appSHNE: The Application of Representation Learning for Semantic-Associated Heterogeneous Networks in Creating Android App Embedding Layers

Braden Riggs, Raya Kavosh, Alexander Friend

University of California, San Diego
La Jolla, 92037, USA

bdriggs@ucsd.edu, rakavosh@ucsd.edu, apfriend@ucsd.edu

Due to the open-source nature of the Android application development system, the threat of malicious software being uploaded to the Android marketplace and compromising the safety of millions of users is prevalent, and growing at an alarming rate. As time goes on, attackers develop new techniques for evading detection. Even the current state-of-the-art models often fall short when trying to represent complex relationships between different features of the code. In this paper, we will be exploring the implementation of both structured and unstructured app features in the creation of representative embedding layers. This multi-faceted approach, at scale, offers researchers greater insight into the inner working of the app with deployment that ranges from malware detection to similarity search and link prediction.

Keywords: Malware Detection; Representation Learning; Semantic-Associated Heterogeneous Networks, Embedding Layers

1. Introduction

In the United States, the number of smartphone users is estimated to reach 275.66 million in 2020 [1]. Of these users, 85 percent own Android devices [2]. Android devices occupy the majority of the smartphone market share, largely because of the flexibility and opportunities for rapid development offered by its open-source system. These benefits, however, are what also allow developers to introduce malicious software, or malware, into users' devices, most commonly for the purpose of gaining profit by holding a users technology or personal information as ransom (ransomware), stealing their credentials, or using deceptive advertising to trick users into sending money to the authors of the malware (adware).

Attackers produce malware by altering the source code of legitimate applications and re-uploading them to the market, or by authoring applications using various techniques to include harmful underlying functions that go undetected. Since malicious code utilizes many of the same dependencies, packages, and method calls as benign code, it is difficult to detect whether an application contains malware based on its contents alone. While signature-based methods for defending against malware exist, they are unable to identify all threats. The code obfuscation techniques used

by attackers are continuously developing, and the appeal of Android application development yields diverse and varied code files, offering a variety of challenges for tasks such as anomaly detection and malware classification. To combat the growing complexity of Android apps, many researchers have begun to explore various methods and models for capturing and understanding an application's structure and purpose. Such methods can be utilized in a variety of goals, including malware classification, anomaly detection, Android app development, and app research.

In this paper, we analyze and experiment with the framework originally presented in *SHNE: Representation Learning for Semantic-Associated Heterogeneous Networks* [3]. Specifically, we apply their pipeline to create and analyze embedding layers that represent relationships amongst extracted features of the code data from benign and malicious applications. These embedding layers may offer improved utility for tasks such as malware detection and application analysis.

1.1 Background

A **heterogeneous information network (HIN)** represents a network of multi-typed data by abstracting and encoding information about each and the relationships amongst them through some variation of nodes and edges. A graph representation with multi-typed nodes constituting various entities in the dataset and edges constituting pre-defined relationships between two nodes provides the foundation for a modular system that can be traversed, used for representation learning, and mined for complex features in a variety of ways. As such, a heterogeneous graph is an advantageous data structure for deep learning in domains where the nuanced relationships between entities is unknown, but necessary for accurate classification and other goals.

Resulting from this network representation is structured data. **Structured data** is data that can is in a tabular form. That is a table of rows and columns can be created by *structured data*. It is typically organized and formatted such that it is easily searchable. In the case of Android applications the relationships between apps, api calls, code blocks, and packages are structured data.

Apps	API Calls	Code Block	Package
A11	C67	B87	P78
A22	C70	B78	P55
A31	C545	B77	P75

Table 1. "A11", "C676", "B87", "P78" are structure App, API Call, Code Block, and Package nodes, respectively

Unstructured data, is data that has no specified organization or format. It cannot be formatted into a tabular structure. For this reason, unstructured data is more difficult to process and utilize in a meaningful way. Common examples of

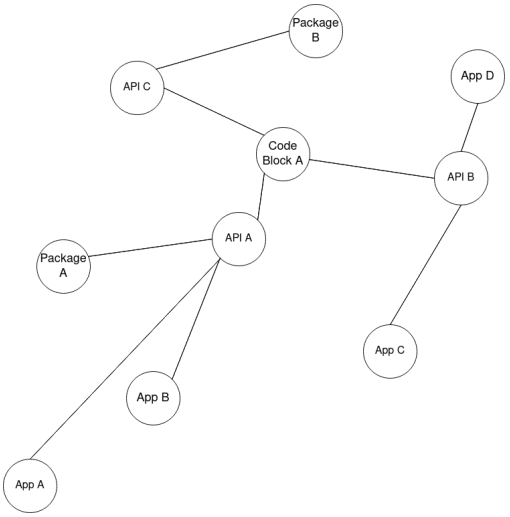

Fig. 1. Example network of App, API call, Code Block, and Package nodes

unstructured data are sound waves or text, as both mediums convey information and data, however said data cannot be structured in a tabular format.

