Institute of System Studies and Analysis, DRDO

Report on

Malware Detection using System API Calls

Submitted by:	Nitin Mohan B. Tech Computer Science (VII Semester) Northern India Engineering College GGSIPU
Duration of Training:	8 weeks
Submitted on:	XX July, 2017
Submitted to:	Mr. Yogesh Chandra Scientist – 'F'

Certificate

This is to certify that the project on ‘Malware Detection using System API Calls’ has successfully been completed by Nitin Mohan. The following report has been prepared for the same, and found worthy of being submitted as Project Report for ‘Summer Training for B. Tech 3rd Year’ which he did under my tutelage during the months of June and July, 2017.

Acknowledgments

I would like to express sincere gratitude to Mr. Shashi Bhushan Taneja, Director ISSA for allowing me with the opportunity to carry out my summer internship at ‘Institute of System Studies and Analysis’.

I would also like to thank Mr. Yogesh Chandra (Scientist – ‘F’) for his guidance and encouragement during the course of the project work and for his mentorship which was of tremendous help towards the successful completion of the project.

ABSTRACT

In the era of ever-growing number of smart devices and ubiquitous sensors, *malware executables* have risen to become an increasingly severe threat to modern computers and internet security. These executables are designed so as to damage computer systems and spread over networks without the knowledge of the owners using the systems. And so, the detection of malwares has turned into an endless battle between ever-evolving malware and antivirus programs. Two approaches have been derived for it i.e. Signature Based Detection and Heuristic Based Detection. These approaches performed well against known malicious programs but cannot catch the new unknown malicious programs. Different researchers have proposed methods using data mining and machine learning for detecting new
Malicious programs. Among them, Dynamic analysis is known to be effective in terms of providing behavioural information.
As malware authors increasingly use obfuscation techniques, it becomes more important to monitor how malware behaves for its detection. In this paper, the detection is approached using dynamic analysis of malware. We adopt DNA sequence alignment algorithms and extract common API call sequence patterns of malicious function from. We find that certain malicious functions are commonly included in malware even in different categories. From checking the existence of certain functions or API call sequence patterns matches, we can even detect new unknown malware. The result of our experiment shows high enough F-measure and accuracy.

INDEX

INTRODUCTION

The proliferation of modern computers, internet users, and communication infrastructure has led to them becoming critical sources of data by generating Big Data every step of the way in this IoT era. This has also led to multiplicative increase in malwares and cyber-attacks targeted at them. Malware variants have evolved to gain unauthorized access of systems, to get the economic benefits by illegal ways. Propagation of malware is havoc to internet security, commercial companies, privacy of users and governments alike.

Malware is derived from malicious software. It is an instance of malicious code with intention to subvert the function of system and has potential to harm a computer or network. It covers a range of threats like Virus, Trojans, Adwares, spywares, etc. They replicate themselves and enter into the system in different ways; either through multimedia or through the most popular way of getting downloaded into the system disguised as the genuine applications. Different malware detection systems have been introduced till date to circumvent the attacks caused by malwares. A malware detection system identifies malwares and defends the system to perform its function.

Currently existing methods for tackling malware are primarily based on two complementary approaches. These are classified according to the type of features they use for discovering malware activity.

The **Signature-based** detection approach relies on the identification of unique string patterns in the binary code. This technique uses *static* technique for creating signatures. It cannot cope with malwares unseen previously, also known as *'zero day attacks'*. When a novel type of malware family is observed, we need to analyse an instance of malware, generate a signature for it and insert it into malware database for reference inducing a classifier. During this period, an instance of this malware might attack several systems or networks. Hence, Signature-based detection approach has become inefficient and intractable.

Knowing the weakness of detection systems, malware designers developed code *obfuscation* techniques like *code reordering*, *garbage insertion*, *variable renaming* to disguise their content. Following this intuition, **Heuristic-based** approach has been introduced an automated classification system. It is based on rules determined by experts, which relies on *dynamic* analysis of malicious behaviour that deviates significantly from normal behaviour. It precisely deals with *unknown* malware discovery. However, this detection approach generates greater amounts of *false alarms* than Signature-based detection since not each of the suspicious executable file is indeed malicious. It has been observed that each of the two approaches had some limitations.

Further antivirus vendors attempted to use *Individual as well as Hybrid* analysis approach for mining features and tackling newly emerging malwares. They achieved a precise detection rate and low false positives compared to existing malware detection methods. In investigated malware detection systems based on integrated static and dynamic analysis features using data mining approaches. An appropriate determination of malware variant depends on the feature type employed for discovering malicious activity. The performance of the system depends on the feature type which is the best indicator of malware and requires least time for quantifying the correlation between malicious activities.

In this work, in order to achieve much more accurate detection, dynamic analysis technique is used. This technique analyses the extracted API calls from the dataset made by compiling a list of executed executables, and traces their behaviour. Two major approaches in dynamic analysis are *control flow analysis* and *API call analysis*. Both approaches detect malware based on analysis of similarity between the behaviour of the new and known samples mentioned in the dataset used. Many of currently available API call analysis techniques fail to detect malware due to some circumvention.

Some techniques focus on extracting APIs that are *frequently observed* in malware executable samples. They monitor APIs that are called and calculate the frequency and total number of events that certain API functions called. Even though they quickly reveal the characteristics of malware, they fail to show the sequence of malware behaviour and can be easily evaded by malware authors' inserting and executing dummy and redundant API calls.
Others extract API call sequence for each type of malware and develop static signature based on it. They are better from the semantic view because they monitor the sequence of calls and the flow of programs. However, simply creating signatures by extracting frequently found call sequences for malware types does not allow them to detect malware in polymorphic or unknown form. It can also be evaded by malware authors' evading tricks such as inserting redundant API calls.

In this work we propose a dynamic malware detection system using data mining techniques that examines the extracted features, i.e., System API calls of malware and clean programs and uses them to classify unknown programs into 'malicious' or 'benign' executables. We adopt Sequence alignment and various other filters in order to extract the similar sub-sequences from the various sequences. Our goal in the evaluation of this system is to simulate the task of detecting new malicious executables.

Objectives

The primary objective of the aforementioned project is to learn about malware types and understand their respective behaviour in order to finally be able to defend systems and the sensitive data from future intrusive attacks at the hands of new unknown malwares.

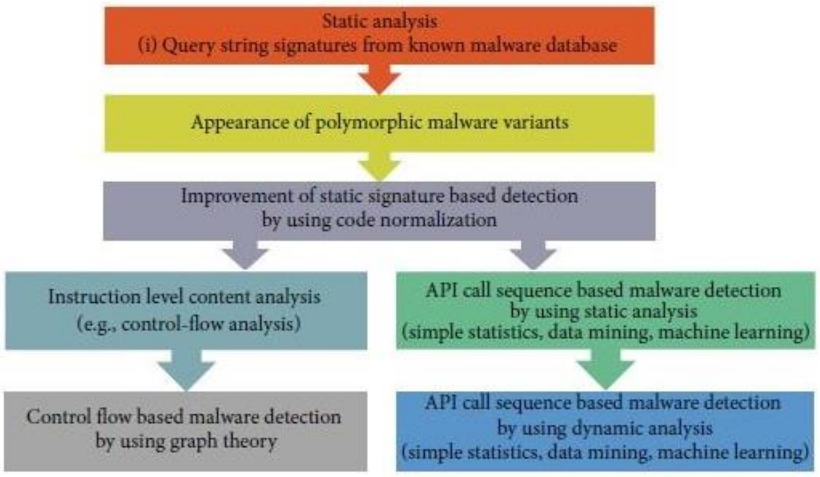

Static analysis
(i) Query string signatures from known malware database
Appearance of polymorphic malware variants
Improvement of static signature based detection by using code normalization
Instruction level content analysis (e.g., control-flow analysis)
API call sequence based malware detection by using static analysis (simple statistics, data mining, machine learning)
Control flow based malware detection by using graph theory
API call sequence based malware detection by using dynamic analysis (simple statistics, data mining, machine learning)

FIGURE 1: Advances made in the field of Malware analysis

We also learn about the workings and many capabilities of the much-acclaimed WEKA (Waikato Environment for Knowledge Analysis) suite for machine learning which is of immense help when it comes to the implementation of the various classification methods on the dataset used.

Also, the various filters included in the WEKA suite are discussed further. These filters are used so as to help the system make proper sense of the dataset and the numerous elements and their respective attributes which contribute to their classification as either *malign* or *benign*.

In addition to this, a comparative side-by-side look at the classification methods applied; their mechanism; their results, along with the responsible facets for such results being observed with regard to the data provided in the dataset are discussed further on.

Finally, this report aims to shed light on the need for making further and continuous improvements in the field of personal computing systems and network security from the malevolent intents of individuals and organisations on the look-out for vulnerabilities in system defences for a chance to exploit and abuse user data for personal gains.

SCOPE

Malware analysis has made numerous advances in terms of the capability of the methods involved in making sense of different types of data and deciphering the nature of the entity with the help of that knowledge.

Signature-based detection was proposed in its early stage. In this stage, automatic generation of malware's signatures as much as possible was assumed to be important and this increased pattern matching speed. However, the signature based detection method shows following weaknesses:

> It requires continuous updates of signature and high maintenance cost. In addition, such method could be easily evaded by malware in polymorphic form. To overcome these weaknesses, it embraces code normalization to capture the canonized original maliciousness. Through this, capturing malicious codes that vary by polymorphic techniques applied becomes possible. However, it is still weak in detecting obfuscated malware. Besides, some execution paths could be only explored after execution.

Malware analysis technique kept advancing due to these unrealised needs; hence, Dynamic analysis was proposed. Dynamic analysis methods are known to perform well for obfuscated malware. Dynamic analysis executes malware, monitors how it behaves, and detects unknown malware that shows similar behaviour in comparison to the known ones. Two major Dynamic analysis methods that are well known are *control flow analysis* and *API call analysis*.
API call information shows how malware behaves. API call information can be extracted by both static and dynamic approaches. With static approach, API list can be extracted from PE format of the executable files. With dynamic approach, the APIs that are called can be observed by running the executable files (usually on Virtual Machines).

API call sequence information collected through the dynamic approach can be used for creating behavioural patterns. The information gathered through the dynamic approach can also be processed using simple statistics such as frequency counting and data mining or machine learning.

In this study, we adopted dynamic method to extract the API call sequences. To get patterns that are rigor, we applied DNA sequence alignment algorithms (MSA and LCS). By using both extracted API call sequence patterns and critical API call sequences, we can detect the unknown malware or variants with high accuracy.

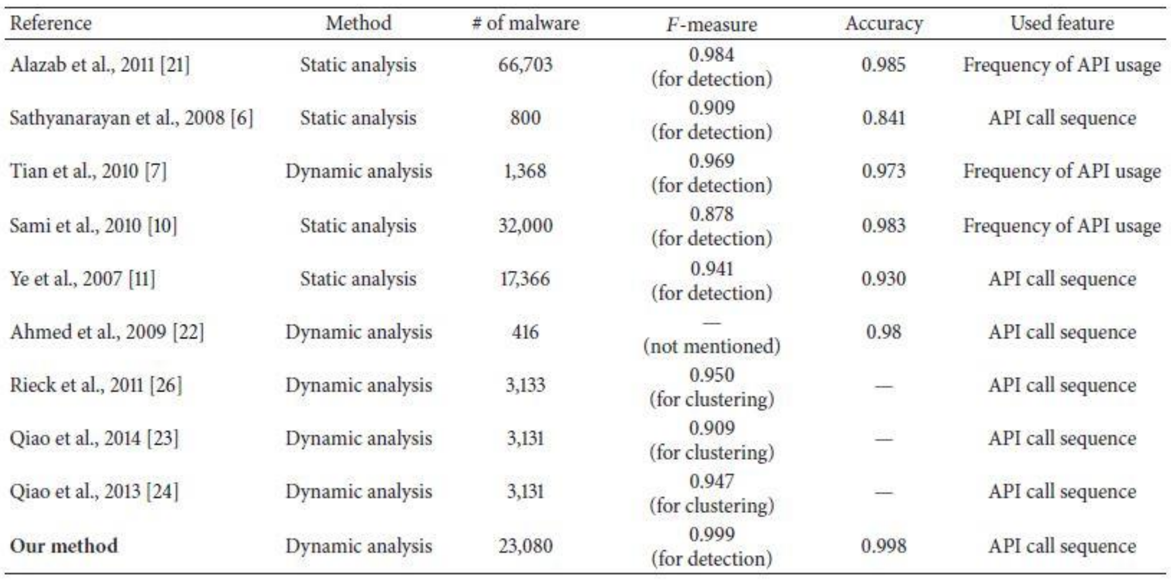
Sequence alignment algorithms are most widely used in the area of biometrics to calculate the similarity between two or more DNA sequences or to find certain DNA sub sequences from full DNA samples. Sequence alignment algorithms fall into two categories, which are *global* and *local* alignment.

> Global alignment aligns entire sequences. It is useful when we attempt to find the sequence of most similar length and strings. The well-known global alignment algorithm is the Needleman-Wunsch algorithm.
>
> Local alignment finds the highly similar sub sequences between given sequences. The Smith-Waterman algorithm is the well-known local alignment algorithm.
>
> > *Pairwise* sequence alignment methods find global alignments of the locally similar sub sequences of two sequences.

Multiple sequence alignment algorithm extends pairwise alignment to handle multiple sequences at the same time.

Since we need to handle multiple API call sequences, we applied multiple sequence alignment algorithm in this study.

The following is a summarised account of the previous empirical studies on API call analysis and their overall performances. We also compare them with our approach.



Reference	Method	# of malware	F-measure	Accuracy	Used feature
Alazab et al., 2011 [21]	Static analysis	66,703	0.984 (for detection)	0.985	Frequency of API usage
Sathyanarayan et al., 2008 [6]	Static analysis	800	0.909 (for detection)	0.841	API call sequence
Tian et al., 2010 [7]	Dynamic analysis	1,368	0.969 (for detection)	0.973	Frequency of API usage
Sami et al., 2010 [10]	Static analysis	32,000	0.878 (for detection)	0.983	Frequency of API usage
Ye et al., 2007 [11]	Static analysis	17,366	0.941 (for detection)	0.930	API call sequence
Ahmed et al., 2009 [22]	Dynamic analysis	416	— (not mentioned)	0.98	API call sequence
Rieck et al., 2011 [26]	Dynamic analysis	3,133	0.950 (for clustering)	—	API call sequence
Qiao et al., 2014 [23]	Dynamic analysis	3,131	0.909 (for clustering)	—	API call sequence
Qiao et al., 2013 [24]	Dynamic analysis	3,131	0.947 (for clustering)	—	API call sequence
Our method	Dynamic analysis	23,080	0.999 (for detection)	0.998	API call sequence

Malware analysis is a technique to study malware behaviour and its structure by extracting features which describes its malevolent intention.

Feature Extraction is the process of transforming the large, vague collection of inputs into a set of features and is seen as the first crucial stage of malware detection mechanism, and involves determine the representation of malicious software files from the various representations present. It is performed when there is abundant input data to an algorithm that tends to be more redundant and irrelevant. It is required to gain the precise measurement of features which influence the classification of input as benign or malicious. It also reduces the dataset for processing, resulting in low computational overhead.

The outcome of the feature extraction phase is a vector containing the frequencies of features extracted. Features extracted are chosen such that it attains maximum classification accuracy.

Feature extraction methods affect the performance of the system in terms *efficiency*, *robustness*, and *accuracy*.

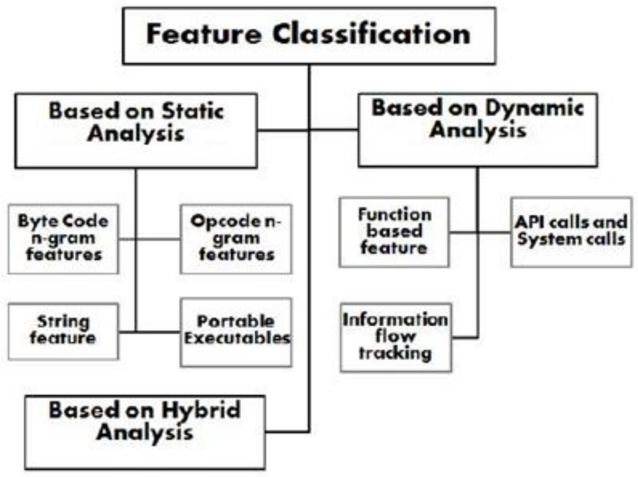


FIGURE 2: Feature extraction methods

(1) *Byte n-gram Features*

Byte n-gram features are sequences of n bytes extracted from malwares used as signature for recognizing malware. Although this type of features do not provide meaningful information, they yield high accuracy in detecting new malware.
Researchers extracted byte n-gram features from the binary code of the file where the L most occurring n-grams of each class in the training set were selected to denote the profile of the class. Every new instance was associated with a class closest profile using K-Nearest Neighbours (KNN) algorithm. Their experiments achieved 98% accuracy on dataset of benign and malware files. Byte n-grams in combination with opcode n-grams

when used as features, provided an extensive evaluation using a test collection. This method represented compact file print for each file type and used mahalanobis distance to determine the closest file type model based on centroids obtained.

