Chapter 1

Applying Natural Language Processing Techniques in Ensemble Feature Extraction Models to Detect Android Malware through Static Analysis

Nakul Aggarwal[a]

[a]Computer Science and Engineering, Indian Institute of Technology Kharagpur

In this chapter, we present a novel malware detection model for Android applications, using natural language processing and deep learning to extract features from the static analysis reports of the applications. We give the motivation behind the proposed model and its relevance in the modern technology. The details of the model and the training dataset to be used in its implementation are also discussed.

Keywords: Android Malware Detection, Static Analysis, NLP, Deep Learning

The chapter is organized as follows: Section I for Background, Section II for Introduction, Section III for Literature Review, Section IV for Framework and Section V for Dataset.

1 Background

Malware investigation is an important and time consuming task for security investigators. The daily volume of malware raises the automation necessity of detection and threat attribution tasks. The diversity of platforms and architectures makes the malware investigation more challenging. The investigator has to deal with variety of malware scenarios from Win32 to Android. The diversity of malware induces the need for portable tools, methods, and techniques in the security investigator's toolbox for malware detection. These tools should not only

be effective but also easily maintainable. Re-usability and re-innovation of the existing malware detectors can help the security investigators to keep up with the continuously evolving and expanding malware families without developing anything from the scratch.

Static and dynamic analyses are sources for security features, which the security investigator uses to decide the maliciousness of a given application. Manual inspection of these features is a tedious task and could be automated using machine learning techniques. For this reason, the majority of the state-of-the-art malware detection solutions use machine learning techniques. Developers must carefully choose between the two analysis techniques depending upon the requirements of the model and the kind of applications the model is expected to deal with.

Static analysis is performed in a non-run-time environment. It basically involves looking at the application from inside out without executing the program, but rather by examining the source code, byte code or application binaries for signs of security vulnerabilities. In the static analysis, the application data and control paths are modeled and then analyzed for security weaknesses. Dynamic analysis adopts the opposite approach and is executed while a program is in operation. It looks at the application by examining it in its running state and trying to manipulate it in order to discover security vulnerabilities. The behaviour of the application in dynamic analysis is studied by emulating it on a sandbox, with a *monkeyrunner* giving random inputs to the application.

Static analysis, with its white-box visibility, is certainly the more thorough approach and may also prove more time and cost efficient with the ability to detect malware. Static analysis can also unearth some properties that would not emerge in a dynamic test. Dynamic analysis, on the other hand, is capable of exposing a subtle vulnerability too complicated for static analysis alone to reveal, especially in cases of code encryption and obfuscation. A dynamic test, however, will only find defects in the part of the code that is actually executed. The developer must weigh up these considerations with the complexities of their own situation in mind while choosing the best analysis technique. Application type, time, and availability of resources are some of the primary concerns.

2 Introduction

Natural Language Processing is widely used to develop an ability in systems for understanding how a language is written. Now it need not necessarily be a language meaningful to humans. The behaviour of any model can be condensed

into a bunch of mathematical equations, attributes and key-value pairs that can be interpreted as the *language* of that model. Similarly, the log files generated by doing analysis (static or dynamic) on an Android application can be treated as a language in its own. If we are somehow able to reduce a bunch of labelled analysis reports into a *language of Android applications* and then teach this language to an automated intelligence-driven model, we would be able to classify an unseen application as benign or malicious just by extracting its analysis report and interpreting it using the knowledge of the learned language. The core argument here is that malicious applications have distinguishable behaviours from benign applications and this difference is translated into words in the analysis reports. These underlying differences further seep into the language learned by the model. We will be using static analysis reports for feature engineering due to limited resources.

The key idea of the proposed model is to develop a vast feature extraction engine made up of multiple feature engineering models, rather than using a single extractor to extract and aggregate the features. Each component feature extractor reduces static analysis reports of applications into a sequence of words and encodes these reports into robust vector representations that effectively capture the permissions, intents, method tags, antagonists, sensitive API calls and third party interventions mentioned in the report. A collection of multiple feature engineering models can produce better quality encodings of the reports, rather than using a single model.

