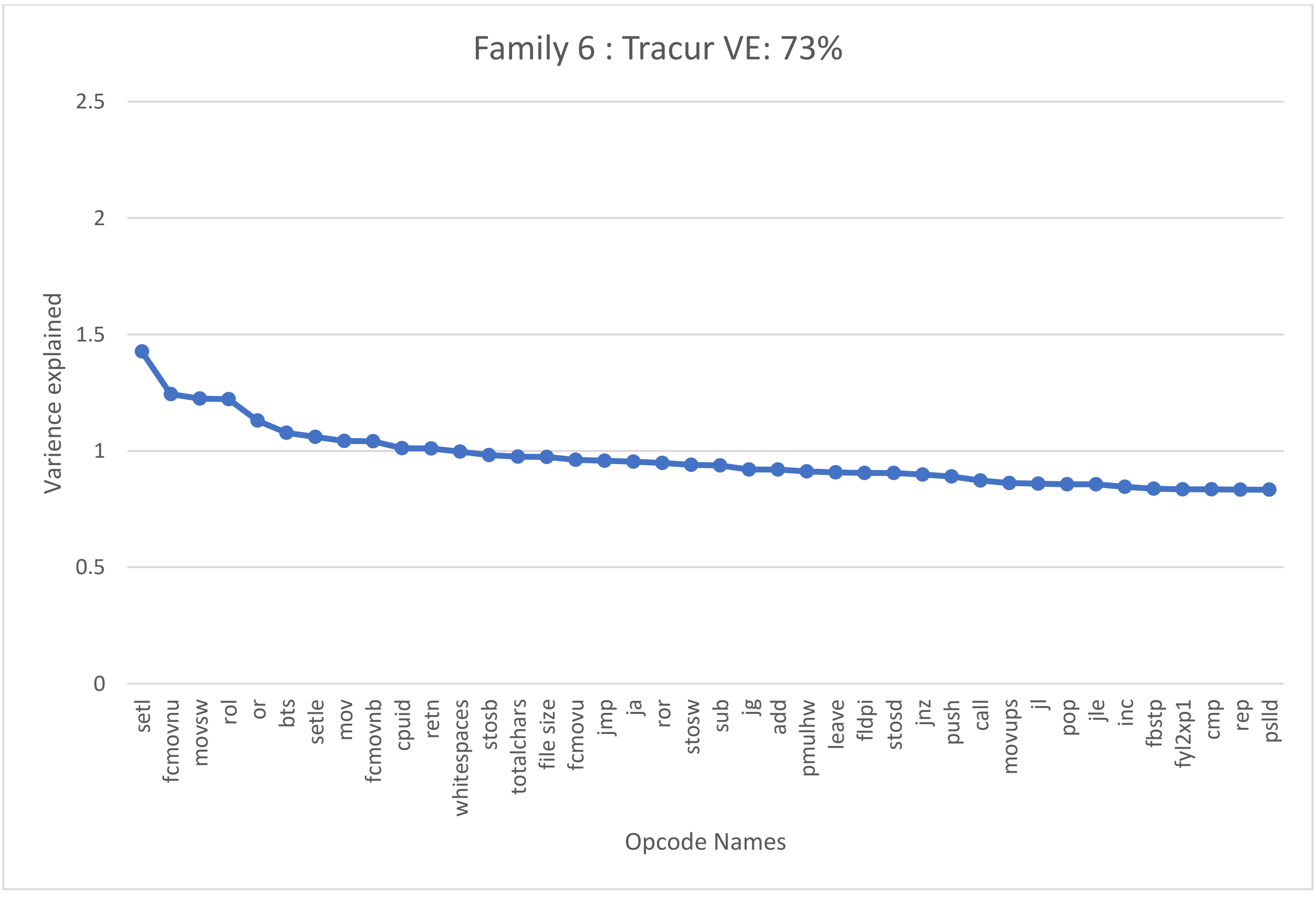


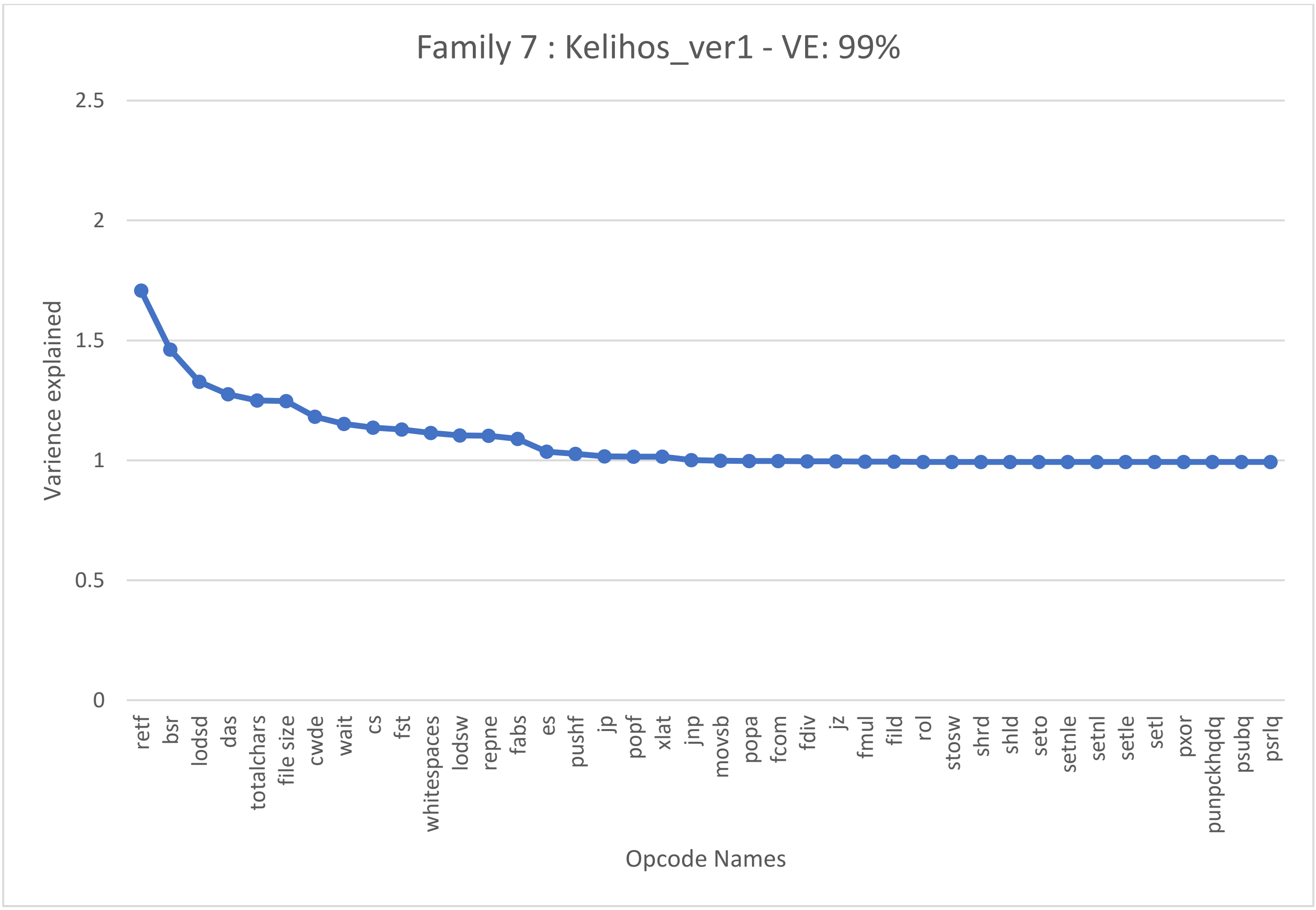



Family 3: Kelihos_ver3 - VE: 97%
Varience explained
2.5
2
1.5
1
0.5
0
popa
rep
xchg
jmp
es
xor
int
retn
sbb
adc
dec
lea
jg
clc
pop
neg
add
repne
rdtsc
whitespaces
lodsd
rcl
ja
totalchars
file size
nop
wait
call
cs
jbe
wcratio
cli
sal
push
jnb
mov
daa
les
jb
inc
Opcode Names