(2) *Opcode n-gram Features*

Previous studies represented that opcodes feature extraction was more efficient and successful for classification. They reveal statistical diversities between malicious and legitimate softwares. Some rare opcodes are better predictors of malicious behaviour. First, all dataset executable files are disassembled and opcodes are extracted. An opcode is the assembly language instruction which describes the operation to be performed. It is short form of *operational code*. An instruction contains an opcode and operands, optionally upon which the operation should act. Some operations have operands upon opcodes may operate, depending on CPU architecture, registers, values stored on memory and stacks, etc. The action of an opcode takes in arithmetic, logical operations, and data manipulation operation. Opcodes are capable to statistically derive the variability between malicious and legitimate software. Researchers presented mean accuracy of the combinations of n-gram opcode sequences. They stated that 2-gram opcode sequence was the best N-gram sequence comparatively, which showed classification accuracy. However, for more than bi-gram opcode sequence the accuracy is decreased.

(3) *Portable Executables*

These features are extracted from certain parts of EXE files. Portable Executables (PE) features are extracted by static analysis using structural information of PE. These meaningful features indicate that the file was manipulated or infected to perform malicious activity. Researchers propounded a real time approach for malware detection based on structural features mined from PE. They tested performance on two datasets Malfease and VXheavens dataset which remarked that PE features has low processing overheads. These features may include part of the pieces of information given as follows:

1. *File pointer*: pointer denotes the position within the file as it is stored on disk, CPU type;

2. *Import Section*: functions from which DLLs were used and Object files, list of DLLs of the executable can be imported.

3. *Exports Section*: describes which functions was exported.

4. *Data extracted* from the PE Header that describes physical and logical structure of a PE binary which may include features like code size, debug size as well as creation time, file size, etc.

5. *Resource Directory*: indexed by a multiple-level binary-sorted tree structure, resources like dialogs and cursors used by a given file.

(4) *String Features*

These features are based on plain text which is encoded in executables like *windows*, *getversion*, *getstartupinfo*, *getmodulefilename*, *messagebox*, *library*, etc. These strings are consecutive printable characters encoded in PE as well as non-PE executables. Strings features are not very robust as they can be modified easily any time. A proposed a malware detection system, SBMDS, which classifies malware using SVM based on interpretable string features, outperformed existing antivirus softwares, and achieved better accuracy and efficiency using string features.

(5) *Function Based Features*

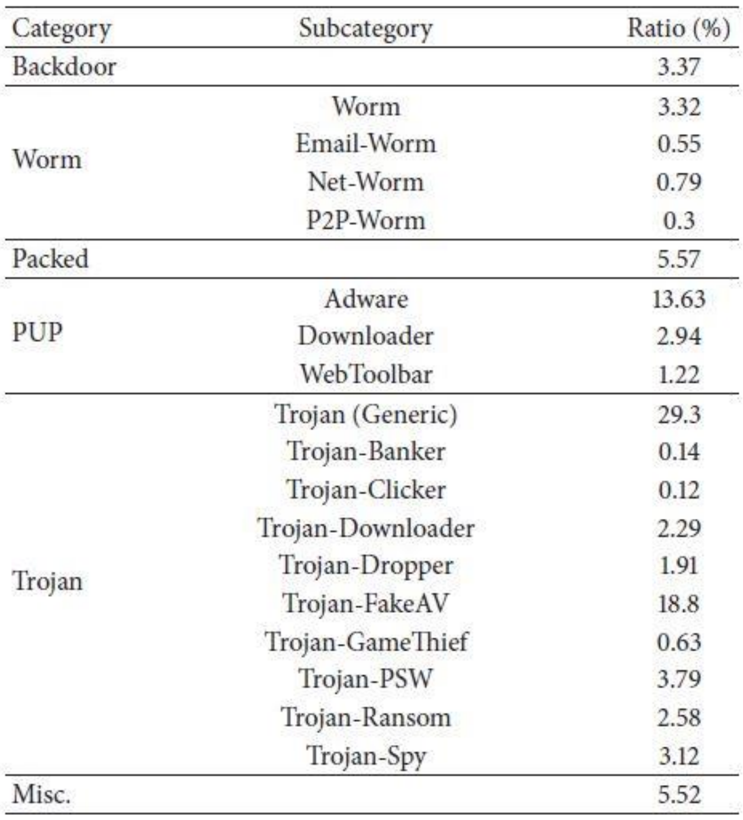
Function based features are extracted over the runtime behaviour of the program file. Function based features involves tracing functions that reside in a file for execution and utilize them to produce various attributes representing the file.
Dynamically analysed function calls including *system calls*, *windows application programming interface (**API**) calls*, their *parameter passing*, *information flow tracking*, *instruction sets*, etc. These functions increase the code reusability and maintenance. It is <u>semantically richer representation</u>.

Any malicious software for execution or replications invokes some kernel level system call to communicate with operating system; it is a sign of malicious activity.

(6) *Hybrid Analysis Features*

These features are obtained by combining both techniques -Static analysis as well as Dynamic analysis. It reduces the effect of counter measures of each static and dynamic technique for analysing malwares and improves the performance and detection rates. Researchers extracted static features of functions such as function length frequency and printable string Information (FLF and PSI) based on the functions of different lengths and the number of distinct printable strings present in unpacked malware executables. Further, they extracted Application Programming Interface (API) function calls and parameters by dynamic analysis. They provided superior results in terms of accuracy on combining the function based features and string features.
Similarly a combination of string and function features is used for classification of malwares which involved the use of different function length frequency ranges and printable string information that was found to perform better over seen malware set.
Subsequently, researchers introduced a hybrid approach eliminating the need for each individual static and dynamic malware analysis using both *emulation* (Qemu) and *simulation* (Wine) techniques for attaining the transparency without interference to the system. They extracted opcode sequences statically and Windows API calls dynamically; characterizing their behaviours in groups of system information, persistence, file creation, process or thread creation, adding registry keys, errors, etc. This method employed <u>classification algorithms</u> such as *KNN*, *SVM*, *Decision trees*, *Bayesian networks*, etc. to discriminate malwares and benign softwares. This provided more accurate results leading to notable increase in performance metrics.

For this study, a dataset having 23,387 randomly chosen executables (23,080 malware samples and 307 benign samples) from the malware dataset of the Malicia-project and Virustotal, has been selected.

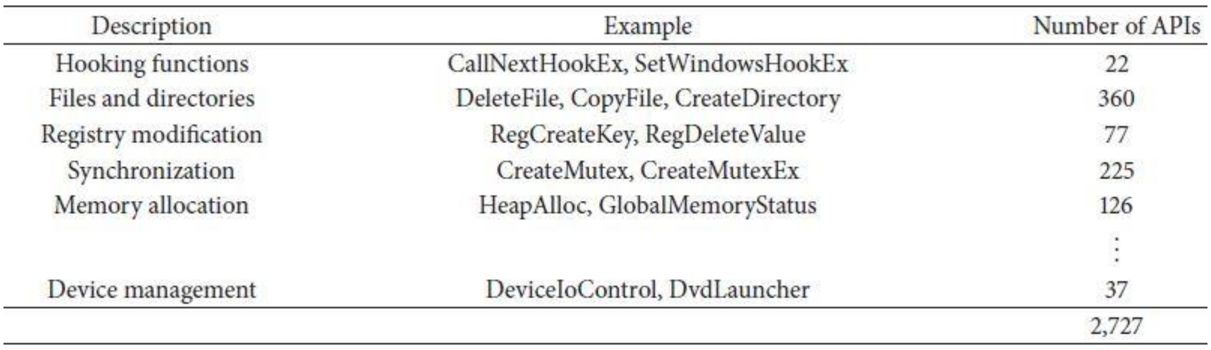


Category	Subcategory	Ratio (%)
Backdoor		3.37
Worm	Worm	3.32
	Email-Worm	0.55
	Net-Worm	0.79
	P2P-Worm	0.3
Packed		5.57
PUP	Adware	13.63
	Downloader	2.94
	WebToolbar	1.22
Trojan	Trojan (Generic)	29.3
	Trojan-Banker	0.14
	Trojan-Clicker	0.12
	Trojan-Downloader	2.29
	Trojan-Dropper	1.91
	Trojan-FakeAV	18.8
	Trojan-GameThief	0.63
	Trojan-PSW	3.79
	Trojan-Ransom	2.58
	Trojan-Spy	3.12
Misc.		5.52

FIGURE 3: Description of selected dataset

Categorization of APIs:

In our dataset, 2,727 kinds of API were found; we categorized them into 26 groups according to MSDN library. For example, *CallNextHook* API and *SetWindowsHook* API are categorized as ***hooking functions***. Likewise, *DeviceIoControl* API and *DvdLauncher* API are categorized as ***device control***.



Description	Example	Number of APIs
Hooking functions	CallNextHookEx, SetWindowsHookEx	22
Files and directories	DeleteFile, CopyFile, CreateDirectory	360
Registry modification	RegCreateKey, RegDeleteValue	77
Synchronization	CreateMutex, CreateMutexEx	225
Memory allocation	HeapAlloc, GlobalMemoryStatus	126
		⋮
Device management	DeviceIoControl, DvdLauncher	37
		2,727

API Call Sequences of Known Malicious Activities:

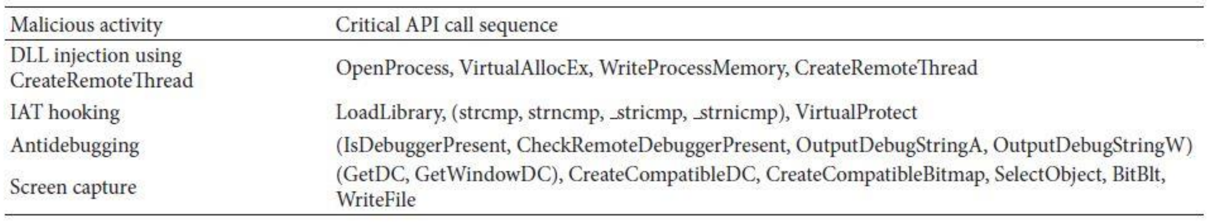
To improve the usefulness of API call sequences as features for detecting malware more accurately, an additional feature was employed; it was assumed that there exists a specific

API call sequence pattern that matches specific malicious activity. If such an API call sequence pattern exists, unknown attacks can be detected by checking whether a critical API call sequence pattern exists. To verify the existence of a critical API call sequence pattern of malware, profiles of the API call sequences of known attacks were created, which can be further used for unknown attack detection. Examples of critical API call sequence patterns found in our sample malware are shown below. It should be noted that round brackets represent the OR relation. For example, in the case of ***IAT hooking***, one possible critical API call sequence pattern is

[LoadLibrary, strcmp, VirtualProtect].

In this case, *strcmp* can be replaced with *strncmp* and it becomes another possible critical API call sequence pattern

[LoadLibrary, strncmp, VirtualProtect].



Malicious activity	Critical API call sequence
DLL injection using CreateRemoteThread	OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread
IAT hooking	LoadLibrary, (strcmp, strncmp, _stricmp, _strnicmp), VirtualProtect
Antidebugging	(IsDebuggerPresent, CheckRemoteDebuggerPresent, OutputDebugStringA, OutputDebugStringW)
Screen capture	(GetDC, GetWindowDC), CreateCompatibleDC, CreateCompatibleBitmap, SelectObject, BitBlt, WriteFile

(i) ***DLL injection***: Various methods for injecting Dynamic Link Library (DLL) into a target process exist, the most popular of which is to use *CreateRemoteThread* API, introduced by Jeffery Ritcher in the 1990s. First, the method gets a handle of the target process by using *OpenProcess*. Then, it allocates some space in the target process' memory using *VirtualAllocateEx* and writes the DLL's name, including full path, to the allocated memory by using *WriteProcessMemory*. Finally, it makes the target process reload the DLL using *CreateRemoteThread*.

(ii) ***IAT hooking***: The Import Address Table (IAT) contains the API's start address. By modifying the address, a different API can be called although the legitimate API is called. The process is as follows: First, it loads the target library using *LoadLibrary*. After finding the DLL from the IAT by comparing related APIs (e.g., *strcmp*), it modifies an attribute of the memory to be writable by using *VirtualProtect* and changes the address of the DLL.

(iii) ***Antidebugging***: There are many methods of detecting a debugging process using the API. For example, if the return value of the *IsDebuggerPresent* API is 1, this means that the program is being debugged. Antidebugging itself may not be a malicious activity, but it is often observed in malware.

(iv) ***Screen capture***: Backdoor often captures the screen and saves it as an image file. The process is as follows. First, it gets a handle of the window using an API, such as *GetDC*. Then, it creates a compatible space for saving the image using *CreateCompatibleDC* and *CreateCompatibleBitmap*. After choosing the image's pointer using *SelectObject*, it copies the image to the memory space using *BitBlt*. Finally, it writes the captured image as a file *usingWriteFile*.

UNDERSTANDING WEKA

WEKA is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from your own Java code. It contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization. It is also well-suited for developing new machine learning schemes.

The WEKA suite for machine learning is then used for processing the selected dataset. It is a free software licensed under the GNU General Public License and can be downloaded from

http://www.cs.waikato.ac.nz/~ml/weka/downloading.html

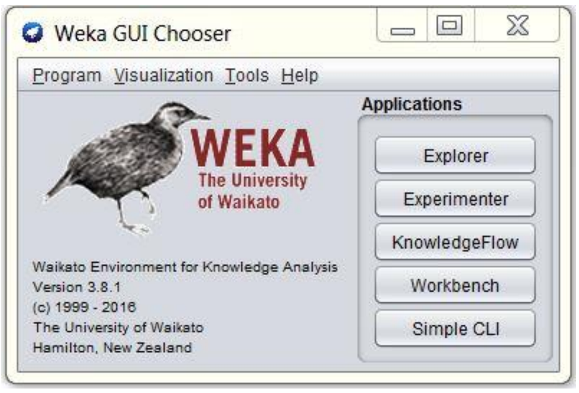
It provides extensive support for the whole process of experimental data mining, including preparing the input data, evaluating learning schemes statistically, and visualizing the input data and the result of learning. Along with a variety of learning algorithms, it includes a wide range of preprocessing tools. This diverse and comprehensive toolkit is accessed through a common interface (GUI) so that its users can compare different methods and identify those that are most appropriate for the problem at hand.

The system is written in **Java** and distributed under the terms of the GNU General Public License. It runs on almost any platform and has been tested under Linux, Windows, and Macintosh operating systems—and even on a personal digital assistant. It provides a uniform interface to many different learning algorithms, along with methods for pre- and postprocessing and for evaluating the result of learning schemes on any given dataset.

All algorithms take their input in the form of a single relational table in the **ARFF** (*Attribute-Relation File Format*) format which is an ASCII text file that describes a list of instances sharing a set of attributes. It consists of: A header (Describes the attribute types) and the Data section (Comma separated list of data).
Along with this, the **CSV** (*Comma-Separated Values*) file format is also quite useful during the implementation of datasets due to its ease of editing with MS-Excel.