In the context of Android app representation, extracted code from files serves as unstructured data. As code, these files have their own structure which involves clearly defined relationships between API calls- computer commands contained in various programming packages- across files. However, when the code is converted to simple text and abstracted into a chosen set of nodes and edges, its semantic value is lost. In order to utilize the semantic relations that may be found amongst the text, the problem requires deep semantic encoding to represent the general layout of the application code files. In order to accurately preserve the activity of an application, the API calls need to be understood through the context in which they appear- both geographically and semantically. Therefore,we need to incorporate both our structured and unstructured data to derive features for our model that not only preserve structural closeness, but encapsulate semantic relations within our applications.

1.2 Related Works/Challenges

While several models have been developed to represent data using a network and to obtain features from that network using various methods, the vast majority have focused specifically on structured data alone, neglecting to leverage their unstructured data as an additional tool for machine learning.

One such model which is incorporated in the SHNE framework is the *node2vec* [4] model. This model learns continuous feature representations for nodes in homogeneous networks through a biased random walk procedure that maximizes the like-

lihood of preserving network neighborhoods of nodes. The SkipGram and random walk techniques used in *node2vec* for learning node embeddings over homogeneous graphs treat different types of nodes as being the same, resulting in indistinguishable graph representations when applied to heterogeneous data.

Building on this idea is the *metapath2vec* model [5], which adapts these walks and the resulting node embedding features to heterogeneous networks by modifying the optimization function to take multiple node types into account. Still, the metapath walks introduced in *metapath2vec* for constructing heterogeneous neighborhoods of nodes, while useful when applied to structured data (nodes and edges), fall short in regards to capturing unstructured semantic relationships (text) that do not occur geographically close in the network.

2. The SHNE Framework

The SHNE, or Semantic-Aware Heterogeneous Network Embedding model, was specifically developed to address these requirements by combining relevant processes from notable deep learning models in a novel way. While it was originally applied to an academic network representing relationships between authors, papers, venues, and organizations, the techniques are applicable to any dataset consisting of structured and unstructured data, such as text.

The SHNE framework formulates an extended, heterogeneous SkipGram which, given an input node v and it's node type t, maximizes the likelihood of each type of context node (a node with an edge connecting it to the given input node v) in order to preserve geographic closeness amongst various types of nodes. In order to generate the sets of context nodes for each type (to be used in the optimization function) they use the resulting sequences from both *node2vec*'s biased random walks and *metapath2vec*'s metapath-based random walks. These walks provide feature representations of the nodes in the network.

To incorporate unstructured semantic relationships as features for model training, they developed a deep semantic encoding architecture that uses **gated recurrent units (GRU)**, a type of recurrent neural network that learns from sequences of data, to encode some types of nodes into fixed-length representations. The GRU accepts a corpus of words and their corresponding embedding sequences, pre-trained by *word2vec* [6], then encodes them as deep semantic embedding layers. After the neural network is trained on the semantic information obtained through the *word2vec* model, the resulting deep semantic embedding layers become concatenated by a mean pooling layer into a general embedding, or layout of parameters, for the type of data passed in. To illustrate, they give the example of the 'paper' type of entity in their dataset of authors, papers, and venues. In that instance, each paper's abstract was represented as a sequence of words which was then used to obtain an embedding sequence of those words, trained by *word2vec*. The sequence of words along with its corresponding embeddings (as a vector) are then passed into the GRU and encoded- along with the rest of the papers in the set- into deep

semantic embeddings (matrices). These deep embeddings of paper abstracts are then concatenated into a single matrix that represents the general semantic layout of 'paper'. The semantic encoding of the words in paper abstracts in the order in which they appear allowed the authors of SHNE to capture semantic relationships as they naturally appeared in unstructured text.