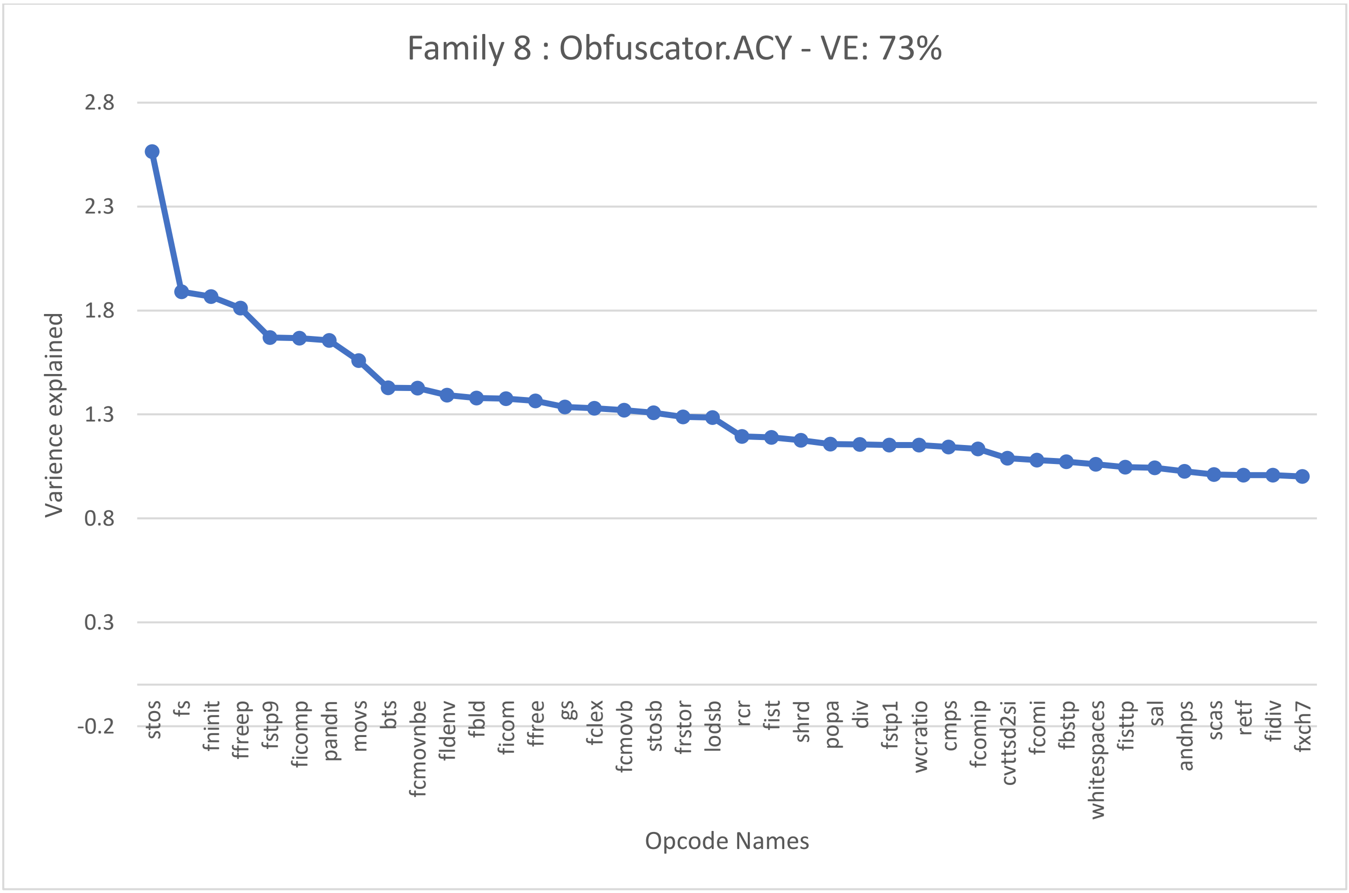

Family 4 : Vundo - VE: 94%
Varience explained
2.5
2
1.5
1
0.5
0
bswap
lea
mul
movsx
ror
leave
into
int
shr
whitespaces
dec
wcratio
ja
out
fldenv
jnz
jecxz
lodsb
das
aaa
jnb
cli
insb
cmp
lodsd
jp
retf
loopne
jz
adc
fcomp
sti
xor
js
fsubr
daa
cwde
jmp
les
outsb
Opcode Names


Family 5 : Simda VE: 99%
2.5
2
1.5
1
0.5
0
Varience explained
movsb
lock
totalchars
file size
ds
fstp
shl
cs
movsx
whitespaces
sub
jp
fdiv
fsubr
div
setnz
setnp
bt
pushf
cld
lodsd
icebp
jnb
fisubr
rol
shr
ja
movzx
and
lahf
jb
daa
sahf
xlat
aam
fst
jz
loop
setz
bswap
Opcode names