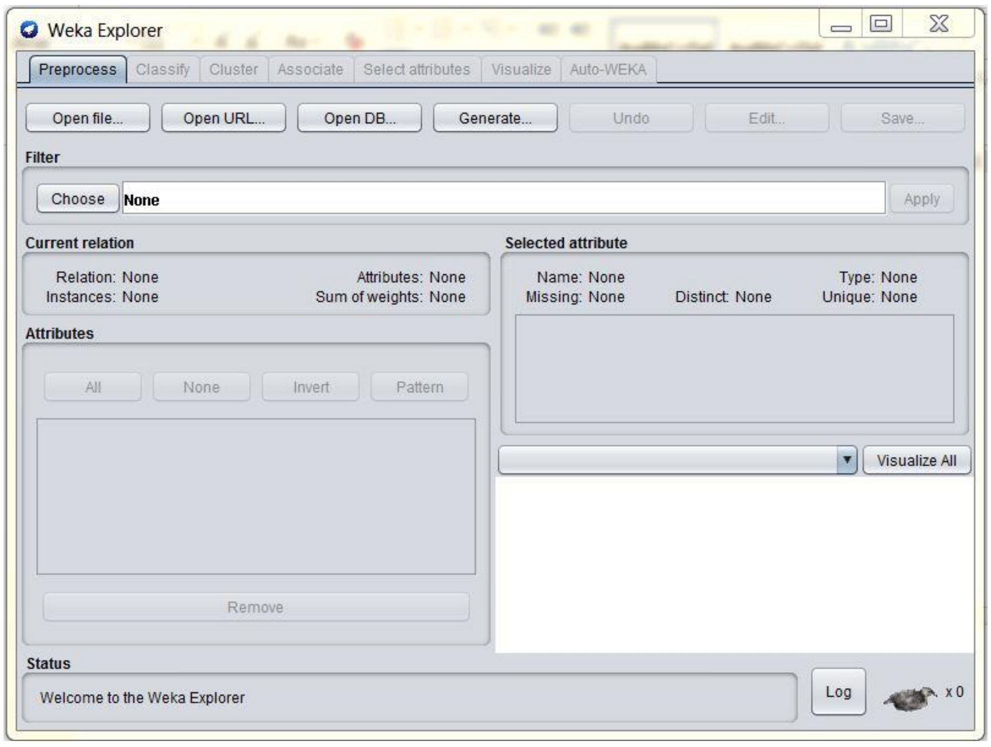
One way of using WEKA is to apply a learning method to a dataset and analyse its output to learn more about the data. Another is to use learned models to generate predictions on new instances. A third is to apply several different learners and compare their performance in order to choose one for prediction.
In the current scope of the study, we will be applying the various classification algorithms present on the selected dataset and comparing them with one another on the basis of the ***accuracy*** and ***precision*** each method displays while classifying the datasets instances as *benign* or *malign*, as well as factors such as, *F-**measure***, ***ROC area***, etc.

The primary screen for the WEKA suite, the 'WEKA GUI chooser' includes 5 interfaces: *Explorer*, *Experimenter*, *KnowledgeFlow*, *Workbench*, *Simple CLI* (Command Line Interface). Among these, the Explorer is the most used as it gives the user access to all of WEKA's facilities by the means of form filling and menu selection (i.e., buttons and drop-down menus)



Waikato Environment for Knowledge Analysis
Version 3.8.1
(c) 1999 - 2016
The University of Waikato
Hamilton, New Zealand

Selecting *Explorer*, the user is taken to the WEKA Explorer GUI interface which consists of 7 tabs: *Preprocessor*, *Classify*, *Cluster*, *Associate*, *Select Attributes*, *Visualise*, and *Auto-WEKA*.



Filter: None

Relation: None
Instances: None
Attributes: None
Sum of weights: None

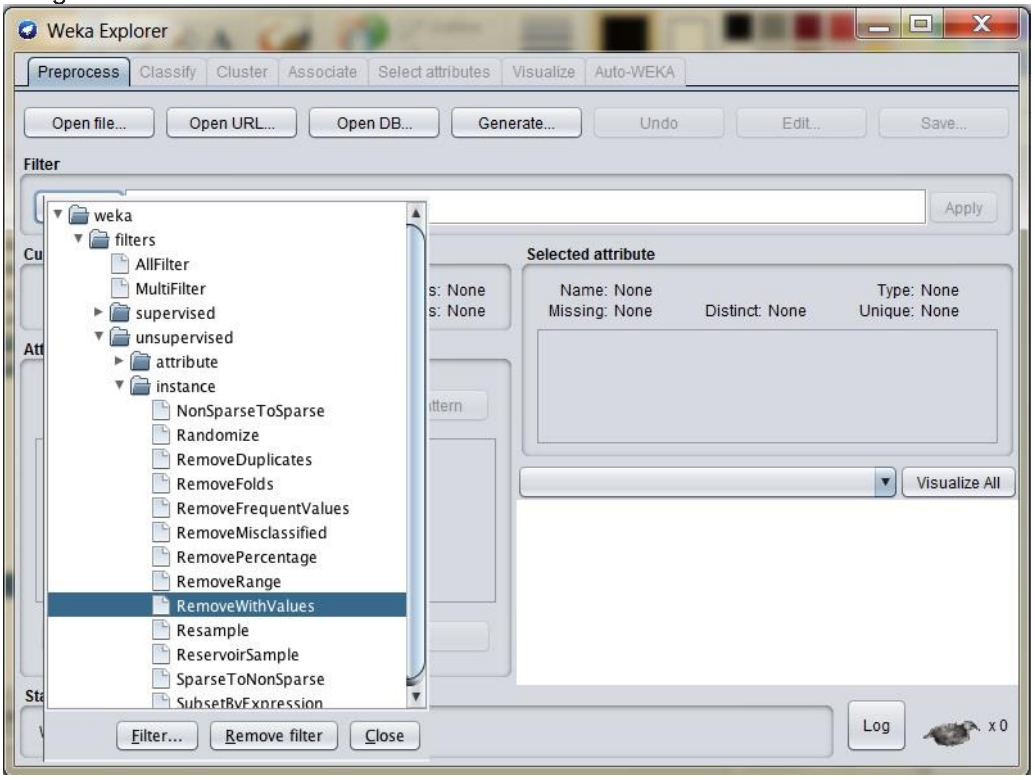
Name: None
Missing: None
Distinct: None
Type: None
Unique: None

Welcome to the Weka Explorer

The **<u>Preprocessor tab</u>** includes options for *opening* a dataset file (out of the various possible formats; and from different sources), *editing* the dataset entries, and *saving* the made changes as a new dataset file.
Also included is the option to choose a ***filter*** to be applied over the dataset in order to make it more precise and convenient for the classifier algorithms to process.

WEKA has a wide range of filters to be chosen from. They are firstly divided as ***Supervised*** and ***Unsupervised*** based on the status of <u>labelling</u> of the data instances present in the dataset

(If data entries are labelled, *Supervised*; otherwise, *Unsupervised*). Further, they are divided between *attribute* and *instance* filters based on whether the application of the selected filter has altering effects on the attribute or instance fields of the dataset.

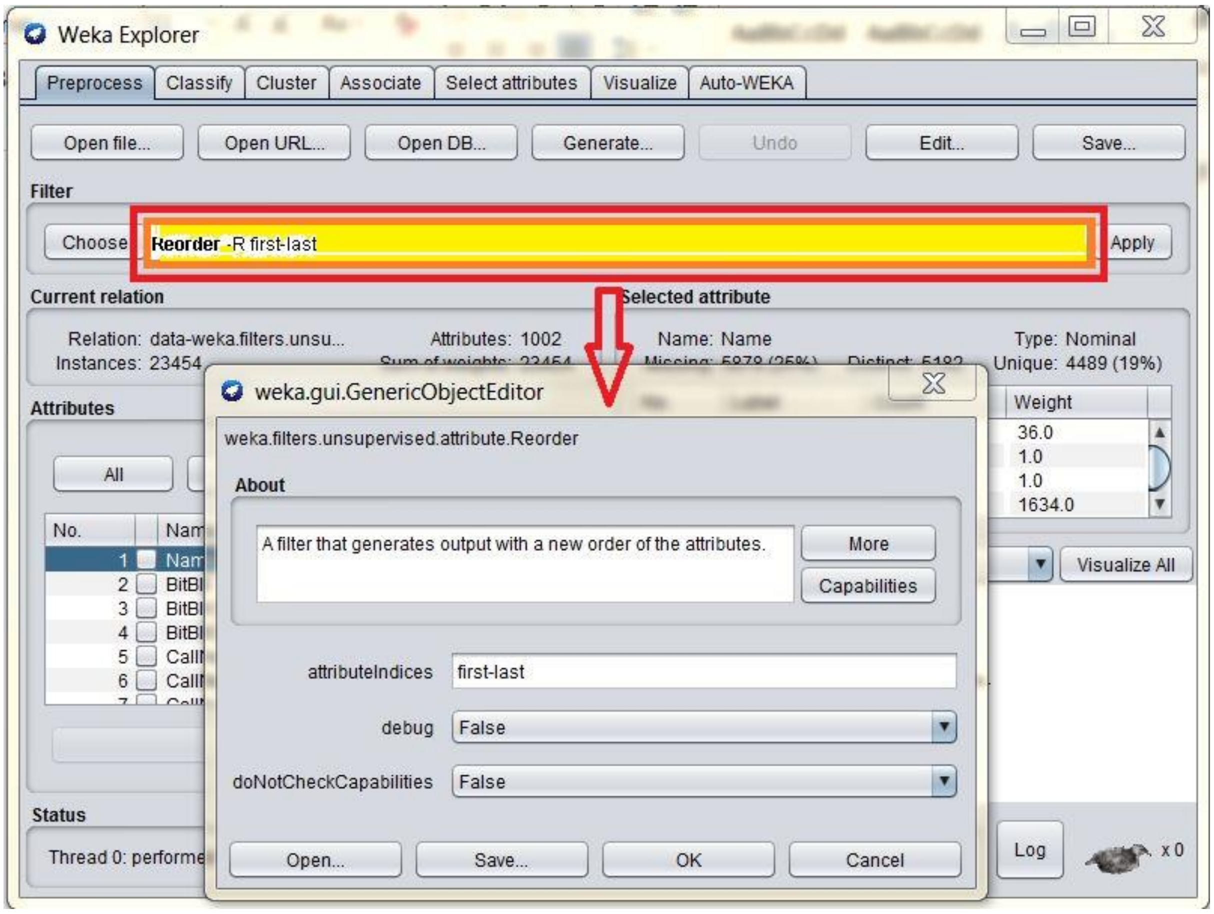


In practice, the unsupervised-attribute filters are used more often compared to other types. This is because real-life data is rarely labelled and consistent. Some regularly used filters are:

1. **Add**: An instance filter that adds a new attribute to the dataset. The new attribute will contain all missing values.
2. **AddExpression**: An instance filter that creates a new attribute by applying a mathematical expression to existing attributes. The expression can contain attribute references and numeric constants.
3. **Center**: Centers all numeric attributes in the given dataset to have zero mean (apart from the class attribute, if set).
4. **ClassAssigner**: Filter that can set and unset the class index.
5. **Copy**: An instance filter that copies a range of attributes in the dataset. This is used in conjunction with other filters that overwrite attribute values during the course of their operation -- this filter allows the original attributes to be kept as well as the new attributes.
6. **Discretize**: An instance filter that discretizes a range of numeric attributes in the dataset into nominal attributes. Discretization is by simple binning. Skips the class attribute if set.
7. **NominalToBinary**: Converts all nominal attributes into binary numeric attributes. An attribute with k values is transformed into k binary attributes if the class is nominal (using the one-attribute-per-value approach). Binary attributes are left binary, if option '-A' is not given. If the class is numeric, you might want to use the supervised version of this filter.
8. **NominalToString**: Converts a nominal attribute (i.e. set number of values) to string (i.e. unspecified number of values).
9. **Normalize**: Normalizes all numeric values in the given dataset (apart from the class attribute, if set). The resulting values are by default in [0,1] for the data used to compute the normalization intervals.
10. **NumericToBinary:** Converts all numeric attributes into binary attributes (apart from the class attribute, if set): if the value of the numeric attribute is exactly zero, the value of the new attribute will be zero. If the value of the numeric attribute is missing, the value of the new attribute will be missing. Otherwise, the value of the new attribute will be one. The new attributes will be nominal.

11. **NumericToNominal**: A filter for turning numeric attributes into nominal ones. Unlike discretization, it just takes all numeric values and adds them to the list of nominal values of that attribute. Useful after CSV imports, to enforce certain attributes to become nominal, e.g., the class attribute, containing values from 1 to 5.
12. **Obfuscate**: A simple instance filter that renames the relation, all attribute names and all nominal (and string) attribute values.
13. **RandomSubset**: Chooses a random subset of attributes, either an absolute number or a percentage. The class is always included in the output (as the last attribute).
14. **Remove**: A filter that removes a range of attributes from the dataset. Will re-order the remaining attributes if invert matching sense is turned on and the attribute column indices are not specified in ascending order.
15. **Reorder**: A filter that generates output with a new order of the attributes. Useful if one wants to move an attribute to the end to use it as class attribute.
    It's not only possible to change the order of all the attributes, but also to leave out attributes. The user can also duplicate attributes
    One can simply inverse the order of the attributes via 'last-first'. After applying the filter, the index of the class attribute is the last attribute.
16. **ReplaceMissingValues**: Replaces all missing values for nominal and numeric attributes in a dataset with the modes and means from the training data.
17. **StringToNominal**: Converts a string attribute (i.e. unspecified number of values) to nominal (i.e. set number of values). You should ensure that all string values that will appear are represented in the first batch of the data.
18. **StringToWordVector**: Converts String attributes into a set of attributes representing word occurrence (depending on the tokenizer) information from the text contained in the strings. The set of words (attributes) is determined by the first batch filtered (typically training data).
19. **SwapValues**: Swaps two values of a nominal attribute.

All of these filters can be further modified according to the desired effect by clicking on the selected filter and then changing the default settings of the filter.



Reorder -R first-last

Relation: data-weka.filters.unsu... Attributes: 1002
Instances: 23454

Name: Name Type: Nominal
Unique: 4489 (19%)

Weight
36.0
1.0
1.0
1634.0

No. Nam
1 Nam
2 BitBl
3 BitBl
4 BitBl
5 Call
6 Call

Thread 0: performe

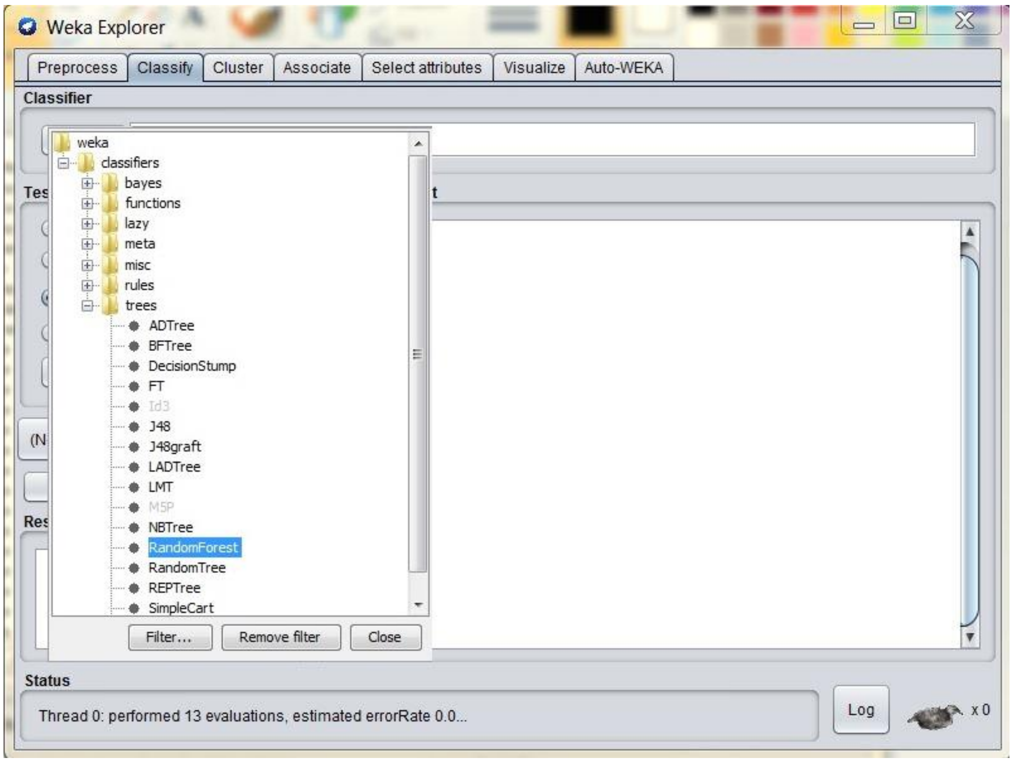
weka.filters.unsupervised.attribute.Reorder

A filter that generates output with a new order of the attributes.

attributeIndices first-last
debug False
doNotCheckCapabilities False

Moving on, the **Classify tab** contains all of the model-building algorithms needed for performing machine learning classifications using the selected dataset.

Classifiers in WEKA are the models for predicting nominal or numeric quantities. The learning schemes available in WEKA include decision *trees* and *lists*, *instance-based classifiers*, *support vector machines*, *multi-layer preceptrons*, *logistic regression*, and *bayes' nets*. "*Meta*"-classifiers include *bagging*, *boosting*, *stacking*, *error-correcting* output codes, and locally weighted learning



Thread 0: performed 13 evaluations, estimated errorRate 0.0...