Similarly, the GRU processes the random and metapath walks performed over the network, and for each type of node included in the walks, outputs an embedding layer for the structured data associated with that type. In total, they produce four embedding layers: Author embeddings, Venue embeddings, Paper embeddings, and Paper Semantic embeddings. Finally, the embedding layers are optimized using cross entropy loss and an Adam-Optimizer [7].

These embedding layers are eventually used in several data mining tasks, including link prediction to connect associated authors who may want to collaborate on research, document retrieval of relevant papers to a query, node recommendation for suitable venues, and relevance searches for authors, venues, and author-venues. Through controlled experimentation, the authors demonstrated that SHNE outperforms the existing models in all of these mining tasks. The results obtained by incorporating joint optimization of heterogeneous SkipGram along with relevant deep semantic encoding are promising for improving data mining performance in several other domains that also rely on the current state-of-the-art models, such as the task of malware detection.

3. Data

Our data consists of benign Android APK files sourced from the most popular and most recently uploaded apps on APKMirror.com and a data set of known malicious Android APK's. Out data set consisted 500 benign and 500 malicious APK files. We ran our analysis on a data set of size 1000 due to memory limitations during the node2vec and metapath2vec metapath walk portion covered later.

APK's are compiled and packaged archive files meant to be run on the Android operating system. An APK contains app code in the form of a ***Dex*** file, as well as a manifest file, resources, and assets. The ***Dex*** file contains all the compiled code, and can be disassembled into human-readable ***smali*** code, and intermediate between Java and ***Dex***. We used decompiled smali code as our raw data.

We extracted the code text from our APK files by using the program APKTool to decompile APK archives into their smali code. From this we we first extracted four relevant pieces of information from each app. Within each app we captured, each code block, and within each code block we captured each API called and their respective package. For the purposes of this project, a "Code Block" is defined as a Java method, with API calls between. These are identifiable as the code in a smali file between the text `.method` and `.end method`. These tables of apps, code blocks, and API calls and their respective API calls and apps were saved as .json files for use in the SHNE framework.

Type	Example
App	`com.strava`
API Call	`invoke-virtual v0, Ljava/lang/Throwable;->getMessage()Ljava/lang/String;`
Package	`Ljava/lang/Throwable;`

Code Block	`.method public static <method name>()<return type package>`
	`...`
	`.end method`

Table 2. Example of different node types and their underlying data

3.1. *Exploratory Data Analysis*

In order to get a preliminary idea of what our data looked like, we ran an exploratory data analysis on our data set of 500 benign and 500 malignant apps. An important trend was that of the ratio of number of relationships between apps and their respective API calls, code blocks, and packages. Since each code block or package almost certainly contains multiple API calls, the number of relationships between apps and API calls will be much greater than the number of relationships between apps and code blocks or packages. This relationship stays the same across different types of apps (See Fig 2). This trend informs us that our network representation of our app library is rich in API call nodes, while the least amount of nodes will be app nodes. Importantly, this means that our metapath walks will need to also be rich in API call nodes in order to capture the relationships between different apps in our network.

3.2. *Data Transformation*

Using the extracted app material we then transform it again in two separate ways. The first is the structured transformation where we take each feature and create nodes within an unstructured multigraph, then we use the relationships between each app, block, package, and API to create the edges. The metadata for this graph can be seen in tables 1 and 2.

Node Type	Node Count
App	1,000
Block	3,433,430
Package	667,773
API	3,917,764

Table 3. Node type and count in unstructured multigraph

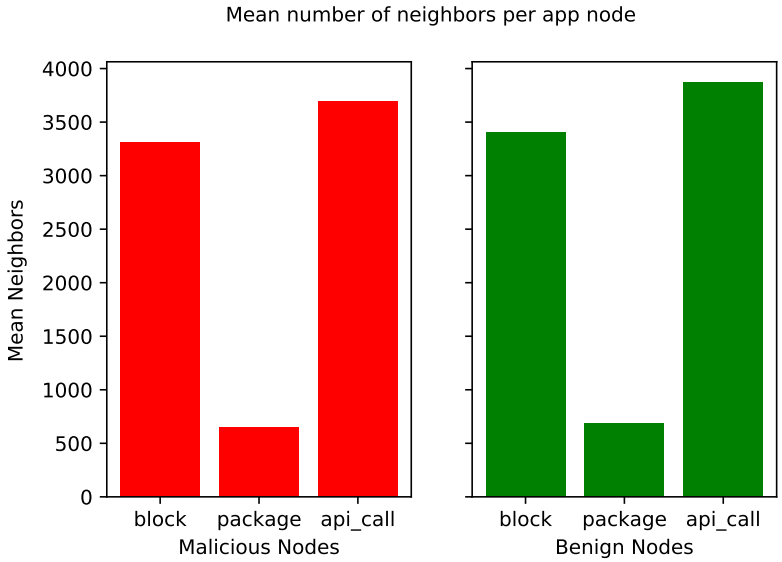

Fig. 2. Graph of average node types for malicious and benign app nodes

Edge Type	Edge Count
API to Block	17,777,302
API to App	15,385,901
API to Package	3,917,764