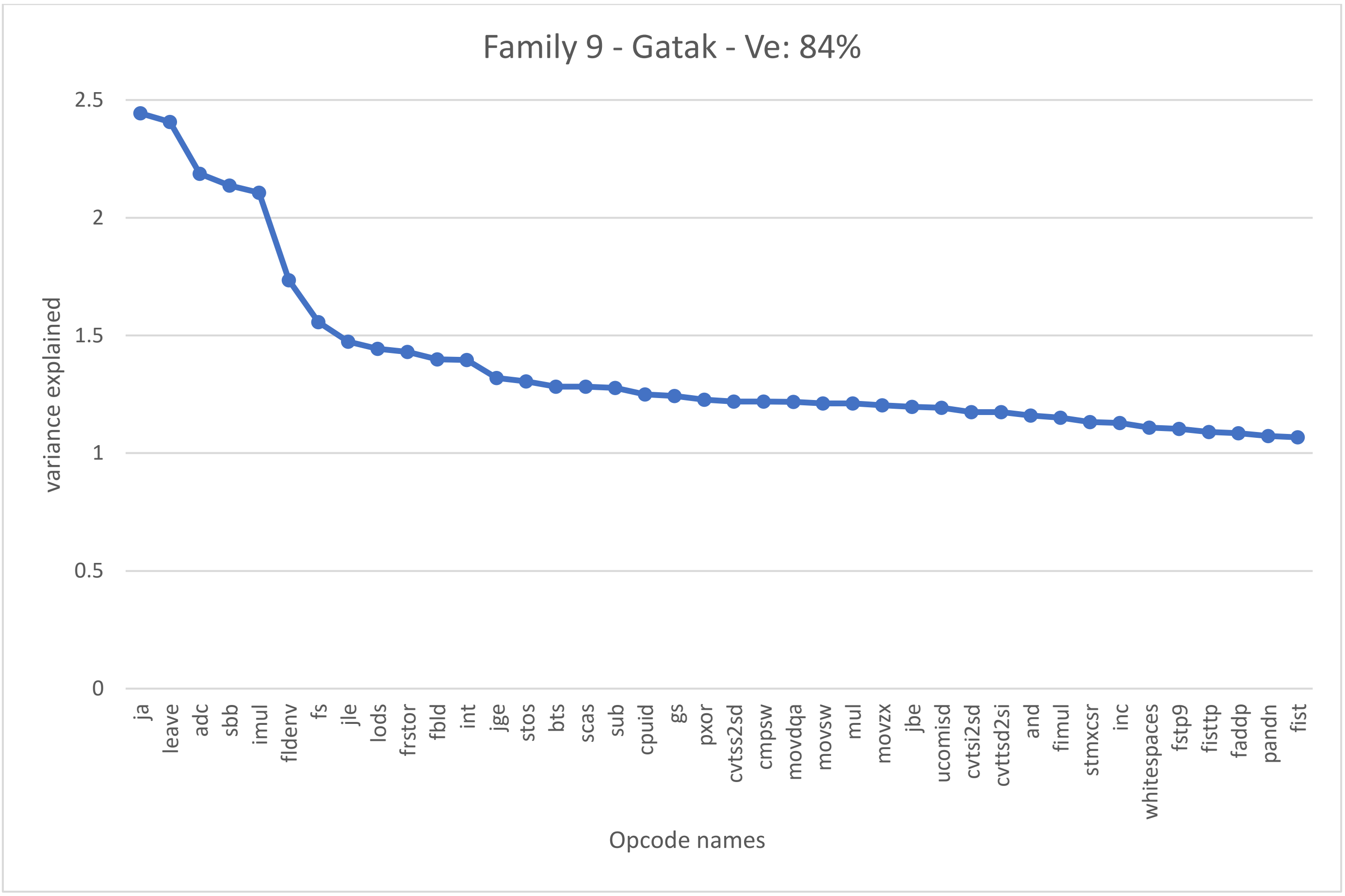

Family 6 : Tracur VE: 73%
2.5
2
1.5
1
0.5
0
Varience explained
setl
fcmovnu
movsw
rol
or
bts
setle
mov
fcmovnb
cpuid
retn
whitespaces
stosb
totalchars
file size
fcmovu
jmp
ja
ror
stosw
sub
jg
add
pmulhw
leave
fldpi
stosd
jnz
push
call
movups
jl
pop
jle
inc
fbstp
fyl2xp1
cmp
rep
pslld
Opcode Names