We apply any of these classifiers on the satisfactorily formatted dataset to build a training model which will be used as the basis for any future classifications or predictions about an unknown executable sample, in order to declare it as *malign* or *benign*.

The classifiers are divided into *bayes*, *functions*, *lazy*, *meta*, *misc*, *rules*, and *trees*.

a) Bayes

1. ***NaiveBayes***
   Class for a Naive Bayes classifier using estimator classes. Numeric estimator precision values are chosen based on analysis of the training data. For this reason, the classifier is not an Updateable-Classifier (which in typical usage are initialized with zero training instances) – if there is a need for Updateable-Classifier functionality, use the *NaiveBayesUpdateable* classifier. The *NaiveBayesUpdateable* classifier will use a default precision of 0.1 for numeric attributes when *buildClassifier* is called with zero training instances.

   It assumes that all of the attributes contribute equally and statistically independently to the final decision, i.e., knowing the value of one of the attributes tells the system nothing about the value of another. This assumption, though never found to be actually true, is still used in practice.

2. ***NaiveBayesMultinomial***
   Class for building and using a multinomial Naive Bayes classifier.

$$P[C_i|D] = (P[D|C_i] \cdot P[C_i])/P[D] \text{ (Bayes rule)}$$

   where C_i is class i and D is a document.

3. ***NaiveBayesSimple***
   Class for building and using a simple Naive Bayes classifier. Numeric attributes are modelled by a normal distribution.

4. ***NaiveBayesUpdateable***
   Class for a Naive Bayes classifier using estimator classes. This is the update-able version of NaiveBayes. This classifier will use a default precision of 0.1 for numeric attributes when buildClassifier is called with zero training instances.

b) Functions

1. ***GaussianProcesses***
   Implements Gaussian Processes for regression without hyperparameter-tuning.

2. ***LinearRegression***
   Class for using linear regression for numeric predictions. Uses the Akaike criterion for model selection, and is able to deal with weighted instances (chosen so as to minimise the squared error on the training data).

3. ***Logistic***
   Class for building and using a multinomial logistic regression model with a ridge estimator.

   If there are k classes for n instances with m attributes, the parameter matrix B to be calculated will be an *m·(k − 1)* matrix. The probability for class *j* with the exception of the last class is

$$Pj(Xi) = \frac{\exp(X_i \cdot B_j)}{\left(\sum_{i=1}^{k-1} \exp(X_i \cdot B_j)\right) + 1}$$

   The last class has probability

$$1 - \left(\sum_{i=1}^{k-1} P_j(X_i)\right) = \frac{1}{\left(\sum_{i=1}^{k-1} exp(X_i * B_j)\right) + 1}$$

   Although original Logistic Regression does not deal with instance weights, we modify the algorithm a little bit to handle the instance weights.

4. ***MultilayerPerceptron***
   A Classifier that uses back propagation to classify instances. This network can be built by hand, created by an algorithm or both. The network can also be monitored and modified during training time. The nodes in this network are all sigmoid (except for when the class is numeric in which case the output nodes become unthresholded linear units).

5. ***SimpleLinearRegression***
   Learns a simple linear regression model. Picks the attribute that results in the lowest squared error. Missing values are not allowed. It can only deal with numeric attributes.

6. ***SimpleLogistic***
   Classifier for building linear logistic regression models.
   LogitBoost with simple regression functions as base learners is used for fitting the logistic models. The optimal number of *LogitBoost* iterations to perform is cross-validated, which leads to automatic attribute selection.

7. ***SGD***
   Implements stochastic gradient descent for learning various linear models (*binary class SVM*, *binary class logistic regression* and *linear regression*). It globally replaces all missing values and transforms nominal attributes into binary ones. It also normalizes all attributes, so the coefficients in the output are based on the normalized data. This implementation can be trained incrementally on (potentially) infinite data streams.

8. ***SGDText***
   Implements stochastic gradient descent for learning a linear binary class SVM or binary class logistic regression on text data. Operates directly on String attributes.

9. ***SMO***
   Implements John Platt's sequential minimal optimization algorithm for training a support vector classifier. This implementation globally replaces all missing values and transforms nominal attributes into binary ones. It also normalizes all attributes by default. (In that case the coefficients in the output are based on the normalized data, not the original data | this is important for interpreting the classifier.)
   Multi-class problems are solved using pairwise classification (1-vs-1 and if logistic models are built pairwise coupling).
   To obtain proper probability estimates, use the option that fits logistic regression models to the outputs of the support vector machine.
   In the multi-class case the predicted probabilities are coupled using Hastie and Tibshirani's pairwise coupling method.

10. ***SMOreg***
    Implements Alex Smola and Bernhard Scholkopf's sequential minimal optimization algorithm for training a support vector regression model. This implementation globally replaces all missing values and transforms nominal attributes into binary ones. It also normalizes all attributes by default. (Note that the coefficients in the output are based on the normalized/standardized data, not the original data.)

11. ***VotedPerceptron***
    Implementation of the voted perceptron algorithm by Freund and Schapire globally replaces all missing values, and transforms nominal attributes into binary ones.

c) Lazy

1. ***IBk***
   K-nearest-neighbour (KNN) classifier uses rote-learning technique and normalized Euclidean distance to find the 'k' training instances *closest* to the given test instance, and predicts the same class as those training instances. If multiple instances have the same (smallest) distance to the test instance, the first one found is used.
   The used learner does nothing until the moment it has to make to predictions.
   This nearest neighbour method produces a Linear Decision Binary.
   It is also helpful when dealing with *noisy* datasets where any of the 'wrong' nearest instances may end up being the closest to the test instance. In fact, the accuracy of the model improves in case of a *noisy* dataset as the value of 'k' increases to some non-extreme level.

   If in a dataset the number of training instances 'n' $\rightarrow \infty$, and also, k $\rightarrow \infty$,
   Such that,

   $$K/n \rightarrow 0,$$

   Then, the error of KNN method approaches a theoretical minimum for that dataset.

2. ***KStar***
   *K** is an instance-based classifier, that is the class of a test instance is based upon the class of those training instances similar to it, as determined by some similarity function. It differs from other instance-based learners in that it uses an entropy-based distance function.

3. ***LWL***
   Locally weighted learning uses an instance-based algorithm to assign instance weights which are then used by a specified *WeightedInstancesHandler*. It can perform classification (e.g. using naive Bayes) or regression (e.g. using linear regression).

d) Meta

1. ***AdaBoostM1***
   Class for boosting a nominal class classifier using the *AdaboostM1* method. Only nominal class problems can be tackled. It often dramatically improves performance, but sometimes ***overfits***.

2. ***AdditiveRegression***
   It is a meta-classifier that enhances the performance of a regression base classifier. Each iteration fits a model to the residuals left by the classifier on the previous iteration. Prediction is accomplished by adding the predictions of each classifier.
   Reducing the shrinkage (learning rate) parameter helps prevent *overfitting* and has a smoothing effect but increases the learning time.

3. ***AttributeSelectedClassifier***
   Dimensionality of training and test data is reduced by attribute selection before being passed on to a classifier.

4. ***Bagging***
   Class for bagging a classifier to reduce variance. It can do classification and regression depending on the base learner.
   It is used when production of several different decision structures is desired. This can be done by having several training sets of the same size by *sampling* the original set.
   A model is built for each of the several training sets using the same machine learning scheme, and then the predictions of these different models are combined by performing *voting* or averaging (in case of regression).
   This technique is very suitable for *unstable* learning schemes where a small change in the training dataset has a large change on the model.

5. ***ClassificationViaRegression***
   Class for doing classification using regression methods. Class is binarized and one regression model is built for each class value.

6. ***CostSensitiveClassifier***
   A meta-classifier that makes its base classifier cost-sensitive. Two methods can be used to introduce cost-sensitivity: reweighting training instances according to the total cost assigned to each class; or predicting the class with minimum expected misclassification cost (rather than the most likely class).
   Performance can often be improved by using a Bagged classifier to improve the probability estimates of the base classifier.

7. ***FilteredClassifier***
   Class for running an arbitrary classifier on data that has been passed through an arbitrary filter. Like the classifier, the structure of the filter is based exclusively on the training data and test instances will be processed by the filter without changing their structure.

8. ***LogitBoost***
   Class for performing additive logistic regression. This class performs classification using a regression scheme as the base learner, and can handle multi-class problems.

9. ***MultiClassClassifier***
   A meta-classifier for handling multi-class datasets with 2-class classifiers. This classifier is also capable of applying error correcting output codes for increased accuracy.

10. ***MultiClassClassifierUpdateable***
    A meta-classifier for handling multi-class datasets with 2-class classifiers. This classifier is also capable of applying error correcting output codes for increased accuracy.
    The base classifier must be an *updateable* classifier.

11. ***MultiScheme***
Class for selecting a classifier from among several using cross validation on the training data or the performance on the training data. Performance is measured based on percent correct (classification) or mean-squared error (regression).

12. ***RandomCommittee***
Class for building an ensemble of randomizable base classifiers. Each base classifiers is built using a different random number seed (but based on the same data). The final prediction is a straight average of the predictions generated by the individual base classifiers.

13. ***RandomSubSpace*** This method constructs a decision tree based classifier that maintains highest accuracy on training data and improves on generalization accuracy as it grows in complexity. The classifier consists of multiple trees constructed systematically by pseudo randomly selecting subsets of components of the feature vector, that is, trees constructed in randomly chosen subspaces.

14. ***Stacking***
Combines several classifiers using the stacking method. Can do classification or regression.
It uses *meta-learners* (instead of voting amongst learners) to combine the predictions of *base-learner* schemes (level-0 models; usually different machine learning schemes which are experts at one particular part only).
The combined output from base learners is used as input by the meta-learners.
It is quite difficult to make stacking work well as predictions on training datasets can't be used to generate data for the level-1 model. Instead, cross-validation-like schemes have to be used. To remedy this, ***StackingC*** was introduced which was found to be much more efficient compared to Stacking.

e) Rules

1. ***DecisionTable***
Class for building and using a simple decision table majority classifier.

2. ***JRip***
This class implements a propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (**RIPPER**), which was proposed by William W. Cohen as an optimized version of IREP.
The algorithm is briefly described as follows:
Initialize RS = , and for each class from the less prevalent one to the more frequent one, DO:

   1. Building stage: Repeat 1.1 and 1.2 until the description length (DL) of the ruleset and examples is 64 bits greater than the smallest DL met so far, or there are no positive examples, or the error rate _ 50%.
      (a) Grow phase: Grow one rule by greedily adding antecedents (or conditions) to the rule until the rule is perfect (i.e. 100% accurate). The procedure tries every possible value of each attribute and selects the condition with highest information gain: *p(log(p/t) - log(P/T)).*

(b) Prune phase: Incrementally prune each rule and allow the pruning of any final sequences of the antecedents; the pruning metric is

$$(p-n)/(p+n)$$

But it's actually

$$2p = (p+n) -1$$

So in this implementation, we simply use

$$p/(p+n)$$

[actually, (p+1) = (p+n+2), thus if $p+n$ is 0, it's 0:5]

2. Optimization stage: after generating the initial ruleset Ri, generate and prune two variants of each rule R_i from randomized data using procedure (a) and (b). But one variant is generated from an empty rule while the other is generated by greedily adding antecedents to the original rule. Moreover, the pruning metric used here is

$$(TP + TN) = (P + N).$$

Then the smallest possible DL for each variant and the original rule is computed. The variant with the minimal DL is selected as the final representative of R_i in the ruleset.
After all the rules in R_i have been examined and if there are still residual positives, more rules are generated based on the residual positives using Building Stage again.

3. Delete the rules from the ruleset that would increase the DL of the whole ruleset if it were in it. And add resultant ruleset to RS.

Note that there seem to be 2 bugs in the original ripper program that would affect the ruleset size and accuracy slightly. This implementation avoids these bugs and thus is a little bit different from Cohen's original implementation. Even after fixing the bugs, since the order of classes with the same frequency is not defined in ripper, there still seems to be some trivial difference between this implementation and the original ripper, especially for audiology data in UCI repository, where there are lots of classes of few instances.

3. ***M5Rules***
Generates a decision list for regression problems using separate-and-conquer. In each iteration it builds a model tree using M5 and makes the "best" leaf into a rule.

4. ***OneR***
Class for building and using a 1R classifier; in other words, uses the minimum-error attribute for prediction, discretizing numeric attributes.
It assumes that *one attribute* is responsible for the entire built model, and that it is this attribute's value that dictates the result.
It works by learning a 1-level decision tree, and gives a set of rules, all of which test for the one particular attribute at each instance of the test dataset

5. ***PART***
Class for generating a PART decision list. Uses separate-and-conquer. Builds a partial C4.5 decision tree in each iteration and makes the "best" leaf into a rule.

6. ***ZeroR***
   Class for building and using a 0-R classifier. Predicts the mean (for a numeric class) or the mode (for a nominal class).
   This classifier is also known as the ***Baseline classifier***, and it looks for the most popular (highest occurrence) class and selects it as its querying guess for each instance of the test dataset.

f) Trees

1. ***DecisionStump***
   Class for building and using a decision stump. Usually used in conjunction with a boosting algorithm. Does regression (based on mean-squared error) or classification (based on entropy). Missing is treated as a separate value.

2. ***HoeffdingTree***
   A *Hoeffding tree (VFDT)* is an incremental, anytime decision tree induction algorithm that is capable of learning from massive data streams, assuming that the distribution generating examples does not change over time. Hoeffding trees exploit the fact that a small sample can often be enough to choose an optimal splitting attribute. This idea is supported mathematically by the Hoeffding bound, which quantities the number of observations (in our case, examples) needed to estimate some statistics within a prescribed precision (in our case, the goodness of an attribute).
   A theoretically appealing feature of Hoeffding Trees not shared by other incremental decision tree learners is that it has sound guarantees of performance. Using the Hoeffding bound one can show that its output is asymptotically nearly identical to that of a non-incremental learner using infinitely many examples.

3. ***J48***
   Class for generating a pruned or unpruned C4.5 decision tree.
   It is based on *recursive divide and conquer* in which the user selects the attribute to make the split on at the root node of the tree, and create a branch for each respective value of the attribute. This splitting leads to formation of subsets of the instances. This procedure is repeated for each node of each of the branches present.
   According to the classifier, a ‘good’ attribute to be used as a root node is the one which is ‘purest’ (either all-YESs or al-NOs) amongst all the other attributes present at that level of the tree.

   The aim of the J48 classifier is to produce the smallest possible tree using the heuristic function, which according to the Information theory, quantify the *entropy* of the model’s results.

   $$[entropy(P_1,P_2,P_3,\ldots P_n) = -P_1 logP_1 - P_2 logP_2 - \ldots - P_n logP_n]$$

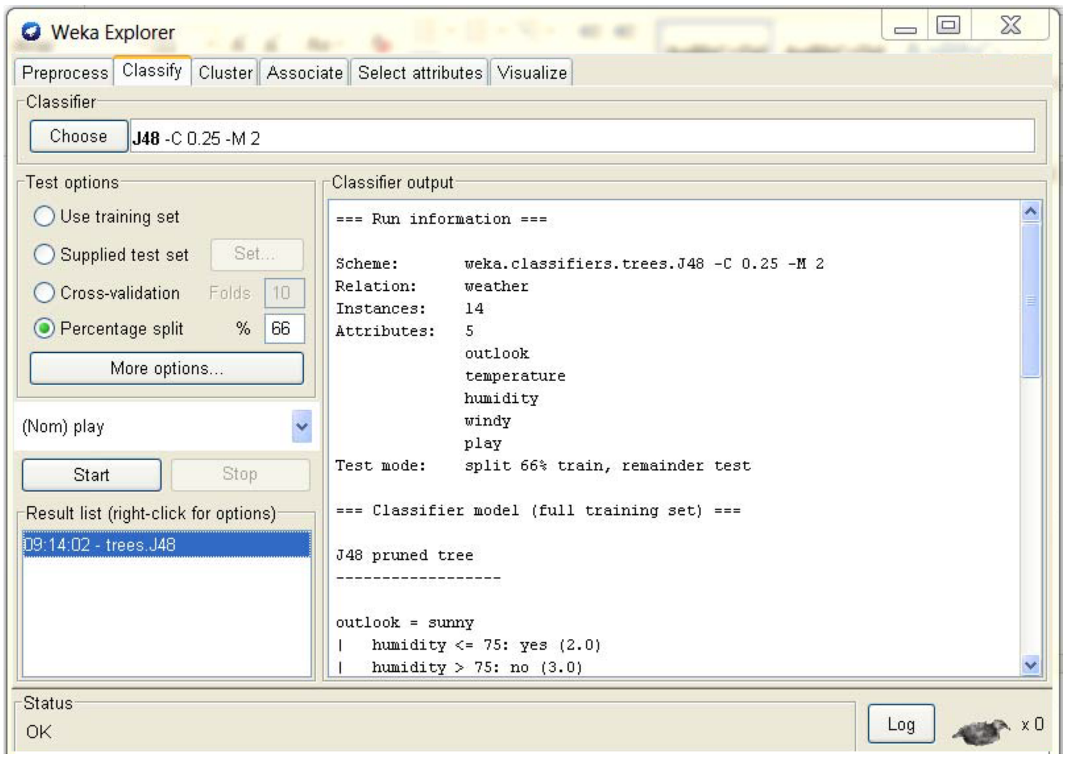
   Information gain is the amount of information gained by knowing the value of the attributes.
   The attribute which gains the most bits of information, i.e., whose information gain is the highest is selected as the root node.