The reason behind formulating multiple feature extractors is the resultant portability of the model. There are many ways, of varying efficacies, for interpreting the analysis reports of applications. Naturally, in practice, one model may or may not be better than the other. For example, a statistical TF-IDF technique can be effective in deducing the important words across multiple documents (or paragraphs), but it cannot capture the numerous contexts in which that word occurs. Similarly, an RNN or CNN have very different architectures and serve different functionalities while processing any natural language text. Traditionally, we think that a CNN is specialized for processing a grid of values such as an image, and an RNN is specialized for processing a sequence of values. But more recently CNNs have also been started to apply in Natural Language Processing problems, like is proven by *Kim et. al*, in their work, *Convolutional Neural Networks for Sentence Classification.*

Ensemble feature engineering techniques help in bringing together the best from various feature extraction models, among which it might be difficult to choose one intuitively. Interpolating several promising feature extractors together

can result in an overall better feature extraction mechanism that balances out the limitations of every individual constituent feature extractor. Though many papers have used ensemble learning in the classification part of the malware detection pipeline by using ensemble machine learning models, no papers were found that use a similar methodology in feature engineering.

3 Literature Review

Karbab et. al (2019) in their *MalDy* tool modelled the dynamic analysis reports into a sequence of words, along with advanced natural language processing and machine learning techniques for automatic engineering of relevant security features to detect and attribute malware without the investigator intervention. More precisely, they proposed to use *bag-of-words* (BoW) NLP model to formulate the behavioral reports and built ML ensembles on top of BoW features. In the automatic security feature engineering process, MalDy uses classical N-Gram Analysis in conjunction with statistical NLP techniques like feature hashing and term frequency-inverse document frequency (TF-IDF). However, the bag-of-words has many disadvantages. The word order is lost, and thus different sentences can have exactly the same representation, as long as the same words are used. Even though bag-of-n-grams considers the word order in short context, it suffers from data sparsity and high dimensionality. Bag-of-words and bag-of-n-grams have very little sense about the semantics of the words or more formally the distances between the feature vectors to which the documents/reports will be mapped in the embedding space.

The other technique used here, i.e, feature hashing also has its own limitations. It has the effect of reducing memory requirements and making it highly suitable for use in streaming applications where a pre-defined vocabulary might not be available or practical. But when using feature hashing, the number of unique words across the corpus is unknown, so the dimensionality of feature vectors must be set in advance. This length is entirely arbitrary but should be sufficiently large to support all required words from the language being used. Hence this approach will create vectors of extremely high dimensionality and so without sparse matrix formats, this approach is not feasible because of the memory requirements necessary to store all the elements in dense matrix format. One potential downside of using this hashing trick is that multiple collocations may translate to the same index in the vector space (hash collisions) resulting in a loss of precision. Another disadvantage of feature hashing is that, once vectors are created, it is difficult to determine which frequency values relate to which terms.

Ma et. al (2021) in their *SVM-L* model proposed for anomaly detection in network traffic. In particular, raw URLs are treated as natural language, and then transformed into mathematical vectors via statistical laws and natural language processing techniques. These vectors are used as the training data for the traffic classifier, a kernel Support Vector Machine. Based on the idea of the dual formulation of kernel SVM and Linear Discriminant Analysis, they further propose an optimization model to adjust the hyper-parameters of the classifier. This paper focuses on the analysis of URLs of HTTP requests to identify abnormal network traffic. The original URLs are transformed into weight vectors based on statistical laws, and are further converted into fixed-length feature vectors by the k-gram technique from natural language processing. SVM-L is quite similar to MalDy in terms of the underlying idea behind the feature engineering.