Family 7 : Kelihos_ver1 - VE: 99%
Varience explained
2.5
2
1.5
1
0.5
0
retf
bsr
lodsd
das
totalchars
file size
cwde
wait
cs
fst
whitespaces
lodsw
repne
fabs
es
pushf
jp
popf
xlat
jnp
movsb
popa
fcom
fdiv
jz
fmul
fild
rol
stosw
shrd
shld
seto
setnle
setnl
setle
setl
pxor
punpckhqdq
psubq
psrlq
Opcode Names



Family 8 : Obfuscator.ACY - VE: 73%
Varience explained
2.8
2.3
1.8
1.3
0.8
0.3
-0.2
stos
fs
fninit
ffreep
fstp9
ficomp
pandn
movs
bts
fcmovnbe
fldenv
fbld
ficom
ffree
gs
fclex
fcmovb
stosb
frstor
lodsb
rcr
fist
shrd
popa
div
fstp1
wcratio
cmps
fcomip
cvttsd2si
fcomi
fbstp
whitespaces
fisttp
sal
andnps
scas
retf
fidiv
fxch7
Opcode Names



<u>Email notification</u>

Using the *time* module, Some tasks took upto 8.7 hours. Waiting for completion before continuing was not a good use of time. It was therefore decided to build a function that could work as a tool to

a) write a small report if needed

b) for this to be sent as an email.

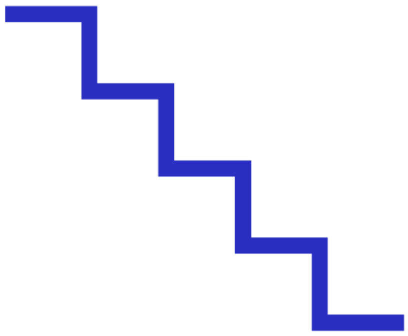
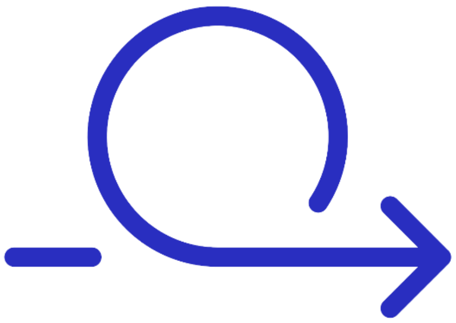
This would then give real time updates of the completion of a task to save on time and to allow for jobs to be left running outside of working hours or whilst other tasks were undertaken.

ReportWriter was mainly used for training models, so its string variables were called *ModelName* and *BestParameters.* These two string are essentially the title of the message and the content of the message being sent. In hindsight these variables could have had more general names. R*eportWriter* would create a text file in the working directory of the function and write to it the title and message passed to the function. The Title being a single instance string passed straight through and the Message being a string that would grow as test data was passed to it.

The *Notify* function required the import of the – smtplib module to open a *Simple Mail Transfer Protocol* client session so that emails could be sent with log in credentials. The smtplib.SMTP_SSL method was called via gmail Smpt server on port 465 to create an instance of a mail connection. The ESMTP method was called to give gmail a more verbose connection. The function then called the login method that takes access credentials. From this point emails can be sent using the .*sendmail* method, which takes the receiver, sender and the message. Although the SMPT_SSL method was encrypted using Secure Sockets Layer, This did expose the computer system to Abuse Case:4 , so it the port was closed on every calling of the function.

Development Methodology

Broadly speaking, software development project methodologies are derived from two ended spectrum with Waterfall at one end and Iterative at the other.

Waterfall	Iterative
	
Stages are completed in strict sequence upon completion of the previous stage	Stages are completed more frequently, to looser requirements to adapt to change and uncertainty.

The Waterfall approach is born from the legacy of large engineering programs that have taken place for many years. In the past for example, it would be infeasible to build the root before you had built the ground floor. Therefore, it was required to take a step by step approach of signing off from one stage to the next.

However, with the rapid change in computer technology, specifications and performances change mid-project. This would alter the requirements therefore many projects now lean towards taking smaller benchmarks and steps throughout a with a flexible approach to experimentation.