Despite this, to allow proper use of J48 classifier in practice, the dataset that is to be used needs to be ***pruned*** first.

4. ***<u>LMT</u>***
   Classifier used for building 'logistic model trees', which are classification trees with logistic regression functions at the leaves. The algorithm can deal with binary and multi-class target variables, numeric and nominal attributes and missing values.

5. ***<u>M5P</u>***
   M5Base implements base routines for generating <u>M5 Model trees</u> and rules. The original algorithm, *M5* was invented by R. Quinlan and Yong Wang made improvements.

WEKA also allows the user to **visualise** a graphical representation of the classification tree.



J48 -C 0.25 -M 2

Use training set, Supplied test set, Cross-validation Folds 10, [✓ Percentage split] % 66

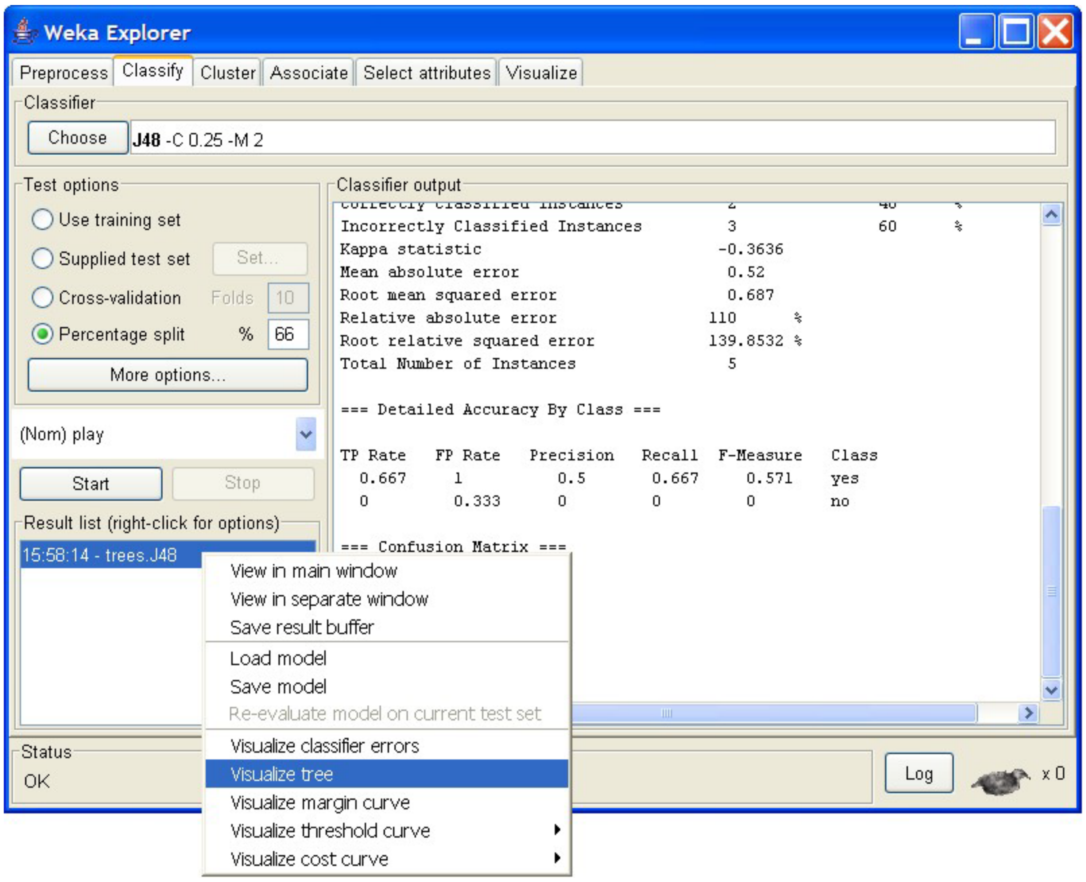
(Nom) play

09:14:02 - trees.J48

Status: OK

```
=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
Relation:     weather
Instances:    14
Attributes:   5
              outlook
              temperature
              humidity
              windy
              play
Test mode:    split 66% train, remainder test

=== Classifier model (full training set) ===

J48 pruned tree
------------------

outlook = sunny
|   humidity <= 75: yes (2.0)
|   humidity > 75: no (3.0)
```

Selecting the item '*Visualize tree*', a new window comes up to the screen displaying the tree.



J48 -C 0.25 -M 2

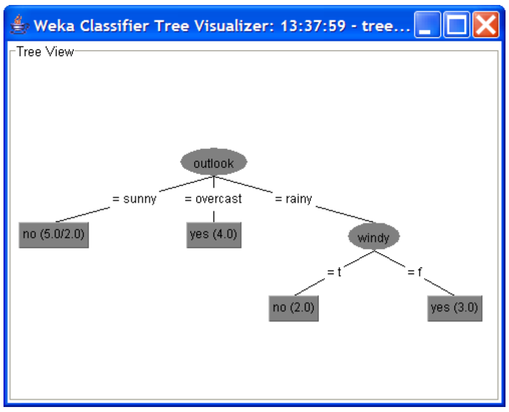
Use training set, Supplied test set, Cross-validation Folds 10, [✓ Percentage split] % 66

(Nom) play

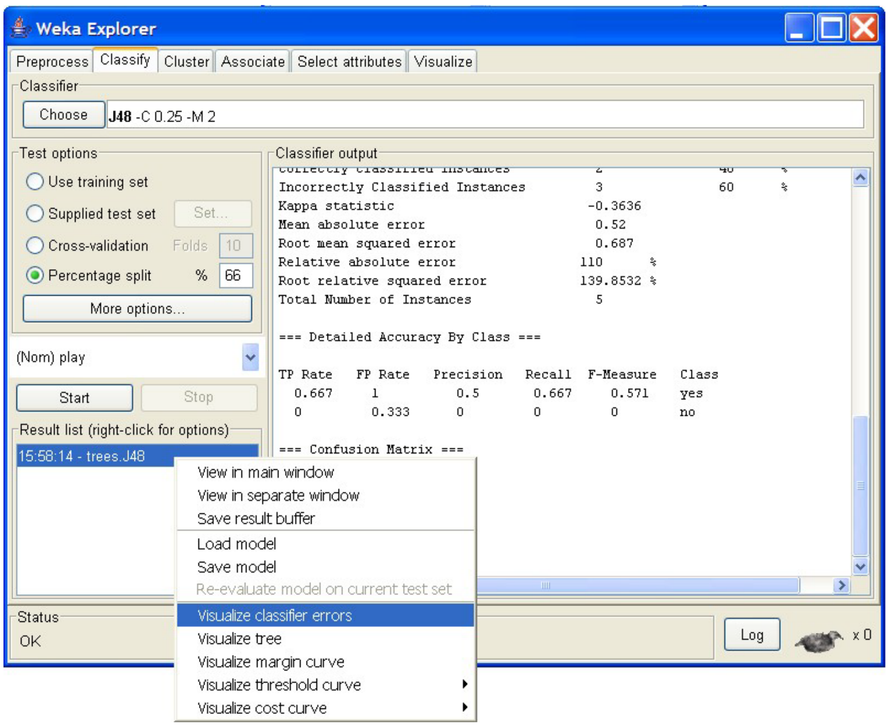
15:58:14 - trees.J48

Status: OK

```
Correctly Classified Instances          2               40      %
Incorrectly Classified Instances        3               60      %
Kappa statistic                        -0.3636
Mean absolute error                     0.52
Root mean squared error                 0.687
Relative absolute error               110      %
Root relative squared error           139.8532 %
Total Number of Instances               5

=== Detailed Accuracy By Class ===

TP Rate   FP Rate   Precision   Recall  F-Measure   Class
  0.667     1         0.5        0.667     0.571    yes
  0         0.333     0          0         0        no

=== Confusion Matrix ===
```



WEKA also allows to visualize classification errors.



J48 -C 0.25 -M 2

Folds 10 · [✓ Percentage split] % 66

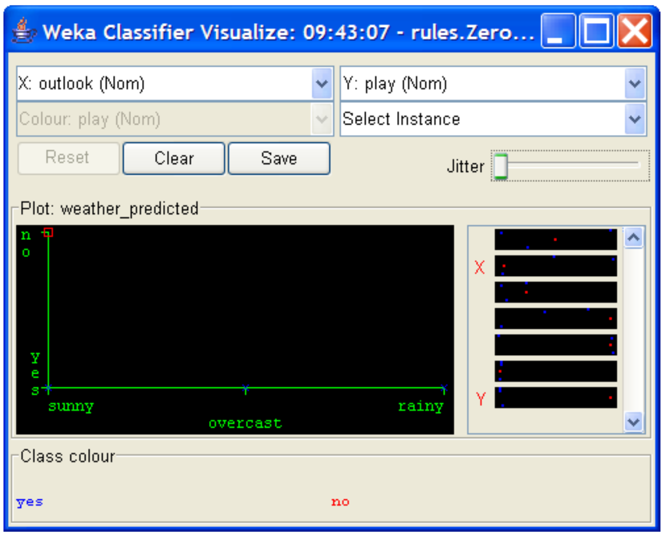
(Nom) play

15:58:14 - trees.J48

Status: OK

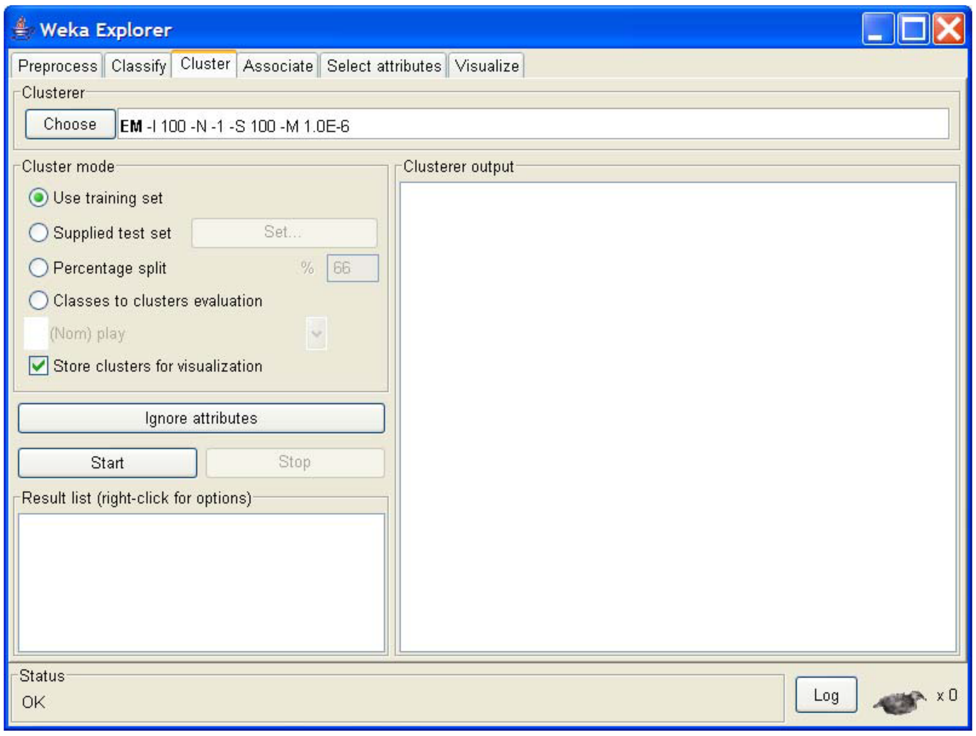
```
Correctly Classified Instances           2               40      %
Incorrectly Classified Instances         3               60      %
Kappa statistic                         -0.3636
Mean absolute error                      0.52
Root mean squared error                  0.687
Relative absolute error                110      %
Root relative squared error            139.8532 %
Total Number of Instances                5

=== Detailed Accuracy By Class ===

TP Rate   FP Rate   Precision   Recall  F-Measure   Class
  0.667     1          0.5       0.667     0.571    yes
  0         0.333      0         0         0        no

=== Confusion Matrix ===
```

WEKA Classifier Visualize' window displaying graph appears on the screen.



WEKA contains “**<u>clusterers</u>**” for finding groups of similar instances in a dataset. The clustering schemes available in WEKA are *k-Means*, *EM*, *Cobweb*, *X-means*, and *FarthestFirst*.

Clusters can be visualized and compared to “true” clusters (if given). Evaluation is based on log-likelihood if clustering scheme produces a probability distribution.

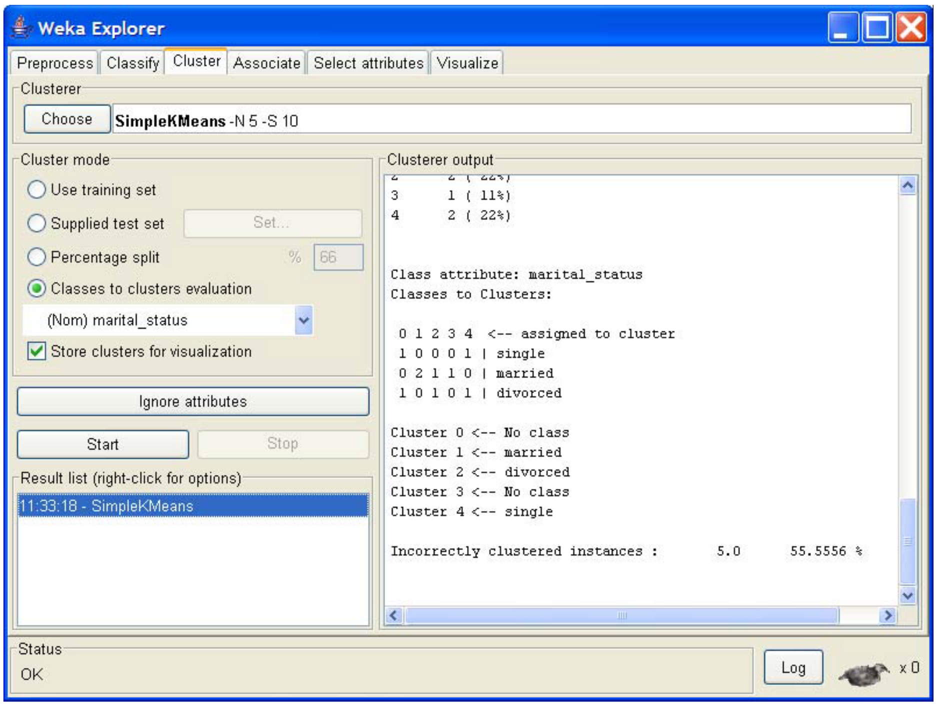


Clusterer: EM -I 100 -N -1 -S 100 -M 1.0E-6

Cluster mode: [✓ Use training set]; Percentage split % 66; (Nom) play; [✓ Store clusters for visualization]

Status: OK

When training set is complete, the ‘Cluster’ output area on the right panel of ‘Cluster’ window is filled with text describing the results of training and testing. A new entry appears in the ‘Result list’ box on the left of the result. These behave just like their classification counterparts



Clusterer: SimpleKMeans -N 5 -S 10

Cluster mode: [✓ Classes to clusters evaluation]; Percentage split % 66; (Nom) marital_status; [✓ Store clusters for visualization]

Result list: 11:33:18 - SimpleKMeans

```
3      1 ( 11%)
4      2 ( 22%)


Class attribute: marital_status
Classes to Clusters:

 0 1 2 3 4  <-- assigned to cluster
 1 0 0 0 1 | single
 0 2 1 1 0 | married
 1 0 1 0 1 | divorced

Cluster 0 <-- No class
Cluster 1 <-- married
Cluster 2 <-- divorced
Cluster 3 <-- No class
Cluster 4 <-- single

Incorrectly clustered instances :      5.0      55.5556 %
```