Table 4. Edge type and count in unstructured multigraph

With the graph structure created we can begin the traversal. As discussed earlier SHNE is dependent of the creation of metapath walks which follow a schema for node traversal. Starting at each app node we performed three different kinds of metapath walks each with a 13 node pattern for length 480. These three walks were repeated 10 times for each of the 1000 apps. The result of this is a data set with 30,000 walks of length 480 which can be used to understand the inherent structure of each of the apps.

Following the execution and completion of all 30,000 walks we then look to create our unstructured transformation. Mimicking the original SHNE paper which uses paper abstracts this unstructured component is created from each app's list of API calls, in order. Where as a paper abstract is composed of words in sentences, our unstructured "abstracts" are composed of API calls in blocks for each app. An app with 100 API calls, some repeating, some not, would consequentially have an abstract of length 100. We then pass these abstracts through a word2vec model and create and embedding for each unique API call.

At this stage we now have three data transformations, a structured list of metapath walks, an unstructured embedding layer for each abstract API, and the original content file, comprised of the apps and their api-abstracts. With these three files we are then ready to incorporate the SHNE architecture.

3.3. *SHNE Modifications*

The original SHNE code provided by Zhang et al. served as the foundation for adapting their model to our domain. Originally, the SHNE code was built for three node types: Authors, Papers (which includes the unstructured layer), and Venues. These existing nodes were converted into their app equivalents, which can be seen in Table 3.

Author nodes	$\longrightarrow$	API nodes
Paper nodes (inc. unstructured)	$\longrightarrow$	App nodes (inc. unstructured)
Venue nodes	$\longrightarrow$	Package nodes

Table 5. SHNE to appSHNE node type equivalency

In addition to the three existing nodes from the original SHNE project, a fourth node type- 'block' was added. This block type node is computationally equivalent to the original Venue type node in the SHNE model. With this modification, the SHNE model is now ready to produce an embedding layer from our transformed app data.

3.4. *Hyperparameter Tuning*

In conjunction with obtaining the results was a lengthy and involved parameter tuning process where we attempted to optimize the *metapath2vec*, *word2vec*, and SHNE models. Given the scope and range of choices at our disposal it is likely our final set of parameter choices wasn't fully optimal. Some conclusions we did make based on multiple iterations of experimentation were such:

- Increasing the window size in *word2vec* makes the model more sensitive to domain similarity rather than sensitive to syntactic similarity.
- Increasing the number of metapath walks allows the model to better capture node neighborhoods and increases the likelihood of finding commonly repeating paths/patterns.
- Walk length also plays a major factor in the performance of the model, as longer walks may be required to capture more complex relationships.

3.5. *metapath2vec Baseline Layer*

In order to understand how incorporating unstructured and structured data in the SHNE pipeline changes the representation of the resulting embedding layer we need

to produce a "baseline" embedding layer for comparison. Since typical app malware detection methods lean heavily on understanding the structured components of app code [8], *metapath2vec* was chosen as a sufficient model for creating our baseline layer. Using the already created metapaths from the data transformation process, we created embedding layers for the apps, blocks, packages, and APIs. These embeddings can then be used for comparison against the resulting appSHNE layers.

4. Results

In this section we will outline and discuss the resulting embedding layer created by the SHNE architecture and how it contrasts to the *metapath2vec* layer, focusing on three main points of interest:

- Distribution: How do the resulting layers compare in terms of distribution of values, and how do these layers contrast to our *metapath2vec* layer.
- Dimentionality reduction: Do the resulting dimention reduced layers cluster into meta categories such as benign-malicious, package types, or app families.
- Classification: Can we use the resulting layers for classification tasks such as predicting whether an app could pose risk to a user.

Although our results are focused on these three topic the utility of the resulting embedding layers extends much further to a plethora of tasks including link-prediction, similarity search, and app research.