Cross Validation

Learning the best parameters of a model and then testing it on the same data would be a methodological mistake: a model that just repeats the labels (e.g. malware family) of the samples that it has just seen would have a perfect score but would fail to predict anything of value on fresh unseen data. This situation is called **overfitting**. To avoid overfitting as much as possible, we hold out part of the available data as a **test set** that has never been seen by the model.

When evaluating different hyperparameters for models, such as Cost (which must be manually set for an SVM's) , the risk of overfitting still exists on the test set because these parameters can be tweaked until the estimator performs optimally. Knowledge about the test set can "leak" into the model and evaluation metrics no longer report on the true generalization performance.

Cross Validation (CV) solves this problem by holding out a further part of the dataset, a so-called "validation set". In this situation the validation set size was a 20% slice of the training data, which would leave the other 80% for training the model. This 20/80 split is a healthy ratio to uphold the prevention of overfitting whilst thoroughly training a model.

The basic approach, called *k*-fold CV. The training set is split into *k* smaller chunks. The procedure is followed in a loop for each of the *k* "folds":

- The model is trained with "K – 1" of the folds as training data;
- the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

The final performance measure reported by *k*-fold CV will be the average of the values computed in the loop. This approach does add a computational expense but makes good use to not waste data which is a major advantage in problems such as inverse inference where the number of samples is very small.

XGBoost Tuning

Based on Jain[24] and the manual[23] the most important **hyperparameters** in XGboost were:

Parameter name	**Description**	**Values of range used**	**Best final training obtained**
learning_rate	the step size shrinkage used to prevent overfitting. Rangeing from 0,1		0.4
min_child_weight:	Minimum sum of instance weight (hessian) needed in a child. The larger, the more conservative the algorithm will be.	1 -6 step 1 [1,2,3,4,5,6]	2
max_depth:	determines the depth each tree is allowed to grow during any boosting round.	[3,4,5,6,7,8,9,10]	5
subsample:	percentage of samples used in each tree. A low value can cause underfitting.	subsample: between 0.5-0.9. [0.5,0.55,0.6,0.65,0.7, 0.75,0.8,0.85,0.9]	0.8
colsample_bytree:	percentage of features used in each tree. A High value can cause overfitting.	between 0.5-0.9. [0.5,0.55,0.6,0.65,0.7, 0.75,0.8,0.85,0.9]	0.8
n_estimators:	number of trees to build.	[35,36,37,38,39,40],	38
objective:	determines the loss function to be used like reg:linear for regression problems, reg:logistic for classification problems with only decision, binary:logistic for	binary:logistic	binary:logistic

	classification problems with probability		
a:	controls whether a given node will split based on the expected reduction in loss after the split.	[0.1,0.2,0.3,0.4,0.5]	0
alpha	L1 **Ridge based** regularization on leaf weights. A large value leads to more regularization.	[0, 0.001, 0.005, 0.01, 0.05]	0
lambda:	L2 **Lasso based** regularization on leaf weights and is smoother than L1 regularization.	[0, 0.001, 0.005, 0.01, 0.05, 1]	1

Tuning

Based on Jain[24] and the manual[23] the most important **hyperparameters** in XGboost were:

Parameter name	**Description**	**Values of range used**	**Best final training**
learning_rate	the step size shrinkage used to prevent overfitting. Rangeing from 0,1		0.4
min_child_weight:	Minimum sum of instance weight (hessian) needed in a child. The larger, the more conservative the algorithm will be.	1 -6 step 1 [1,2,3,4,5,6]	2

max_depth:	determines the depth each tree is allowed to grow during any boosting round.	[3,4,5,6,7,8,9,10]	5
subsample:	percentage of samples used in each tree. A low value can cause underfitting.	subsample: between 0.5-0.9. [0.5,0.55,0.6,0.65,0.7,0.75,0.8,0.85,0.9]	0.8
colsample_bytree:	percentage of features used in each tree. A High value can cause overfitting.	between 0.5-0.9. [0.5,0.55,0.6,0.65,0.7,0.75,0.8,0.85,0.9]	0.8
n_estimators:	number of trees to build.	[35,36,37,38,39,40],	38
objective:	determines the loss function to be used like reg:linear for regression problems, reg:logistic for classification problems with only decision, binary:logistic for classification problems with probability	binary:logistic	binary:logistic
gamma:	controls whether a given node will split based on the expected reduction in loss after the split.	[0.1,0.2,0.3,0.4,0.5]	0
alpha	L1 **Ridge based** regularization on leaf weights. A large value	[0, 0.001, 0.005, 0.01, 0.05]	0

	leads to more regularization.		
lambda:	L2 **Lasso based** regularization on leaf weights and is smoother than L1 regularization.	[0, 0.001, 0.005, 0.01, 0.05, 1]	1