Status: OK

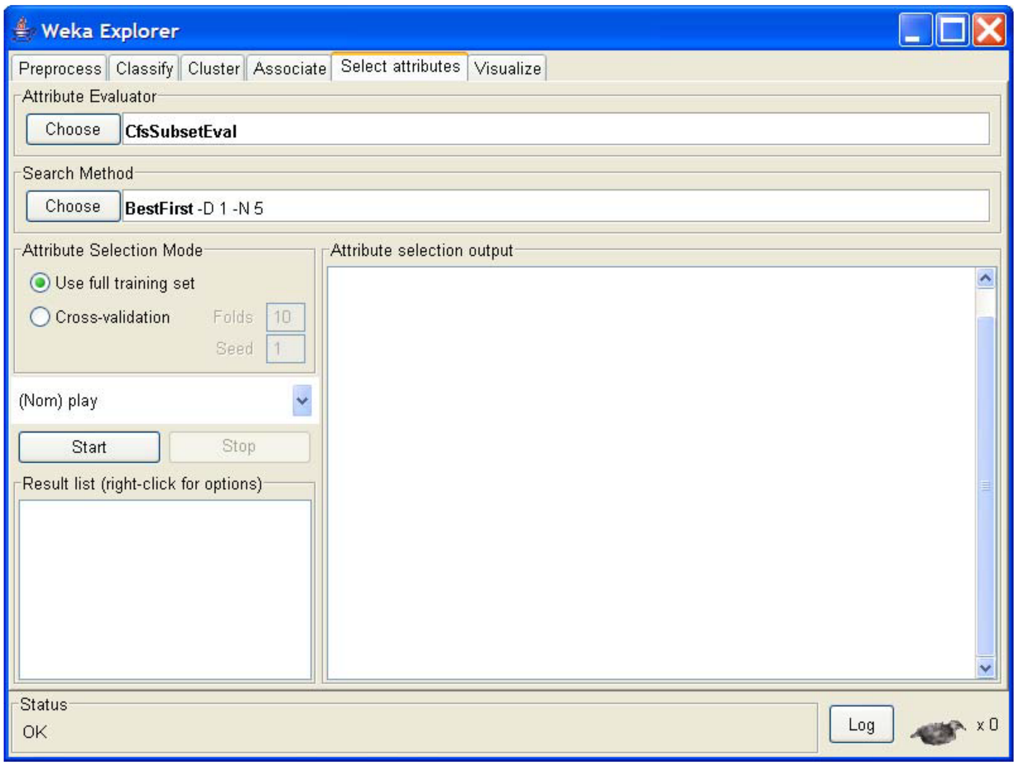
WEKA contains an implementation of the *Apriori algorithm* for learning **association** rules. This is the only currently available scheme for learning associations in WEKA. It works only with discrete data and will identify statistical dependencies between groups.
Apriori can compute all rules that have a given minimum support and exceed a given confidence.
The association rule scheme cannot handle numeric values

WEKA also has the capability of performing **Attribute selection** searches through all possible combinations of attributes in the data and finds which subset of attributes works *best* for prediction.

Attribute selection methods contain two parts:

1. A search method such as *best-first*, *forward selection*, *random*, *exhaustive*, *genetic algorithm*, *ranking*, and,
2. An evaluation method such as *correlation-based*, *wrapper*, *information gain*, *chi-squared*.

Attribute selection mechanism is very flexible - WEKA allows (almost) arbitrary combinations of the two methods



Attribute Evaluator: CfsSubsetEval

Search Method: BestFirst -D 1 -N 5

Attribute Selection Mode
- [✓ Use full training set]
- Cross-validation — Folds 10, Seed 1

(Nom) play

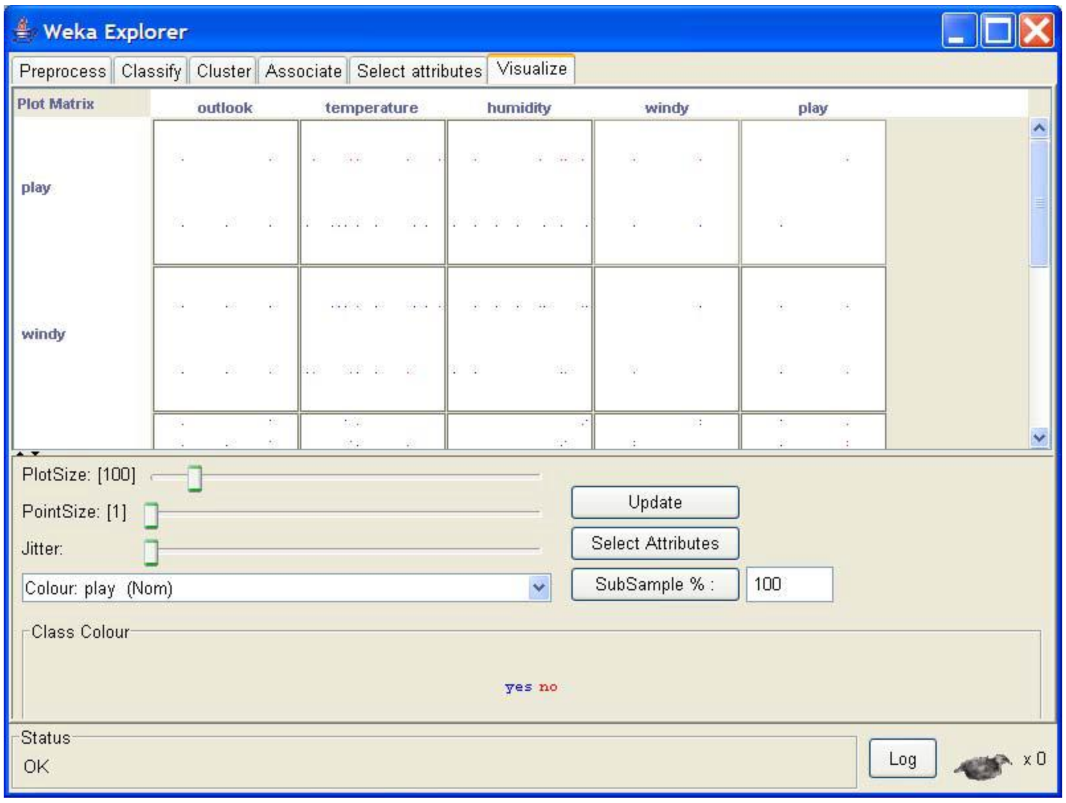
Attribute selection output

Result list (right-click for options)

Status
OK

x 0

WEKA's **visualization** allows the users to visualize a 2-D plot of the current working relation.
Visualization is very useful in practice, it helps to determine difficulty of the learning problem. WEKA can visualize single attributes (1-d) and pairs of attributes (2-d), rotate 3-d visualizations (Xgobi-style). WEKA has "Jitter" option to deal with nominal attributes and to detect "hidden" data points.



PlotSize: [100]
PointSize: [1]
Jitter:
Colour: play (Nom)
SubSample % : 100
Status
OK

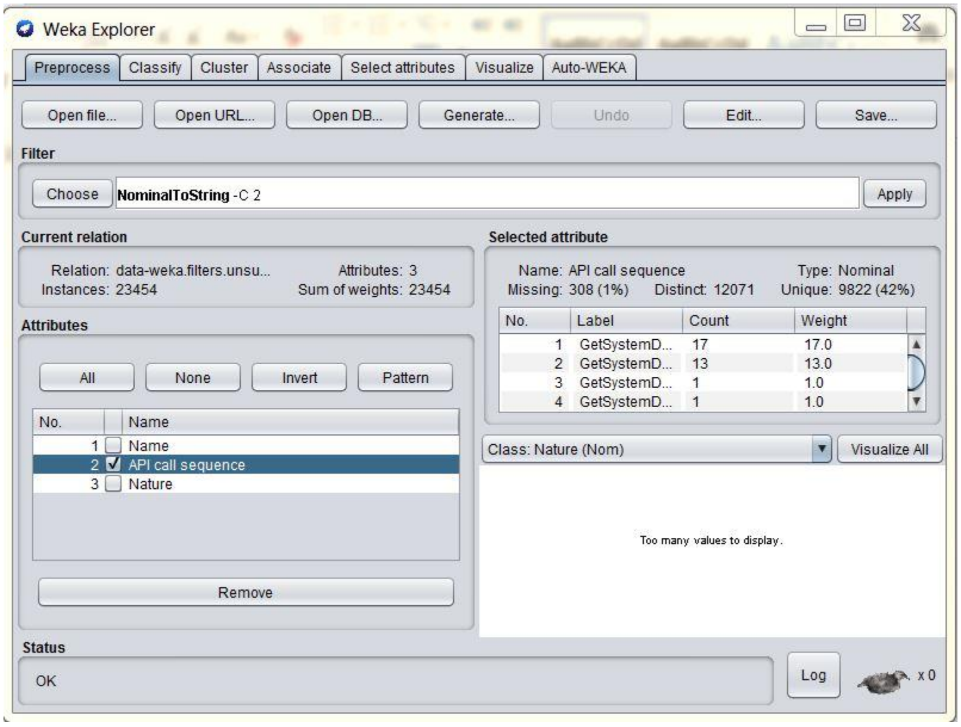
In the *visualization* window, beneath the X-axis selector there is a drop-down list, 'Colour', for choosing the *colour scheme*. This allows users to choose the colour of points based on the attribute selected. Below the plot area, there is a legend that describes what values the colours correspond to. For better visibility the user should change the colour of a label by left-clicking on the label in the 'Class colour' box and selecting a lighter colour from the colour palette.
To the right of the plot area there are series of horizontal strips. Each strip represents an attribute, and the dots within it show the distribution values of the attribute. You can choose what axes are used in the main graph by clicking on these strips (left-click changes X-axis, right click changes Y-axis).

IMPLIMENTATION

A system with 8 GB of RAM, with a 2.8GHz processor and Windows7 operating sys was used for performing the required steps to make the necessary classification during this project.
As discussed earlier, the dataset used for the project includes **23,387** randomly chosen executables (23,080 *malware* samples and 307 *benign* samples) from the malware dataset of the Malicia-project and Virustotal.

Waikato Environment for Knowledge Analysis
Version 3.8.1
(c) 1999 - 2016
The University of Waikato
Hamilton, New Zealand

Firstly, the dataset file (in .arff format), is opened in the WEKA Explorer. After using the *Remove* filter on any unnecessary attributes, the 'API call sequence' attribute entries were converted to strings by the use of the *NominalToString* filter.



Filter
NominalToString -C 2

Current relation
Relation: data-weka.filters.unsu... Attributes: 3
Instances: 23454 Sum of weights: 23454

Selected attribute
Name: API call sequence Type: Nominal
Missing: 308 (1%) Distinct: 12071 Unique: 9822 (42%)

No.	Label	Count	Weight
1	GetSystemD...	17	17.0
2	GetSystemD...	13	13.0
3	GetSystemD...	1	1.0
4	GetSystemD...	1	1.0

Attributes

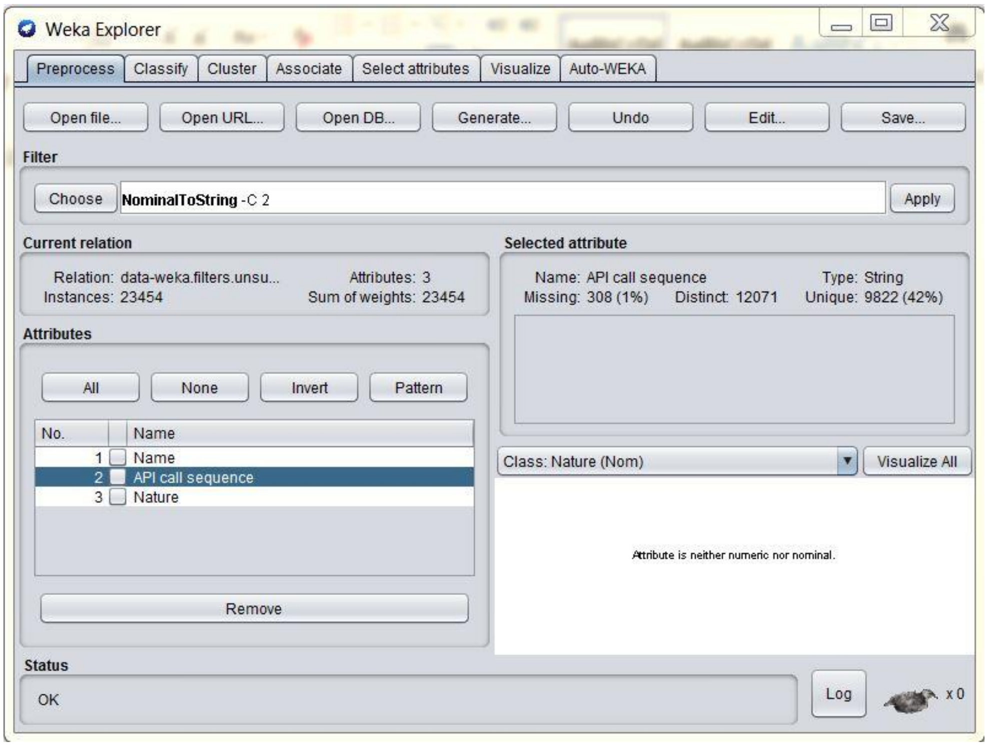
No.	Name
1	☐ Name
2	☑ API call sequence
3	☐ Nature

Class: Nature (Nom)

Too many values to display.

Status
OK

This lead to the different calls becoming readable to WEKA.
The changes can be viewed in the screenshot below. The calls are no longer displayed as separate entries for the numerous instances. Instead, they have now been converted to singular strings for each respective instance.


Weka Explorer

Preprocess | Classify | Cluster | Associate | Select attributes | Visualize | Auto-WEKA

Filter: **NominalToString** -C 2

Current relation

Relation: data-weka.filters.unsu... Attributes: 3
Instances: 23454 Sum of weights: 23454

Selected attribute

Name: API call sequence Type: String
Missing: 308 (1%) Distinct: 12071 Unique: 9822 (42%)

Attributes

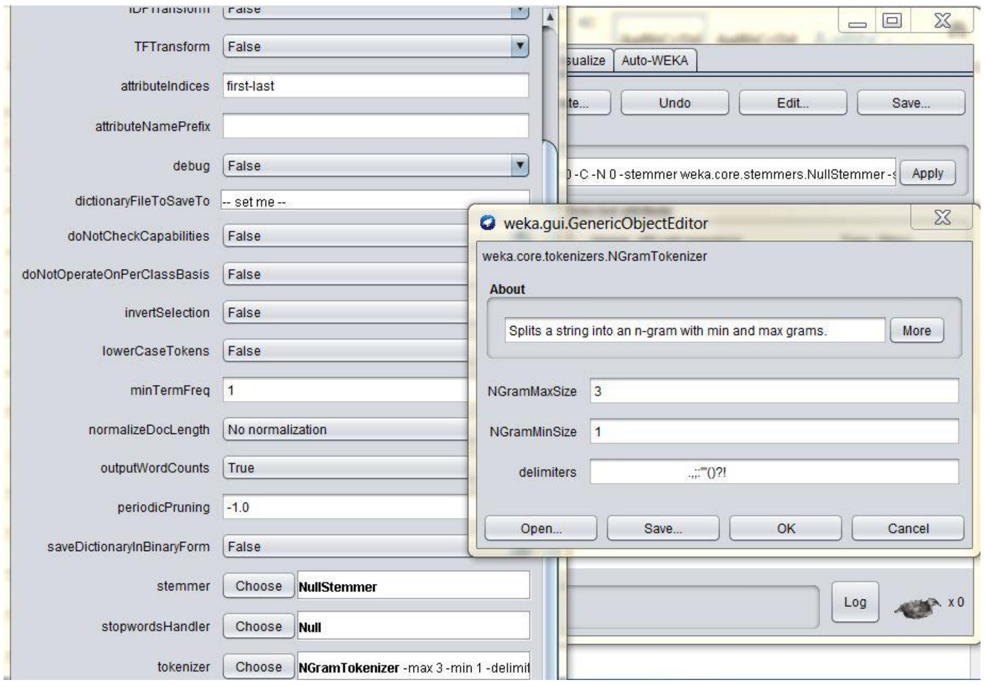
No.	Name
1	Name
2	API call sequence
3	Nature

Class: Nature (Nom)

Attribute is neither numeric nor nominal.

Status: OK

The counts of the API calls were then procured by the use of the *StringToWordVector* filter along with *NGramTokenizer* (with a maximum gram size of 3).