The authors in *Yang, Zuo, Cui* (2019) construct a hand-crafted vocabulary of keywords in malicious URLs, which is used for word embedding of URLs. The obtained embedding vectors are fed into a convolutional gated-recurrent-unit neural network. In *Kolari et al.* (2016), the bag-of-words model is used for data transformation. A dictionary of words in URLs is constructed, and each word in the dictionary becomes a feature. If a word appears in the URL, the corresponding value of the feature takes 1, and 0 otherwise. *Blum et al.* (2010) apply the bi-gram technique to enhance the lexical features extracted by the bag-of-words method. In addition, the appearances of all two-word combinations in the URLs are considered as extra features. The obtained lexical features are then forwarded into machine learning models for anomaly detection.

A major difficulty of Generative Adversarial Networks (GANs) is to generate fake samples that represent the context. *Mimura et. al* (2020) have used NLP (*Paragraph Vector*) to generate fake text vectors to enhance small training samples. Paragraph Vector is a model to convert text into vectors, which represents the context and numerical distance. These features allow to directly vary each element of the vectors. This method adds random noise to the vectors to generate fake text vectors which represent the context. Their work applies this technique to detect new malicious VBA (Visual Basic for Applications) macros to address the practical problem. This generic technique could be used for not only malware detection, but also any imbalanced and contextual data. To simulate small training samples, the malicious samples are reduced, and fake samples from the reduced ones are generated.

To solve the inefficiency problem of manual feature engineering that is not only very time consuming but also requires experience, *Zhang et. al* (2021) proposed an automatic framework for Android malware detection based on text

classification method, called *TC-Droid*. This model feeds on the text sequence of analysis reports generated by *AndroPyTool*, and uses a convolutional neural network (CNN) to explore significant information under original report text, instead of manual feature engineering. The proposed approach uses a *TextCNN* model for text classification.

4 Framework

In this section we describe a novel malware detection framework using ensemble feature engineering. First, we describe the preliminary information on static attributes of an application. Next, we give a mathematical overview of the proposed model.

4.1 Static Attributes of Android Applications

The static analysis samples of Android applications include the following important attributes, among several others.

• **Permissions** - This is one of the most important security mechanisms in an Android system. It provides authority to an application to get access to sensitive resources or execute some high-risk operations.

• **Services** - This is a list of different service points used by the app. This list of service names is an important feature set, as it may help in identifying well-known components of different malware.

• **Intents** - Similar to services, intents are also a component of the Android system. Besides, they are a key communication element as they enable an app to seek permission from the OS to execute other application components or processes. For example, an intent can require the execution of actions such as enabling camera or accessing photo album, etc.

• **Receivers** - This a global listener, which belongs to one of the four major components of Android. The broadcast receiver is used to receive broadcast intents asynchronously.