4.1. *Distribution*

The output of appSHNE and the output of *metapath2vec* can be agrrigated by summation into 1000x1 vectors which can be plotted on a histogram to outline the general distribution of the data. This process of aggregation and plotting can be seen in figure 1 where the histograms are plotted along the same axes.

The distribution of the layers contrasts greatly. When compared to the *metapath2vec* layer the appSHNE layer lacks a significant amount of variance. Because both layers are built from the same metapath walks, appSHNE is likely losing most of its variance because of the addition of the unstructured data. This suggests that appSHNE is unable to identify which app-API, API-API, API-Package, or API-block combinations are most representative of malware, or which combinations suggest that the app deviates from the norm. A likely explanation for this is the sheer quantity of APIs when compared to apps. Considering that SHNE was originally developed on a author-paper-venue dataset where the ratio of authors to papers is roughly similar it is plausible that the architecture doesn't scale well with noisy data.

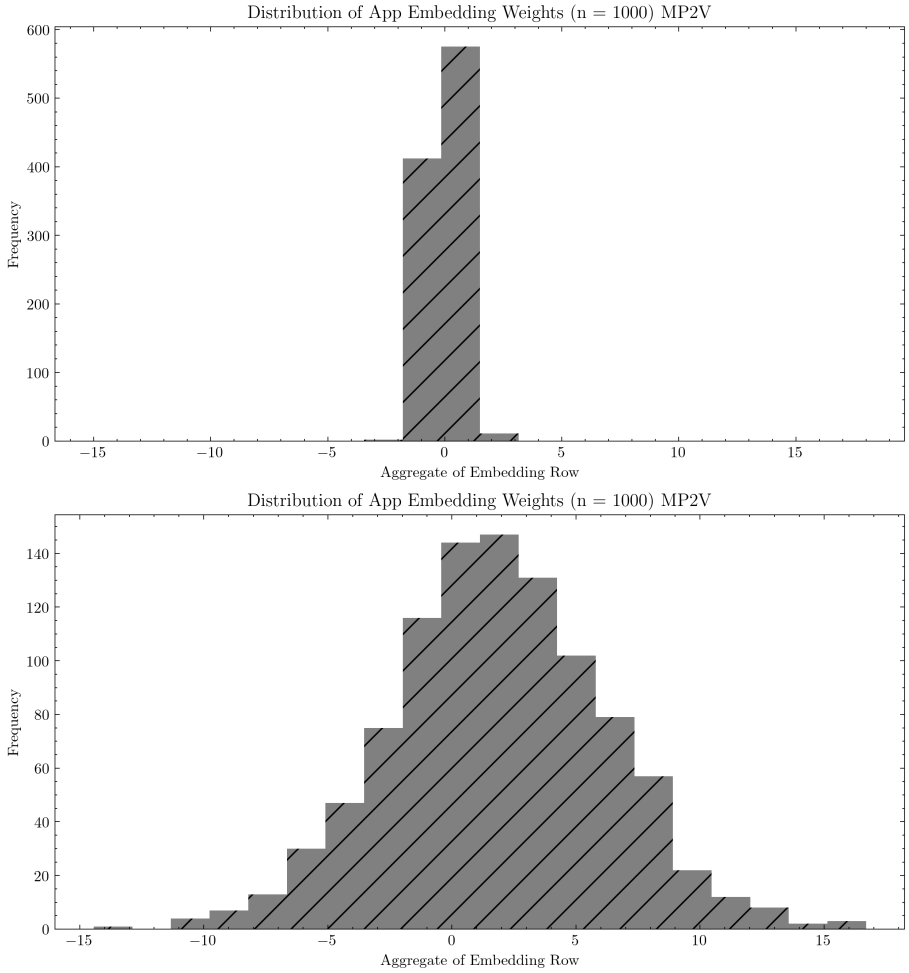

Fig. 3. The distribution of the summed embedding layer for appSHNE (Top) and MP2V (Bottom)

4.2. *Dimensionality reduction*

Dimensionality reduction on the generated embeddings for *metapath2vec* and appSHNE can be done in a number of ways. Specific to this project we focused on two main methods, Principle Component Analysis which reduces the dimentionality by prioritising the dimensions with the most variance and t-distributed stochastic neighbor embedding or tSNE which works by constructing probability distributions on the dimensions of the data. Figure 2 highlights the results for PCA and figure 3 highlights the results for tSNE.