TFTransform	False
attributeIndices	first-last
attributeNamePrefix	
debug	False
dictionaryFileToSaveTo	-- set me --
doNotCheckCapabilities	False
doNotOperateOnPerClassBasis	False
invertSelection	False
lowerCaseTokens	False
minTermFreq	1
normalizeDocLength	No normalization
outputWordCounts	True
periodicPruning	-1.0
saveDictionaryInBinaryForm	False
stemmer	**NullStemmer**
stopwordsHandler	**Null**
tokenizer	**NGramTokenizer** -max 3 -min 1 -delimit

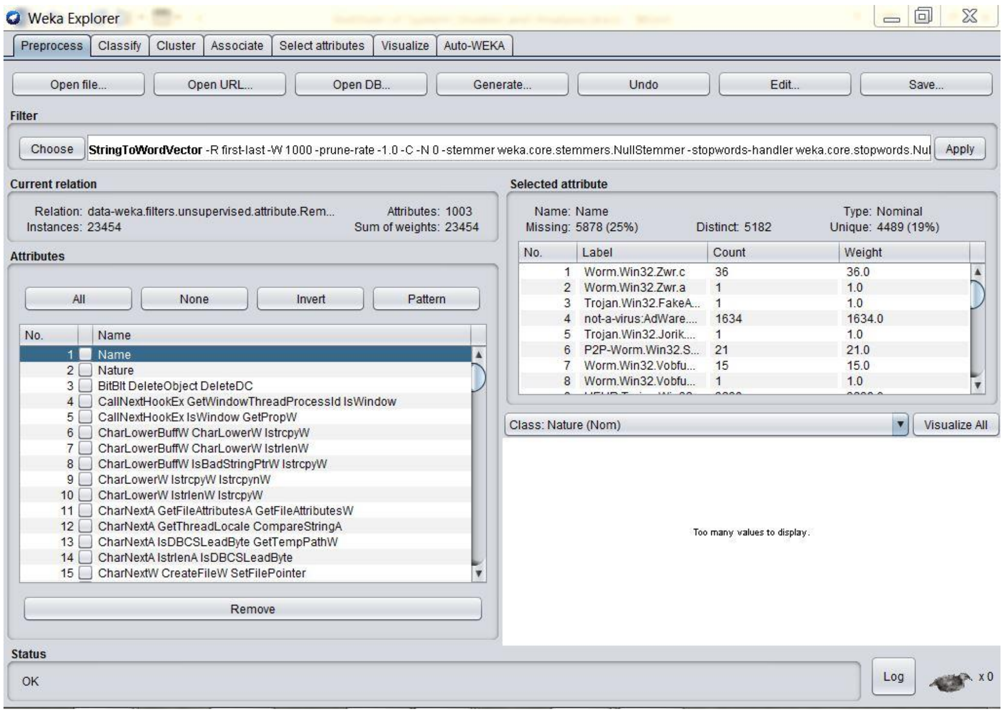
weka.gui.GenericObjectEditor

weka.core.tokenizers.NGramTokenizer

About: Splits a string into an n-gram with min and max grams.

NGramMaxSize	3
NGramMinSize	1
delimiters	.,;:'"()?!

Applying *StringToWordVector*, all the individual calls were counted in a groups of 3 to imply a sequence of calls which would establish the nature of that sequence's presence in later executables.



Filter

StringToWordVector -R first-last -W 1000 -prune-rate -1.0 -C -N 0 -stemmer weka.core.stemmers.NullStemmer -stopwords-handler weka.core.stopwords.Nul

Current relation

Relation: data-weka.filters.unsupervised.attribute.Rem... Attributes: 1003
Instances: 23454 Sum of weights: 23454

Attributes

No.	Name
1	Name
2	Nature
3	BitBlt DeleteObject DeleteDC
4	CallNextHookEx GetWindowThreadProcessId IsWindow
5	CallNextHookEx IsWindow GetPropW
6	CharLowerBuffW CharLowerW lstrcpyW
7	CharLowerBuffW CharLowerW lstrlenW
8	CharLowerBuffW IsBadStringPtrW lstrcpyW
9	CharLowerW lstrcpyW lstrcpynW
10	CharLowerW lstrlenW lstrcpyW
11	CharNextA GetFileAttributesA GetFileAttributesW
12	CharNextA GetThreadLocale CompareStringA
13	CharNextA IsDBCSLeadByte GetTempPathW
14	CharNextA lstrlenA IsDBCSLeadByte
15	CharNextW CreateFileW SetFilePointer

Selected attribute

Name: Name Type: Nominal
Missing: 5878 (25%) Distinct: 5182 Unique: 4489 (19%)

No.	Label	Count	Weight
1	Worm.Win32.Zwr.c	36	36.0
2	Worm.Win32.Zwr.a	1	1.0
3	Trojan.Win32.FakeA...	1	1.0
4	not-a-virus:AdWare....	1634	1634.0
5	Trojan.Win32.Jorik....	1	1.0
6	P2P-Worm.Win32.S...	21	21.0
7	Worm.Win32.Vobfu...	15	15.0
8	Worm.Win32.Vobfu...	1	1.0

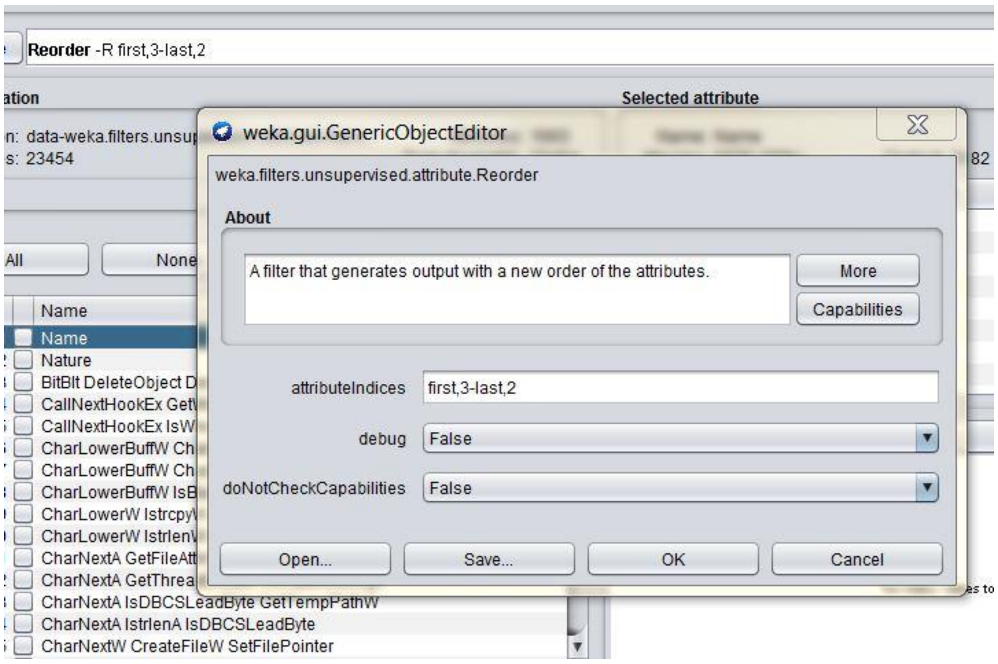
Class: Nature (Nom)

Too many values to display.

Status

OK

Once this was done, the dataset was reordered using the Reorder filter



Reorder -R first,3-last,2

weka.gui.GenericObjectEditor

weka.filters.unsupervised.attribute.Reorder

About

A filter that generates output with a new order of the attributes.

attributeIndices first,3-last,2

debug False

doNotCheckCapabilities False

Once this is done, the dataset can be used for performing classification.

The datasets is classified using a *percentage-split of 90%.* That is, 90 percent of the original dataset is used for the purpose of training the model which is in turn used to classify the remaining 10 percent of the dataset which has been randomly chosen to act as Test set.

Since the objective of this report is to find a classification technique which can be considered to be the best amongst the plethora of methods available in WEKA, a comprehensive comparison is performed on the basis of factors such as ***F-Measure***, ***ROC Area*** and the ***Accuracy*** of prediction displayed by the classifier. The results can be viewed in the following table.

Deciding factors	Used Classifiers									
	ZeroR	*PART*	*OneR*	***JRip***	***RandomForest***	*J48*	*DecisionStump*	*IBk*	*SMO*	*NaiveBayes*
F-Measure	0.985	0.999	0.985	**1.000**	**1.000**	0.999	0.997	0.999	1.000	0.998
ROC Area	0.500	1.000	0.500	**1.000**	**1.000**	1.000	0.998	1.000	1.000	1.000
Accuracy (%)	98.976	99.914	98.976	**100**	**100**	99.914	99.658	99.944	99.987	99.769

It is clear from the above readings that the *JRip* and *RandomForest* classifiers are best when it comes to making predictive classification of novel, unknown executables based on a model made according to the dataset that had been chosen earlier.

Despite this seeming equality in final results, the ***RandomForest*** classifier can clearly be considered to be the better of the two, and hence is the one chosen.

CODING

The following are the respective JAVA codes for the aforementioned filters and classifiers used. These were executed using JAVA 8 JDK.

1. To convert nominal entries to string using NominalToString filter.

```
package weka.api;

//import required classes
import weka.core.Instances;
import weka.core.converters.ArffSaver;
import java.io.File;
import weka.core.converters.ConverterUtils.DataSource;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.NominalToString;

public class Attfilter{
    public static void main(String args[]) throws Exception{
        //load dataset
        DataSource source = new DataSource("E:\\source\\location\\of\\dataset\\data2.arff");
        Instances dataset = source.getDataSet();

        //use a simple filter to Nom2Str a certain attribute
        //set up options to Nom2Str 2nd attribute
        String[] opts = new String[]{ "-C", "2"};
        //create a NominalToString object (this is the filter class)
        NominalToString Nom2Str = new NominalToString();
        //set the filter options
        Nom2Str.setOptions(opts);
        //pass the dataset to the filter
        Nom2Str.setInputFormat(dataset);
        //apply the filter
        Instances newData = Filter.useFilter(dataset, Nom2Str);

        //now save the dataset to a new file as we learned before
        ArffSaver saver = new ArffSaver();
        saver.setInstances(newData);
        saver.setFile(new File("E:\\destination\\folder\\address\\datastringed.arff"));
        saver.writeBatch();
    }
}
```

2. <u>To convert the API Calls from strings to countable form using StringToWordVector filter.</u>

```
package weka.api;

//import required classes
import weka.core.Instances;
import weka.core.converters.ArffSaver;
import weka.core.stemmers.*;
import weka.core.tokenizers.*;
import java.io.File;
import weka.core.converters.ConverterUtils.DataSource;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.*;

public class StringToWordVector{
   public static void main(String args[]) throws Exception{
      //load dataset
      DataSource source = new DataSource("E:\\source\\location\\of\\dataset\\datastringed.arff");
      Instances dataset = source.getDataSet();

      //use a simple filter to Str2Wrod a certain attribute
      //set up options to Str2Word 2nd attribute
      String[] options = new String[10];
      //create list of options
      options[0] = "-R"; options[1] = "2";
      options[6] = "-C";
      options[13] = "-tokenizer"; options[14] = "-max"; options[15] = "3"; options[16] = "-min";
options[17] = "3"; options[18] = "-delimiters"; options[19] = "\\\"
\\\\r\\\\n\\\\t.,;:\\\\\\'\\\\\\\"()?!\\\"\"";
      //create a StringToWordVector object (this is the filter class)
      StringToWordVector Str_2_Wrd = new StringToWordVector();
      //set the filter options
      Str_2_Wrd.setOptions(options);
      //pass the dataset to the filter
      Str_2_Wrd.setInputFormat(dataset);
      //apply the filter
      Instances newData = Filter.useFilter(dataset, Str_2_Wrd);

      //now save the dataset to a new file as we learned before
      ArffSaver saver = new ArffSaver();
      saver.setInstances(newData);
      saver.setFile(new File("E:\\destination\\folder\\adress\\datastring2worded.arff"));
      saver.writeBatch();
   }
         private void setOptions(String[] options) {
                  // TODO Auto-generated method stub

         }

         private void setInputFormat(Instances dataset) {
                  // TODO Auto-generated method stub

         }
}
```

3. <u>To rearrange the attributes of the dataset so as to make the 'nature' attribute the class of the dataset using Reorder filter.</u>

```
package weka.api;

//import required classes
import weka.core.Instances;
import weka.core.converters.ArffSaver;
import java.io.File;
import weka.core.converters.ConverterUtils.DataSource;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.NominalToString;
import weka.filters.unsupervised.attribute.Reorder;

public class Reorder{
   public static void main(String args[]) throws Exception{
      //load dataset
      DataSource source = new DataSource("E:\\ source\\location\\of\\dataset
\\data2_3tokensarff.arff");
      Instances dataset = source.getDataSet();

      //use a simple filter to reOrd a certain attribute
      //set up options to reOrd 2nd attribute
      String[] opts = new String[]{ "-R", "1,3-last,2"};
      //create a Reorder object (this is the filter class)
      Reorder reOrd = new Reorder();
      //set the filter options
      reOrd.setOptions(opts);
      //pass the dataset to the filter
      reOrd.setInputFormat(dataset);
      //apply the filter
      Instances newData = Filter.useFilter(dataset, reOrd);

      //now save the dataset to a new file as we learned before
      ArffSaver saver = new ArffSaver();
      saver.setInstances(newData);
      saver.setFile(new File("E:\\ destination\\folder\\adress \\datareordered.arff"));
      saver.writeBatch();
   }

         private void setOptions(String[] opts) {
                  // TODO Auto-generated method stub

         }

         private void setInputFormat(Instances dataset) {
                  // TODO Auto-generated method stub

         }
}
```

1. DecisionStump

```
package weka.api;

import weka.classifiers.trees.DecisionStump;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import weka.classifiers.Evaluation;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;
import java.lang.Math;


public class decisionStumpNewtest
{
static String DecisionStumpModelPath = "E:\\ source\\location\\of\\model \\decisionstump.model";

public static void main(String args[]) throws Exception{

double percent = 90.0;
BufferedReader breader;
breader =new BufferedReader (new
FileReader("E:\\source\\location\\of\\dataset\\dataordered.arff"));
Instances inst = new Instances(breader);

inst.setClassIndex(inst.numAttributes() - 1); //set the last column as the class attribute
///PARAMETERS are set

DecisionStump deciStumptree = new DecisionStump();
int seed=1;
Random rnd = new Random(seed);
inst.randomize(rnd);

String[] options;

options = weka.core.Utils.splitOptions("");
deciStumptree.setOptions(options);

System.out.println("Performing " +percent+" %split evaluation");



int trainSize = (int) Math.round(inst.numInstances()*percent/100);

int testSize = inst.numInstances()-trainSize;


Instances train = new Instances (inst, 0, trainSize);
Instances test = new Instances (inst, trainSize,testSize);

deciStumptree.buildClassifier(train);
Evaluation eval = new Evaluation(train );
eval.evaluateModel(deciStumptree, test);


breader.close();


System.out.println(eval.toSummaryString("\nResults\n======\n", false));


System.out.println("training performance results of: " + deciStumptree.getClass().getSimpleName()
+ "\n———————————");
System.out.println(eval.toSummaryString("\nResults",true));
System.out.println("fmeasure: " +eval.fMeasure(1) + " Precision: " + eval.precision(1)+ " Recall: "+
eval.recall(1));
System.out.println(eval.toMatrixString());
System.out.println(eval.toClassDetailsString());
System.out.println("AUC = " +eval.areaUnderROC(1));
System.out.println("Training complete, please validate trained model");
// weka.core.SerializationHelper.write(J48ModelPath,J4tree);

}
}
```

2. <u>J48</u>

```
package weka.api;

import weka.classifiers.trees.J48;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import weka.classifiers.Evaluation;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;
import java.lang.Math;


public class J48Classifiertestnew
{
static String J48ModelPath = "E:\\ source\\location\\of\\model \\J48.model";

public static void main(String args[]) throws Exception{

double percent = 90.0;
BufferedReader breader;
breader =new BufferedReader (new FileReader("E:\\ source\\location\\of\\dataset\\
dataordered.arff"));
Instances inst = new Instances(breader);
inst.setClassIndex(inst.numAttributes() - 1); //set the last column as the class attribute
///PARAMETERS are set

J48 J4tree = new J48();
int seed=1;
Random rnd = new Random(seed);
inst.randomize(rnd);

String[] options;

options = weka.core.Utils.splitOptions("-C 0.25 -M 2");
J4tree.setOptions(options);

System.out.println("Performing " +percent +"%split evaluation");



int trainSize = (int) Math.round(inst.numInstances()*percent/100);

int testSize = inst.numInstances()-trainSize;


Instances train = new Instances (inst, 0, trainSize);
Instances test = new Instances (inst, trainSize,testSize);

J4tree.buildClassifier(train);

Evaluation eval = new Evaluation(train );
eval.evaluateModel(J4tree, test);


breader.close();


System.out.println(eval.toSummaryString("\nResults\n======\n", false));


System.out.println("training performance results of: " + J4tree.getClass().getSimpleName()
+ "\n————————————");
System.out.println(eval.toSummaryString("\nResults",true));
System.out.println("fmeasure: " +eval.fMeasure(1) + " Precision: " + eval.precision(1)+ " Recall: "+
eval.recall(1));
System.out.println(eval.toMatrixString());
System.out.println(eval.toClassDetailsString());
System.out.println("AUC = " +eval.areaUnderROC(1));
System.out.println("Training complete, please validate trained model");
// weka.core.SerializationHelper.write(J48ModelPath,J4tree);

}
}
```