GLOSSARY

Abuse Case – [21] Sometimes known as a *Misuse Case* are hypothetical situations used in requirements where the software you are building is incorrectly or maliciously used. The basic idea is to represent the actions that a system should prevent alongside those that it should support as so that security analysis of requirements is easier.

Apriori Algorithm

Based on the idea that a subset of a frequent itemset must also be frequent itemset itself.
The Algorithm works by using frequent item sets to generate association rules for the item sets, within frequent itemset sets by using a level-wise search with K-item sets to explore K+1 item sets. Support count = freq items by size of the transaction database. Infrequent item sets can be pruned.

The algorithm uses a *candidate generation process.* At every iteration a Candidate list and Frequent Item list is built using the support count. After that we use the Frequent list of K-itemsets in determining the candidate list of K=1 itemsets with pruning.
This is repeated until there is an empty Candidate or Frequent Lists of the K-item sets.
At the end a List of K-1-itemsets is returned.

Argument – a value that is passed between functions or subroutines in a program

Cosine Similarity is a measure of similarity , generally used on terms within documents.

$$cos(V_1, V_2) = (V_1 \cdot V_2) / ||V_1|| \text{ x } ||V_2||$$

The dot product of each vector divided by the product of each vectors magnitude.

Cost – In Support Vector Machines, the Cost parameter allows specification of the cost of a violation to the margin. It helps to determine the extent to which the model underfits or overfits the data.

Dynamic malware analysis - executing code on a virtual environment with a subsequent behaviour report based on an execution trace.

Expectation Maximization algorithm – is a probabilistically based way of soft clustering where each cluster corresponds to a generative model (Gaussian or Multinomial). Each cluster corresponds to a probability distribution to discover the parameters. (Eg mean/covariance in a Gaussian model). EM algorithm allows you to infer those parameter values. Like the K-means , the EM will place the data points randomly in space, however EM will determine the probability that it came from a certain class rather than assign it to a cluster. Once it has computed the assignment that it came from a certain class. It will use those estimates as the new parameters, Mean /Variance to fit the point assignment more accurately and reiterate until convergence.

Gini index – is a number that measures the distance between the line of perfect equality and the Lorenz curve.

$$Gini\ Coefficiant = \frac{A}{A+B}$$

A = the space between the lorenz curve and the line of perfect equality
B = the line under the Lorenz curve
The number produced will be between 0-1. 0 perfect equality and 1 inequality

Hidden Markov Models – A model of the probability that a given observation is malware based on the sequence of observations (malware features) where some observations are still hidden (unknown features). These are formally known as the Hidden States. The probability of being malware is the Transition Probability and the probability of a feature is the Emission Probability.

Hyperparameters – are the parameters that are set before a machine learning mode l can begin to learn.

Hex dump - a hex dump is a hexadecimal view (on screen or paper) of computer data, from RAM or from a file or storage device. Looking at a hex dump of data is commonly done as a part of debugging, or of reverse engineering.

Illegal opcode - an undocumented opcode instruction that is not mentioned in any official documentation released by the CPU's designer or manufacturer, which nevertheless has an effect.

Inverse inference - the process of calculating from a set of observations the causal factors that produced them

Inverse inference - the process of calculating from a set of observations the causal factors that produced them.

Jaccard similarity Measure – of two sets is the size of the intersection divided by the size of their union.

$$\mathrm{Sim}(C_1, C_2) = \frac{|\, C_1 \cap C_2|}{|\, C_1 \cup C_2|}$$

Local Binary Patterns (LBP) are a method of texture classification for images, in this case Malware. The intensity of a central pixel in a 3 x 3 grid (window) is taken as the reference point. Depending on if each local pixel in the grid is brighter or lower than the central reference. This will determine a new score for the neighbours of either a 1 or a zero. This string of binary digits is interpreted as the LBP value for that window. LBP is illumination invariant meaning , pixel intensity spread overall, will not change the relationships between the values. They will all increase/decrease by the same degree.