For both examples appSHNE was unable to reduce the dimentionallity of the data in a way that produces meaningful clusters, let alone meaningful clusters for the benign-malware division in the dataset. This contrasts to the *metapath2vec* em-

Fig. 4. Principle component analysis for appSHNE (Top) and *metapath2vec* (Bottom)

bedding layer which is able to produce some level of clustering in both the PCA example and the tSNE example. Since both layers are constructed from the same metapath walks the inability for appSHNE to produce embeddings with intrinsic clusters suggests that the addition of unstructured data in the appSHNE architecture may be interfering with the creating of an embedding layer with sufficient varience.

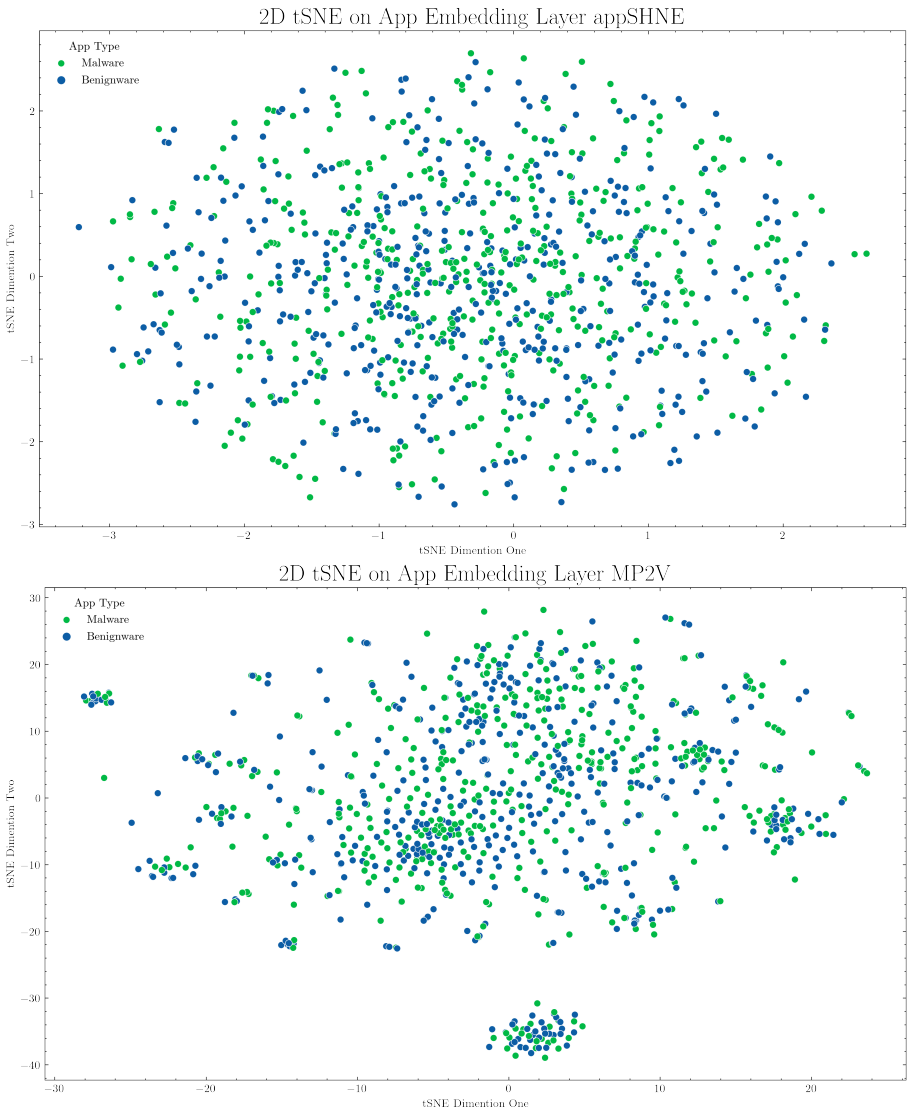

Fig. 5. t-distributed Stochastic Neighbor Embedding for appSHNE (Top) and *metapath2vec* (Bottom)

4.3. *Classification*

To evaluate the quality of the appSHNE output embedding layer we trained and contrasted two rudimentary classification models, a logistic regression model and a support vector machine. Identical models were also trained for the output *metapath2vec* layer. The results of the classifications models can be seen in table 5 where a true-positive represents an app correctly classified as benign and a true-negative

represents an app correctly classified as malicious. To evaluate the performance of these models two metrics were chosen, accuracy and F1 score.