3. <u>JRip</u>

```
package weka.api;

import weka.classifiers.rules.JRip;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import weka.classifiers.Evaluation;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;
import java.lang.Math;


public class jRIP
{
static String DecisionStumpModelPath = "E:\\source\\location\\of\\model\\jrip.model";

public static void main(String args[]) throws Exception{

double percent = 90.0;
BufferedReader breader;
breader =new BufferedReader (new FileReader("E:\\source\\location\\of\\dataset
\\dataordered.arff"));
Instances inst = new Instances(breader);

inst.setClassIndex(inst.numAttributes() - 1); //set the last column as the class attribute
///PARAMETERS are set

JRip jayRip = new JRip();
int seed=1;
Random rnd = new Random(seed);
inst.randomize(rnd);

String[] options;

options = weka.core.Utils.splitOptions("-F 3 -N 2.0 -O 2 -S 1");
jayRip.setOptions(options);

System.out.println("Performing " +percent+" %split evaluation");



int trainSize = (int) Math.round(inst.numInstances()*percent/100);

int testSize = inst.numInstances()-trainSize;


Instances train = new Instances (inst, 0, trainSize);
Instances test = new Instances (inst, trainSize,testSize);

jayRip.buildClassifier(train);
Evaluation eval = new Evaluation(train );
eval.evaluateModel(jayRip, test);


breader.close();


System.out.println(eval.toSummaryString("\nResults\n======\n", false));


System.out.println("training performance results of: " + jayRip.getClass().getSimpleName()
+ "\n---------------------------------");
System.out.println(eval.toSummaryString("\nResults",true));
System.out.println("fmeasure: " +eval.fMeasure(1) + " Precision: " + eval.precision(1)+ " Recall: "+
eval.recall(1));
System.out.println(eval.toMatrixString());
System.out.println(eval.toClassDetailsString());
System.out.println("AUC = " +eval.areaUnderROC(1));
System.out.println("Training complete, please validate trained model");
// weka.core.SerializationHelper.write(J48ModelPath,J4tree);

}
}
```

4. NaiveBayes

```
package weka.api;

import weka.classifiers.bayes.NaiveBayes;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import weka.classifiers.Evaluation;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;
import java.lang.Math;


public class naiveBayestest
{
static String DecisionStumpModelPath = "E:\\Nitin\\source\\location\\of\\model
\\naiveBayes.model";

public static void main(String args[]) throws Exception{

double percent = 90.0;
BufferedReader breader;
breader =new BufferedReader (new FileReader("E:\\source\\location\\of\\dataset
\\dataordered.arff"));
Instances inst = new Instances(breader);
inst.setClassIndex(inst.numAttributes() - 1); //set the last column as the class attribute
///PARAMETERS are set

NaiveBayes NBays = new NaiveBayes();
int seed=1;
Random rnd = new Random(seed);
inst.randomize(rnd);

String[] options;

options = weka.core.Utils.splitOptions("");
NBays.setOptions(options);

System.out.println("Performing " +percent+" %split evaluation");



int trainSize = (int) Math.round(inst.numInstances()*percent/100);

int testSize = inst.numInstances()-trainSize;


Instances train = new Instances (inst, 0, trainSize);
Instances test = new Instances (inst, trainSize,testSize);

NBays.buildClassifier(train);

Evaluation eval = new Evaluation(train );
eval.evaluateModel(NBays, test);


breader.close();


System.out.println(eval.toSummaryString("\nResults\n======\n", false));


System.out.println("training performance results of: " + NBays.getClass().getSimpleName()
+ "\n---------------------------------");
System.out.println(eval.toSummaryString("\nResults",true));
System.out.println("fmeasure: " +eval.fMeasure(1) + " Precision: " + eval.precision(1)+ " Recall: "+
eval.recall(1));
System.out.println(eval.toMatrixString());
System.out.println(eval.toClassDetailsString());
System.out.println("AUC = " +eval.areaUnderROC(1));
System.out.println("Training complete, please validate trained model");
// weka.core.SerializationHelper.write(J48ModelPath,J4tree);

}
}
```

5. OneR

```
package weka.api;

import weka.classifiers.rules.OneR;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import weka.classifiers.Evaluation;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;
import java.lang.Math;


public class oneRtest
{
static String DecisionStumpModelPath = "E:\\source\\location\\of\\model\\oneR.model";

public static void main(String args[]) throws Exception{

double percent = 90.0;
BufferedReader breader;
breader =new BufferedReader (new FileReader("E:\\lsource\\location\\of\\dataset
\\dataordered.arff"));
Instances inst = new Instances(breader);

inst.setClassIndex(inst.numAttributes() - 1); //set the last column as the class attribute
///PARAMETERS are set

OneR wunaaR = new OneR();
int seed=1;
Random rnd = new Random(seed);
inst.randomize(rnd);

String[] options;

options = weka.core.Utils.splitOptions("-B 6");
wunaaR.setOptions(options);

System.out.println("Performing " +percent+" %split evaluation");



int trainSize = (int) Math.round(inst.numInstances()*percent/100);

int testSize = inst.numInstances()-trainSize;


Instances train = new Instances (inst, 0, trainSize);
Instances test = new Instances (inst, trainSize,testSize);

wunaaR.buildClassifier(train);
Evaluation eval = new Evaluation(train );
eval.evaluateModel(wunaaR, test);


breader.close();


System.out.println(eval.toSummaryString("\nResults\n======\n", false));


System.out.println("training performance results of: " + wunaaR.getClass().getSimpleName()
+ "\n---------------------------------");
System.out.println(eval.toSummaryString("\nResults",true));
System.out.println("fmeasure: " +eval.fMeasure(1) + " Precision: " + eval.precision(1)+ " Recall: "+
eval.recall(1));
System.out.println(eval.toMatrixString());
System.out.println(eval.toClassDetailsString());
System.out.println("AUC = " +eval.areaUnderROC(1));
System.out.println("Training complete, please validate trained model");
// weka.core.SerializationHelper.write(J48ModelPath,J4tree);

}
}
```

6. PART

```
package weka.api;

import weka.classifiers.rules.PART;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import weka.classifiers.Evaluation;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;
import java.lang.Math;


public class paArtTest
{
static String DecisionStumpModelPath = "E:\\source\\location\\of\\model\\part.model";

public static void main(String args[]) throws Exception{

double percent = 90.0;
BufferedReader breader;
breader =new BufferedReader (new FileReader("E:\\source\\location\\of\\dataset
\\dataordered.arff"));
Instances inst = new Instances(breader);
inst.setClassIndex(inst.numAttributes() - 1); //set the last column as the class attribute
///PARAMETERS are set

PART par8 = new PART();
int seed=1;
Random rnd = new Random(seed);
inst.randomize(rnd);

String[] options;

options = weka.core.Utils.splitOptions("-M 2 -C 0.25 -Q 1");
par8.setOptions(options);

System.out.println("Performing " +percent+" %split evaluation");



int trainSize = (int) Math.round(inst.numInstances()*percent/100);

int testSize = inst.numInstances()-trainSize;


Instances train = new Instances (inst, 0, trainSize);
Instances test = new Instances (inst, trainSize,testSize);

par8.buildClassifier(train);

Evaluation eval = new Evaluation(train );
eval.evaluateModel(par8, test);


breader.close();


System.out.println(eval.toSummaryString("\nResults\n======\n", false));


System.out.println("training performance results of: " + par8.getClass().getSimpleName()
+ "\n———————————————");
System.out.println(eval.toSummaryString("\nResults",true));
System.out.println("fmeasure: " +eval.fMeasure(1) + " Precision: " + eval.precision(1)+ " Recall: "+
eval.recall(1));
System.out.println(eval.toMatrixString());
System.out.println(eval.toClassDetailsString());
System.out.println("AUC = " +eval.areaUnderROC(1));
System.out.println("Training complete, please validate trained model");
// weka.core.SerializationHelper.write(J48ModelPath,J4tree);

}
}
```

7. RandomForest

```
package weka.api;

import weka.classifiers.trees.RandomForest;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import weka.classifiers.Evaluation;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;
import java.lang.Math;


public class randomforTest
{
static String RandomForestModelPath = "E:\\source\\location\\of\\model\\randomForest.model";

public static void main(String args[]) throws Exception{

double percent = 90.0;
BufferedReader breader;
breader =new BufferedReader (new FileReader("E:\\source\\location\\of\\dataset
\\dataordered.arff"));
Instances inst = new Instances(breader);

inst.setClassIndex(inst.numAttributes() - 1); //set the last column as the class attribute
///PARAMETERS are set

RandomForest randomfortree = new RandomForest();
int seed=1;
Random rnd = new Random(seed);
inst.randomize(rnd);

String[] options;

options = weka.core.Utils.splitOptions("-P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1");
randomfortree.setOptions(options);

System.out.println("Performing " +percent+" %split evaluation");



int trainSize = (int) Math.round(inst.numInstances()*percent/100);

int testSize = inst.numInstances()-trainSize;


Instances train = new Instances (inst, 0, trainSize);
Instances test = new Instances (inst, trainSize,testSize);

randomfortree.buildClassifier(train);
Evaluation eval = new Evaluation(train );
eval.evaluateModel(randomfortree, test);


breader.close();


System.out.println(eval.toSummaryString("\nResults\n======\n", false));


System.out.println("training performance results of: " + randomfortree.getClass().getSimpleName()
+ "\n———————————");
System.out.println(eval.toSummaryString("\nResults",true));
System.out.println("fmeasure: " +eval.fMeasure(1) + " Precision: " + eval.precision(1)+ " Recall: "+
eval.recall(1));
System.out.println(eval.toMatrixString());
System.out.println(eval.toClassDetailsString());
System.out.println("AUC = " +eval.areaUnderROC(1));
System.out.println("Training complete, please validate trained model");
// weka.core.SerializationHelper.write(J48ModelPath,J4tree);

}
}
```

8. <u>SMO</u>

```
package weka.api;

import weka.classifiers.functions.SMO;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import weka.classifiers.Evaluation;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;
import java.lang.Math;


public class SMoTest
{
static String DecisionStumpModelPath = "E:\\source\\location\\of\\model\\SMO.model";

public static void main(String args[]) throws Exception{

double percent = 90.0;
BufferedReader breader;
breader =new BufferedReader (new FileReader("E:\\source\\location\\of\\dataset
\\dataordered.arff"));
Instances inst = new Instances(breader);
inst.setClassIndex(inst.numAttributes() - 1); //set the last column as the class attribute
///PARAMETERS are set

SMO esemOH = new SMO();
int seed=1;
Random rnd = new Random(seed);
inst.randomize(rnd);

String[] options;

options = weka.core.Utils.splitOptions("-C 1.0 -L 0.001 -P 1.0E-12 -N 0 -V -1 -W 1 -K
\"weka.classifiers.functions.supportVector.PolyKernel -E 1.0 -C 250007\" -calibrator
\"weka.classifiers.functions.Logistic -R 1.0E-8 -M -1 -num-decimal-places 4\"");
esemOH.setOptions(options);

System.out.println("Performing " +percent+" %split evaluation");



int trainSize = (int) Math.round(inst.numInstances()*percent/100);

int testSize = inst.numInstances()-trainSize;


Instances train = new Instances (inst, 0, trainSize);
Instances test = new Instances (inst, trainSize,testSize);

esemOH.buildClassifier(train);

Evaluation eval = new Evaluation(train );
eval.evaluateModel(esemOH, test);


breader.close();


System.out.println(eval.toSummaryString("\nResults\n======\n", false));


System.out.println("training performance results of: " + esemOH.getClass().getSimpleName()
+ "\n——————————————");
System.out.println(eval.toSummaryString("\nResults",true));
System.out.println("fmeasure: " +eval.fMeasure(1) + " Precision: " + eval.precision(1)+ " Recall: "+
eval.recall(1));
System.out.println(eval.toMatrixString());
System.out.println(eval.toClassDetailsString());
System.out.println("AUC = " +eval.areaUnderROC(1));
System.out.println("Training complete, please validate trained model");
// weka.core.SerializationHelper.write(J48ModelPath,J4tree);

}
}
```

9. ZeroR

```
package weka.api;

import weka.classifiers.rules.ZeroR;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import weka.classifiers.Evaluation;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;
import java.lang.Math;


public class zer0Rtest
{
static String DecisionStumpModelPath = "E:\\source\\location\\of\\model\\zeroR.model";

public static void main(String args[]) throws Exception{

double percent = 90.0;
BufferedReader breader;
breader =new BufferedReader (new FileReader("E:\\source\\location\\of\\dataset
\\dataordered.arff"));
Instances inst = new Instances(breader);

inst.setClassIndex(inst.numAttributes() - 1); //set the last column as the class attribute
///PARAMETERS are set

ZeroR zer0ARR = new ZeroR();
int seed=1;
Random rnd = new Random(seed);
inst.randomize(rnd);

String[] options;

options = weka.core.Utils.splitOptions("");
zer0ARR.setOptions(options);

System.out.println("Performing " +percent+" %split evaluation");



int trainSize = (int) Math.round(inst.numInstances()*percent/100);

int testSize = inst.numInstances()-trainSize;


Instances train = new Instances (inst, 0, trainSize);
Instances test = new Instances (inst, trainSize,testSize);

zer0ARR.buildClassifier(train);
Evaluation eval = new Evaluation(train );
eval.evaluateModel(zer0ARR, test);


breader.close();


System.out.println(eval.toSummaryString("\nResults\n======\n", false));


System.out.println("training performance results of: " + zer0ARR.getClass().getSimpleName()
+ "\n―――――――――――――――");
System.out.println(eval.toSummaryString("\nResults",true));
System.out.println("fmeasure: " +eval.fMeasure(1) + " Precision: " + eval.precision(1)+ " Recall: "+
eval.recall(1));
System.out.println(eval.toMatrixString());
System.out.println(eval.toClassDetailsString());
System.out.println("AUC = " +eval.areaUnderROC(1));
System.out.println("Training complete, please validate trained model");
// weka.core.SerializationHelper.write(J48ModelPath,J4tree);

}
}
```

CONCLUSION

In this paper, we proposed a novel method of API call sequence analysis. We extract API call sequence patterns from malware in different categories, focusing on common malware functions. Experimentally, our system showed promising detection results with high accuracy and extremely low error rate.

In the end, we found that the RandomForest classifier is the surest method for classification of unknown executables based on the current dataset used by us in the project.

Many malware detection systems rely on the signature of a malware’s static information, such as file size, process, and its artefacts. Therefore, they fail to detect new unknown malware until the signature has been updated. In contrast to the signature of the malware’s static information, our signature database of dynamic information of critical API call sequence patterns allows us to generalize malware behaviour and facilitates the effective detection of new unknown malware. Our method can be applied to both of the traditional PCs and new smart devices if the devices can extract and send the extracted API call sequence information to the analysis module outside. Because API call sequence is well abstracted behavioural data that can be extracted from most of the device, therefore, our method can detect the attack by analysing the big data that are extracted from the ubiquitous devices such as sensors, smart devices, PCs, and servers.

REFERENCES

1. WEKA 3.8 - HTTP://WWW.CS.WAIKATO.AC.NZ/ML/WEKA/DOWNLOADING.HTML
2. DATASET USED - HTTP://OCSLAB.HKSECURITY.NET/APIMDS-DATASET
3. WEKA ANALYSIS TUTORIAL - HTTPS://WWW.YOUTUBE.COM/PLAYLIST?LIST=PLZVF1NAQI9VMC96TBVOPMKXTOSMBMHJN7
4. DATA MINING PRACTICAL MACHINE LEARNING TOOLS AND TECHNIQUES 3RD EDITION - HTTPS://GITHUB.COM/WQ19901103WQ/TODO/BLOB/MASTER/DATA%20MINING%20PRACTICAL%20MACHINE%20LEARNING%20TOOLS%20AND%20TECHNIQUES%203RD%20EDITION.PDF
5. WEKA API DOCUMENTATION - HTTP://WEKA.SOURCEFORGE.NET/DOC.STABLE/