3 x 3 pixels intensity grid				0/1=low/hi grid				Binary number		
1	3	9		0	0	1				
6	4	2	→	1	4	0	→	00101011	→	43
7	3	5		1	0	1				

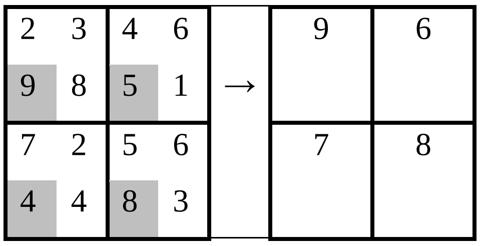
LogLoss - is a soft measurement of accuracy that incorporates probability. It is the Cross entropy between the distribution of the true labels and the predicted probabilities. From equation 2, it is the negative log likelihood of the model.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(p_{ij}) \qquad (2)$$

where N is the number of observations, M is the number of class labels, log is the natural logarithm, y_{ij} is 1 if observation i is in class j and 0 otherwise, and p_{ij} is the predicted probability that observation i is in class j.

Lorenz curve – Visual indicator on a 2d plot that shows the relation of population versus another derived variable (In economics, income.)

Maxpooling is a data reduction technique for images and was used to simplify the processing of the malware samples. Moving through the matrix with a non-overlapping window you take the max value in the filter pool for that step and transposes it to a reduced size grid. Referring to the diagram below, A *window size* and a *stride size* values need to be decided is selected to traverse a pixelated image. As a toy example, we shall use a window size of 2 on the grid bellow and a stride size also of 2. The maximum of the top left pool is 9. Striding to the next window, we find the next Max Pool value and so on, simplifying our image down. Alternatives are Average pooling.



Minhashing - The probability (over all permutations of the rows that
The Minhash function of a column C is h(C) = the number of the first (in the permutated order) row in which column C has a 1 (or the value being searched for). Each Minhashing hash function is associated with a permutation of the rows of the Matrix in question, not a fully permutated version of the matrix. This is where a saveing occurs.

Mutual information – A measure from Information theory represented as the symbol I and is given as the entropy of Y minus the entropy of X given Y.

$$I(x,y) = H(y) - H(x,y)$$

It is a measure that indicates how statistically dependant two variables are.

Overfitting – is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably" - OxfordDictionaries.com
Operation codes (OPC) are the mnemonic representation of machine code, which symbolize assembly instruction that specifies the operation to be performed eg arithmetics, data movment, logical , or program control.
Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” or “flattening”; however, to avoid confusion, the terms “pickling” and “unpickling” used. Any

object in Python can be pickled so that it can be saved on disk. The poicke module "serialises" the object before writing it to file. It is a way to convert a Python object (list, dict, etc.) into a character stream that contains all the information necessary to reconstruct the object in another Python programme.

Packing – By use of a Packer tool to compresses, encrypts, and/or modifies a malicious file's format. Packings aim is t o decrease the chance of detection by antimalware products and help to avoid analysis.

Portable executable (PE) file – is a file format for executables used in 32 and 64 bit windows Operating systems.

Regularisation refers to an optimisation control that trades quality of fit in the training data against model simplicity/coefficient size and therefore improves on generalisation error. Specifically for gradient boosting, this controls tree depth/.complexity and the number of trees I the model.

Shrinkage is a technique to improve an estimator by averaging the estimate with a simpler estimator that is less informed on the training data. In GBM shrinkage is controlling a learning rate that controls how much the latest term of the model is averaged into earlier terms of the model.

Static malware analysis - where the code is disassembled without being trun and the control flow is explored for malicious patterns

Target variable –The "**target variable**" is the **variable** whose value is to be modelled/predicted by other **variables, in this case the family of Malware.**

TFIDF or term frequency–inverse document frequency - a numerical statistic intended to reflect how important a word is to a document in a corpus. It is often used as a weighting factor in searches of information retrieval, text mining, and user modelling. The tf–idf value increases proportionally to the number of appearances of a word in a document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

Use Case – a Hypothetical case in which software may be used, Often presented on a card. Elements include: Basic Info, Preconditions, Normal course, Alternative Course, Post conditions, Exceptions, Summary Iinputs/OutPuts, Business rules.

Variance - the expectation of the squared deviation of a variable from its mean.