• **Sensitive APIs** - These are the critical interfaces to which the app has an access. These might include APIs related to File manager, Telephone manager, Activity manager, WiFi manager, Package Manager and Location manager. If an application has access to these APIs, it can pull the sim serial number, current location, subscriber identity and IP address of the device.

```
{"intent_actions": ["android.intent.action.MAIN", "android.intent.action.PACKAGE_ADDED", "android.intent.action.PACKAGE_REMOVED", "com.elm.downloadManager",
"android.net.conn.CONNECTIVITY_CHANGE", "com.zdt.action.ALARM_ACTION", "android.intent.action.USER_PRESENT", "android.intent.action.CREATE_SHORTCUT",
"android.intent.action.PACKAGE_ADDED", "android.net.conn.CONNECTIVITY_CHANGE", "android.intent.action.USER_PRESENT"], "num_permissions": 18, "intent_consts":
["'android.intent.action.VIEW'", "'android.intent.action.SEND'", "'android.intent.extra.TEXT'", "'android.intent.extra.SUBJECT'", "'android.intent.action.SEND'",
"'android.intent.extra.TEXT'", "'android.intent.action.VIEW'", "'android.intent.action.CREATE_SHORTCUT'", "'android.intent.action.VIEW'",
"'android.intent.action.VIEW'", "'android.intent.action.MAIN'", "'android.intent.category.LAUNCHER'", "'android.intent.action.GET_CONTENT'",
"'android.intent.action.PACKAGE_ADDED'", "'android.intent.action.USER_PRESENT'", "'android.intent.action.VIEW'", "'android.intent.action.VIEW'",
"'android.intent.action.VIEW'", "'android.intent.action.SEND'", "'android.intent.extra.SUBJECT'", "'android.intent.extra.STREAM'", "'android.intent.extra.TEXT'",
"'android.intent.extra.shortcut.NAME'", "'android.intent.extra.shortcut.INTENT'", "'android.intent.extra.shortcut.ICON'", "'android.intent.action.MAIN'",
"'android.intent.category.LAUNCHER'", "'android.intent.action.VIEW'", "'android.intent.extra.videoQuality'", "'android.intent.action.MAIN'",
"'android.intent.category.LAUNCHER'"], "num_intent_const_android_intent": 18, "used_permissions": {"android.permission.ACCESS_FINE_LOCATION": 2,
"android.permission.VIBRATE": 6, "android.permission.READ_PHONE_STATE": 4, "android.permission.ACCESS_WIFI_STATE": 2, "android.permission.ACCESS_COARSE_LOCATION": 2,
"android.permission.WAKE_LOCK": 4, "android.permission.ACCESS_NETWORK_STATE": 10, "android.permission.INTERNET": 4}, "file": {"DEX": 1, "Java": 1, "ASCII": 2, "ELF":
1, "JPEG": 2, "data": 6, "PNG": 92}, "method_tags": {"UTIL": 48, "WIDGET": 58, "DATABASE": 19, "PROVIDER": 7, "DALVIK_SYSTEM": 2, "TEXT": 21, "APP": 84,
"JAVA_REFLECTION": 7, "LOCATION": 4, "CONTENT": 210, "TELEPHONY": 7, "PREFERENCE": 2, "GRAPHICS": 41, "WEBKIT": 13, "ANDROID": 448, "OS": 78, "NET": 25, "VIEW": 32},