Method	TP	FP	TN	FN	F1	Acc
MP2V SVM	73	106	87	64	0.462	48.48 %
MP2V Logistic Regression	93	86	78	73	0.53	51.81 %
appSHNE Logistic Regression	76	103	101	30	0.498	53.60 %
appSHNE SVM	75	104	106	45	0.5	54.85 %

Table 6. Simple logisitic regression and support vector machine classification results for appSHNE and *metapath2vec*.

The performance of all models, both those trained on *metapath2vec* and those trained on appSHNE was mediocre. The best performing model by accuracy was the appSHNE SVM which produced a model slightly better than random chance. This result isn't significant and suggests that the produced *metapath2vec* and appSHNE embedding layers aren't effective tools for classification in their current states. The likely explanation for this under-performance is that the metapath walks failed to capture the inherent relationships between the apps and the APIs and hence were unable to establish a significant difference between benign apps and malicious apps.

4.4. *Discussion*

During our discussion of the project and further review of the original SHNE experiment, one key limitation of appSHNE is the ratio of app nodes to API nodes. In the original SHNE paper the ratio of author nodes to paper nodes is significantly less exaggerated at a author to paper ratio of $1.44 : 1$. This contrasts greatly to the ratio of app to API nodes at $1 : 3917$. As a result the metapath walks are unable to properly capture the relationship of specific API calls to specific apps, failing to create enough variance in the resulting embedding layer. This shortcoming is further exaggerated by the unstructured layer which captured the API call relationships. Because of the scale of APIs and the limited number of apps the SHNE model was likely overwhelmed with noise, rather than a selection of the most descriptive API combinations. This results in the appSHNE output embedding layers lacking much significant insight into the apps or relationships between the app's components.

5. Future Work

Given the utility and range of possibilities for the created embedding layers there is much future work possible with relation to SHNE and app research. With sufficient time and resources some key areas of interest include:

- Dramatically increasing the ratio of app nodes to API nodes.

- Optimizing and producing better meta-paths for capturing the structured component of the app data.
- Optimizing meta-paths for better understanding of api, package, and block relationships, rather than primarily for understanding app relationships.
- Capturing more node types within the structured data, specifically, api invoke types.
- Using app manifests for the unstructured encoding, rather than the current api call occurrence lists.

This list is by no means exhaustive and is centered specifically on mobile apps. This research could be applied to exploring the relationships between unstructured and structured data across many domains much like how the original SHNE paper explored its application in author-venue-paper tasks.

6. Conclusions

Based on the results obtained from the appSHNE architecture we can clearly see that more work needs to be done developing the approach and technique. Whilst not successful at providing significantly meaningful insight into the nature and composition of the android application, it does hold promise. The symbiosis of structured and unstructured app data holds potential for unlocking greater insight into the purpose of an application and its design. Given that techniques such as MetaPath2Vec have already been proven to provide significant results leveraging the structured component of the data, the logical next step is to further develop and incorporate the unstructured component. Although appSHNE was not a success it does serve as lesson to which further exploration in this field can learn and follow from.

Acknowledgments

Special Thanks to Aaron Fraenkel and Shivam Lakhotia for mentoring this project. Thanks to the UCSD-DSMLP server for hosting the project.

References

[1] O'Dea, S. 2020. Number of smartphone users in the United States from 2018 to 2024 (in millions). Statista. https://www.statista.com/statistics/201182/forecast-of-smartphone-users-in-the-us/

[2] IDC. 2020. Smartphone Market Share. https://www.idc.com/promo/smartphone-market-share/os.

[3] Zhang, Chuxu, Ananthram Swami, and Nitesh V. Chawla. 2019. "SHNE: Representation Learning for Semantic-Associated Heterogeneous Networks." WSDM

2019, no. Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining (January): 690-698. https://doi.org/10.1145/3289600.3291001.

[4] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In KDD. 855–864

[5] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable Representation Learning for Heterogeneous Networks. In KDD. 135–144.

[6] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In NIPS. 3111–3119.

[7] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014).

[8] Fan, Yujie Hou, Shifu Zhang, Yiming Ye, Yanfang Abdulhayoglu, Melih. (2018). Gotcha - Sly Malware!: Scorpion A Metagraph2vec Based Malware Detection System. 253-262. 10.1145/3219819.3219862.