"num_intent_action_android_intent": 7, "num_files": 105, "num_intent_const_other": 13, "num_libraries": 0, "num_intent_action_other": 2, "e_sh_flags": {"A": 7, "AX":
2, "MS": 1, "AL": 1, "WA": 5}, "version_code": "308", "e_phnum": 5, "e_shentsize": 40, "ascii_obfuscation": 0, "symbols_shared_libraries": ["memcpy", "malloc"],
"num_receivers": 2, "e_phentsize": 32, "num_providers": 0, "num_third_part_permissions": 0, "num_intent_action_android_net": 2, "main_activity":
"hu.tonuzaba.android.CaricatureCreatorActivity", "receivers": ["com.elm.LMR", "com.kvh2.lj.p.LReceiver"], "num_services": 2, "is_valid_APK": true, "services":
["com.elm.LMR", "com.kvh2.lj.p.LReceiver"], "e_shstrndx": 18, "e_ehsize": 52, "antagonist": 0, "permissions": ["android.permission.INTERNET",
"android.permission.WAKE_LOCK", "android.permission.READ_PHONE_STATE", "android.permission.ACCESS_NETWORK_STATE", "android.permission.WRITE_EXTERNAL_STORAGE",
"android.permission.ACCESS_WIFI_STATE", "android.permission.GET_TASKS", "android.permission.SYSTEM_ALERT_WINDOW", "android.permission.CHANGE_WIFI_STATE",
"com.android.launcher.permission.INSTALL_SHORTCUT", "android.permission.WRITE_EXTERNAL_STORAGE", "android.permission.ACCESS_COARSE_LOCATION",
"android.permission.VIBRATE", "android.permission.RESTART_PACKAGES", "android.permission.WRITE_EXTERNAL_STORAGE", "android.permission.INTERNET",
"android.permission.ACCESS_NETWORK_STATE", "android.permission.ACCESS_COARSE_LOCATION"], "reflection": 4, "package": "funny.photo", "num_intent_actions": 11,
"num_activities": 5, "sensitive_API": {"TelephonyManager:getNetworkOperator": 2, "Process:killProcess": 2, "PackageManager:getInstalledPackages": 5,
"TelephonyManager:getSimSerialNumber": 0, "ActivityManager:getRunningServices": 0, "TelephonyManager:getSubscriberId": 1, "File:mkdir": 10, "Intent:setDataAndType":
3, "File:ListFiles": 0, "LocationManager:getLastKnownLocation": 2, "SmsManager:sendTextMessage": 0, "WifiManager:getConnectionInfo": 2, "Context:getFilesDir": 7,
"File:exists": 29, "ContentResolver:insert": 2, "URL:openConnection": 2, "HttpURLConnection:connect": 1, "URLConnection:getInputStream": 1,
"Context:getApplicationInfo": 4, "File:delete": 12, "WifiManager:getIpAddress": 0, "TelephonyManager:getCellLocation": 0, "TelephonyManager:getLine1Number": 0,
"WifiManager:isWifiEnabled": 0, "TelephonyManager:getDeviceId": 3, "ConnectivityManager:getActiveNetworkInfo": 8, "ActivityManager:getMemoryInfo": 0,
"ContentResolver:delete": 0, "Intent:addFlags": 4, "LocationManager:requestLocationUpdate": 0, "ContentResolver:query": 0, "Process:myPid": 2,
"ActivityManager:restartPackage": 0, "Intent:setFlags": 4, "Context:openFileOuput": 0}, "dynamic_code": 2, "emulator": 12, "native_code": 4, "e_shnum": 19}
```


Figure 1: Static analysis report of an Android app with Adware malware


```
{"intent_actions": ["android.intent.action.MAIN", "android.net.conn.CONNECTIVITY_CHANGE", "kr.go.assembly.nacast.NotifyBroadcast"], "num_permissions": 9,
"intent_consts": ["'android.intent.action.MEDIA_BUTTON'", "'android.intent.extra.KEY_EVENT'", "'android.intent.action.VIEW'", "'android.intent.action.VIEW'",
"'android.intent.action.VIEW'", "'android.intent.action.PICK'", "'android.intent.action.PICK'", "'android.intent.action.GET_CONTENT'",
"'android.intent.action.PACKAGE_ADDED'", "'android.intent.action.PACKAGE_REMOVED'", "'android.intent.action.PACKAGE_CHANGED'",
"'android.intent.action.PACKAGE_INSTALL'", "'android.intent.action.VIEW'", "'android.intent.action.PACKAGE_ADDED'", "'android.intent.action.PACKAGE_CHANGED'",
"'android.intent.action.PACKAGE_INSTALL'", "'android.intent.action.PACKAGE_REMOVED'", "'android.intent.action.MAIN'", "'android.intent.category.LAUNCHER'",
"'android.intent.action.VIEW'", "'android.intent.action.BATTERY_CHANGED'", "'android.intent.action.PICK'", "'android.intent.action.BATTERY_CHANGED'",
"com.google.android.c2dm.intent.REGISTRATION'", "'com.google.android.c2dm.intent.RECEIVE'", "'android.intent.action.HEADSET_PLUG'",
"'android.intent.action.HEADSET_PLUG'", "'android.intent.action.MEDIA_BUTTON'"], "num_intent_const_android_intent": 24, "used_permissions":
{"android.permission.ACCESS_FINE_LOCATION": 7, "android.permission.VIBRATE": 7, "android.permission.READ_PHONE_STATE": 3, "android.permission.CAMERA": 2,
"android.permission.ACCESS_COARSE_LOCATION": 7, "android.permission.WAKE_LOCK": 20, "android.permission.WRITE_SETTINGS": 1, "android.permission.ACCESS_WIFI_STATE":
2, "android.permission.INTERNET": 20, "android.permission.ACCESS_NETWORK_STATE": 14, "android.permission.RECORD_AUDIO": 1, "android.permission.READ_CONTACTS": 1,
"android.permission.WRITE_CONTACTS": 1}, "file": {"DEX": 1, "C": 1, "data": 59, "ELF": 13, "ASCII": 5, "PNG": 121}, "method_tags": {"TEXT": 8, "HARDWARE": 19,
"WEBKIT": 16, "NET": 13, "OPENGL": 20, "SPEECH": 4, "LOCATION": 8, "WIDGET": 313, "MEDIA": 37, "JAVA_REFLECTION": 46, "CONTENT": 217, "PREFERENCE": 4, "GRAPHICS":
156, "DEBUG": 1, "ANDROID": 1071, "UTIL": 112, "DATABASE": 31, "APP": 102, "TELEPHONY": 5, "PROVIDER": 4, "OS": 168, "VIEW": 179},
"num_intent_action_android_intent": 1, "num_files": 200, "num_intent_const_other": 4, "num_libraries": 0, "num_intent_action_other": 1, "e_sh_flags": {"A": 7, "AX":
2, "MS": 1, "AL": 1, "WA": 7}, "version_code": "23", "e_phnum": 6, "e_shentsize": 40, "ascii_obfuscation": 0, "symbols_shared_libraries": ["fork", "malloc",
"malloc", "malloc", "memcpy", "ioctl", "malloc", "memcpy"], "num_receivers": 1, "e_phentsize": 32, "num_providers": 0, "num_third_part_permissions": 0,
"num_intent_action_android_net": 1, "main_activity": "kr.go.assembly.nacast.SplashView", "receivers": ["kr.go.assembly.nacast.NotifyBroadcast"], "num_services": 0,
"is_valid_APK": true, "e_shstrndx": 20, "e_ehsize": 52, "antagonist": 0, "permissions": ["android.permission.INTERNET", "android.permission.VIBRATE",
"android.permission.WAKE_LOCK", "android.permission.GET_TASKS", "android.permission.KILL_BACKGROUND_PROCESSES", "android.permission.WRITE_EXTERNAL_STORAGE",
"android.permission.ACCESS_NETWORK_STATE", "android.permission.ACCESS_WIFI_STATE", "android.permission.CHANGE_WIFI_STATE"], "reflection": 15, "package":
"kr.go.assembly.nacast", "num_intent_actions": 3, "num_activities": 38, "sensitive_API": {"TelephonyManager:getNetworkOperator": 0, "Process:killProcess": 0,
"PackageManager:getInstalledPackages": 1, "TelephonyManager:getSimSerialNumber": 0, "ActivityManager:getRunningServices": 0, "TelephonyManager:getSubscriberId": 0,
"File:mkdir": 4, "Intent:setDataAndType": 0, "File:ListFiles": 0, "LocationManager:getLastKnownLocation": 2, "SmsManager:sendTextMessage": 0,
"WifiManager:getConnectionInfo": 2, "Context:getFilesDir": 0, "File:exists": 15, "ContentResolver:insert": 1, "URL:openConnection": 13, "HttpURLConnection:connect":
27, "URLConnection:getInputStream": 9, "Context:getApplicationInfo": 1, "File:delete": 8, "WifiManager:getIpAddress": 0, "TelephonyManager:getCellLocation": 0,
"TelephonyManager:getLine1Number": 1, "WifiManager:isWifiEnabled": 0, "TelephonyManager:getDeviceId": 2, "ConnectivityManager:getActiveNetworkInfo": 2,
"ActivityManager:getMemoryInfo": 0, "ContentResolver:delete": 1, "Intent:addFlags": 15, "LocationManager:requestLocationUpdate": 2, "ContentResolver:query": 2,
"Process:myPid": 0, "ActivityManager:restartPackage": 0, "Intent:setFlags": 131, "Context:openFileOuput": 0}, "dynamic_code": 0, "emulator": 5, "native_code": 448,
"e_shnum": 21}
```


Figure 2: Static analysis report of a benign Android app

Snapshot of a static analysis report of an application with *Adware* malware is shown in *fig. 1.* Snapshot of a static analysis report of a benign application is shown in *fig. 2.* (Source : *CICMalDroid2020, UNB*)

4.2 Feature Extraction Engine

The *feature extraction engine* is the most important element of the entire model. It consists of several independent feature extractors, each one of which has a unique architecture. The feature extraction engine feeds training data to each one of its component extractors. The features of the applications in the training

dataset learned by these individual components are interpolated (aggregated) together by the engine to generate the final feature vectors of the training samples, on which the machine learning classifier will be trained.

The interpolation schemes and the related mathematical formulations are discussed in the following section(s).

4.3 Interpolation of Feature Vectors

The idea of feature interpolation is inspired from the interpolation of language models which is a very effective technique used in natural language processing to reduce the perplexity of the resultant model. It is observed that the performance of a pure n-gram language model can be improved by mixing it with a naive uniform language model. Here mixing basically means taking a weighted combination of the probability distributions of the respective pure models. And not only that, even lower perplexities can be obtained if multiple n-gram models, say unigram, bigram and trigram, are interpolated together.

Let ***FEE*** be a feature extraction engine that consists of ***FEC***, a set of individual feature extraction components and ***FI***, a feature interpolator (aggregator). Let $TD = \{A_1, A_2, A_3, ..., A_i, ..., A_n\}$ be the training dataset where A_i is the static analysis report of the i^{th} application. Note that this dataset is labelled, i.e., for every analysis report it is known if the corresponding application is benign or malicious. Let $FEC = \{FE_1, FE_2, FE_3, ..., FE_j, ..., FE_m\}$.

Each component FE_j in FEC is fed the training dataset TD. The component reduces the analysis reports into a sequence of words and learns feature vectors for the encoded sequences using a *cross entropy loss* function for the binary classification. Every component in FEC is trained independently on the same training dataset. Let the feature vector of the i^{th} training sample learned by the j^{th} component be FV_{ji}, which has a dimension of k. Therefore, $FV_{ji} = \{f_{ji_1}, f_{ji_2}, f_{ji_3}, ..., f_{ji_t}, ..., f_{ji_k}\}^T$.

Once all the feature extraction components are trained and the feature vectors of the training samples learned by each of them are saved, FEE transfers the saved files to FI, the feature interpolator. Though interpolation of language models is usually linear, here a variety of interpolation schemes can be applied using some activation functions. However, we would focus only on linear interpolation. Study and implementation of non-linear interpolation of feature extraction components is left as a future work.

For linear interpolation, FI can be parameterized by a single $1 \times m$ column matrix, called IM. Note that FI here is not implemented as a trainable neural

network. The interpolation weights (entries in IM) forming a probability distribution over the m component extractors are chosen manually. The goal is to study the variation in the performance of the overall engine by tweaking the weight distributions. One hot distribution will be the trivial case of no interpolation. Similarly uniform, normal and binomial distributions are other possibilities. Various distributions can be analysed to check if interpolation of extraction models really works as well as the interpolation of language models. FI can be very easily converted into a shallow neural network with only one dense layer. It can then be trained on the feature vectors learned by FEC to obtain an optimum distribution of weights over the component feature extractors. This task is left as a future work.

$$\text{Let } EnFeM(i) = \begin{bmatrix} f_{1i_1} & f_{1i_2} & f_{1i_3} & . & . & . & f_{1i_k} \\ f_{2i_1} & f_{2i_2} & f_{2i_3} & . & . & . & f_{2i_k} \\ f_{3i_1} & f_{3i_2} & f_{3i_3} & . & . & . & f_{3i_k} \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ f_{mi_1} & f_{mi_2} & f_{mi_3} & . & . & . & f_{mi_k} \end{bmatrix}$$

be the ***En***semble ***Fe***ature ***M***atrix of the i^{th} training sample, where i ranges from 1 to n. It is basically a matrix of the feature vectors extracted by all the extraction components in FEC. Let $InFeV(i)$ be the ***In***terpolated ***Fe***ature ***V***ector of the i^{th} application.

$$\text{Then, } InFeV(i) = IM \times EnFeM(i)$$

4.4 Logistic Regression Classifier

Till now we have only engineered the features from the static analysis reports of applications. For a novel application, the model is not just expected to extract a vector of features representing that application, but is rather expected to classify the application as benign or malicious. A machine learning classifier has to be used to classify an application based on the features extracted by the feature extraction engine.

A *logistic regression classifier* is a machine learning model that in its basic form uses a logistic formulation to model a binary dependent variable, although many more complex extensions exist. It can be implemented as a regularized feed forward neural network with multiple hidden dense and dropout layers, with the output of the topmost layer being a two-dimensional vector that when passed

through a softmax layer gives a probability distribution over the two classes of the application - benign and malicious.

The classifier will be trained on the interpolated feature vectors of the training samples with *cross entropy loss* as the objective function. Therefore, the training dataset of the classifier is $TD' = \{InFeV(1), InFeV(2), ..., InFeV(i), ..., InFeV(n)\}$. Note that TD' is a labelled dataset having the same labels as the respective entries in TD.

4.5 Feature Extraction Components

In the previous sections, we gave a general-case mathematical description of our malware detection model, where the feature extraction engine is an ensemble of m extractors, where m can be any natural number in theory. For practical purposes it would not be feasible to have a very large number of feature extractors in our ensemble due to some obvious reasons. First of all, it would be computationally very expensive to train a large number of independent extractors. Secondly, a manual interpolation analysis of the pure extraction models would become increasingly tedious. Lastly, even after the model is trained and ready to be deployed, the classification of an application can become very slow if a large number of extractors are in a queue to extract features from its static attributes.

Therefore, it is important to choose a small (not too small) number of extractors in our ensemble for practical purposes. Another concern besides the size of the ensemble is the composition of the ensemble, i.e, what kind of models will go inside the ensemble. There are many famous models to extract features from text documents spread across various domains. Some of them are mentioned below (*Liang et. al* (2017)).

• **Filtering Method** : Filtration is quick and particularly suitable for large-scale text feature extraction. Filtration method of feature extraction mainly includes *word frequency*, *information gain*, and *mutual information method*.

• **Fusion Method** : Fusion needs integration of specific classifiers. *Weighted KNN* (K Nearest Neighbours) and *Center Vector Weighted Method* are covered by this domain.

• **Mapping Method** : Mapping has been widely applied to text classification and achieved good results. It is commonly used in *Latent Semantic Analysis*, *Least Squares Mapping* method and *Principal Component Analysis*.

• **Clustering Method** : Clustering takes the essential comparability of text features primarily to cluster text features into consideration. Then the center of each

class is utilized to replace the features of that class. *Chi-square Clustering* method and *Concept Indexing* are covered under this domain.

• **Statistical Methods** : These include some classical NLP or statistical techniques like *TF-IDF* and *feature hashing*.

• **Deep Learning Approach** : This approach covers neural networks of various architectures like *Recurrent*, *Convolutional* and *Feedforward*. *Autoencoders*, *Restricted Boltzmann Machines* and *Deep Belief Networks* can also be used.

• **Distributed Memory Models** : *PV-DM* (*Distributed Memory Model of Paragraph Vectors*) is an unsupervised algorithm that learns fixed-length feature representations from variable-length pieces of texts, such as sentences, paragraphs, and documents. It represents each document by a dense vector which is trained to predict words in the document (*Distributed Representations of Sentences and Documents*, *Mikolov et. al* (2014)).

Many promising candidates for the feature extraction components are available. We will have to choose only a few of them (3 to 4) due to the fixed size of the ensemble. The benefit of using an ensemble methodology in feature engineering is that one can very easily remove or add extraction components to the engine to improve the performance of the overall model. In this work, we design an ensemble made up of 3 components —— a ***CNN**-based extractor*, an ***RNN**-based extractor* and a ***PV-DM**-based extractor*.

5 Dataset

CICMalDroid (*Mahdavifar et. al* (2020)) is the training dataset used in this project. It has more than 17,341 Android samples from several sources including *VirusTotal* service, *Contagio* security blog, *AMD*, *MalDozer*, and other datasets. The samples were collected from December 2017 to December 2018. It spans between five distinct malware categories — *Adware*, *Banking* malware, *SMS* malware, *Riskware*, and benign.