Embedding Network Information for Machine Learning-based Intrusion Detection

Jonathan D. DeFreeuw

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Engineering

Joseph G. Tront, Chair
Randy Marchany
Yaling Yang

December 12, 2018
Blacksburg, Virginia

Keywords: intrusion detection, machine learning, word embeddings

Embedding Network Information for Machine Learning-based Intrusion Detection

Jonathan D. DeFreeuw

(ABSTRACT)

As computer networks grow and demonstrate more complicated and intricate behaviors, traditional intrusion detections systems have fallen behind in their ability to protect network resources. Machine learning has stepped to the forefront of intrusion detection research due to its potential to predict future behaviors. However, training these systems requires network data such as NetFlow that contains information regarding relationships between hosts, but requires human understanding to extract. Additionally, standard methods of encoding this categorical data struggles to capture similarities between points. To counteract this, we evaluate a method of embedding IP addresses and transport-layer ports into a continuous space, called IP2Vec. We demonstrate this embedding on two separate datasets, CTU'13 and UGR'16, and combine the UGR'16 embedding with several machine learning methods. We compare the models with and without the embedding to evaluate the benefits of including network behavior into an intrusion detection system. We show that the addition of embeddings improve the F1-scores for all models in the multiclassification problem given in the UGR'16 data.

Embedding Network Information for Machine Learning-based Intrusion Detection

Jonathan D. DeFreeuw

(GENERAL AUDIENCE ABSTRACT)

As computer networks grow and demonstrate more complicated and intricate behaviors, traditional network protection tools like firewalls struggle to protect personal computers and servers. Machine learning has stepped to the forefront to counteract this by learning and predicting behavior on a network. However, this learned behavior fails to capture much of the information regarding relationships between computers on a network. Additionally, standard techniques to convert network information into numbers struggles to capture many of the similarities between machines. To counteract this, we evaluate a method to capture relationships between IP addresses and ports, called an embedding. We demonstrate this embedding on two different datasets of network traffic, and evaluate the embedding on one dataset with several machine learning methods. We compare the models with and without the embedding to evaluate the benefits of including network behavior into an intrusion detection system. We show that including network behavior into machine learning models improves the performance of classifying attacks found in the UGR'16 data.

Acknowledgments

I would like to thank Dr. Tront and Professor Marchany for their patience and wisdom in the last few years as I learned what it meant to truly research and learn. Their efforts in helping me succeed cannot be overstated, and their help has certainly set me up for success in my future career.

I would like to acknowledge the love and support of my parents, Brian and Dana, my girlfriend, Jessica, and Brian and Deanne Burch. Without their positivity during my times of struggle, this thesis would not have been possible.

I want to thank the members of the IT Security Lab, particularly Ryan Kingery and Zachary Burch, for being awesome sounding boards for my thoughts throughout this entire work.

I also want to acknowledge the National Science Foundation for their funding through the CyberCorps: Scholarships for Service program.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Research Problem . . . 2

1.2 Proposed Solution . . . 3

1.3 Thesis Outline . . . 3

2 Background 4

2.1 Categorical Data Representation . . . 4

2.1.1 Encoding . . . 5

2.1.2 Embedding . . . 5

2.2 Word2Vec . . . 8

2.3 Visualization . . . 10

2.3.1 PCA — Principal Component Analysis . . . 10

2.3.2 t-SNE — t-Distributed Stochastic Neighbor Embedding . . . 11

2.3.3 Comparison . . . 12

3 Review of Literature **16**

3.1 Security Datasets 16

3.1.1 DARPA 16

3.1.2 KDD'99 17

3.1.3 NSL-KDD 18

3.1.4 CTU'13 19

3.1.5 UNSW-NB15 19

3.1.6 UGR'16 20

3.2 Machine Learning 21

3.2.1 Clustering 22

3.2.2 Neural Networks 23

3.2.3 Decision Trees 25

3.2.4 Categorical Data Representation 27

4 Experimental Design **28**

4.1 Binned IP2Vec 29

4.1.1 Choosing Word Pairs 29

4.1.2 Embedding Model Design 31

5 Evaluation **35**

5.1 Binned IP2Vec 35

5.1.1 CTU'13 . . . 36
5.1.2 UGR'16 . . . 40
5.2 Intrusion Detection . . . 45
5.2.1 Data Engineering . . . 46
5.2.2 Feature Space . . . 48
5.2.3 Supervised Learning . . . 49
5.2.4 Analysis . . . 55

6 Discussion **57**

6.1 Future Work . . . 57

7 Conclusion **59**

Bibliography **60**

Appendices **71**

Appendix A Feature Importances **72**

List of Figures

2.1 Example of an embedding of a TCP port 6

2.2 Comparison between categorical data representations 7

2.3 Sliding window for determining target and context words in word embedding 8

2.4 Samples of the MNIST handwriting dataset 13

2.5 PCA reduction on the MNIST dataset 14

2.6 t-SNE reduction on the MNIST dataset 15

4.1 Choosing word pairs in IP2Vec 29

4.2 Network design for IP2Vec 31

5.1 Graphing pipeline for IP2Vec 37

5.2 2D t-SNE of 32-dimensional IP2Vec embedding with *min_samples* = 1 38

5.3 2D t-SNE of 32-dimensional IP2Vec embedding with *min_samples* = 2 39

5.4 2D t-SNE of 32-dimensional IP2Vec embedding with *min_samples* = 5 39

5.5 Rolling average loss of IP2Vec on UGR'16 42

5.6 t-SNE reduction of 32-dimensional IP2Vec embedding on UGR'16, with `min_samples = 1` 44

5.7 t-SNE reduction of 32-dimensional IP2Vec embedding on UGR'16, with `min_samples = 2` 45

A.1 Importances for Features Ranked 1-75 in Non-Embedded XGBoost 73

A.2 Importances for Features Ranked 76-150 in Non-Embedded XGBoost 74

A.3 Importances for Features Ranked 1-75 in Embedded XGBoost 75

A.4 Importances for Features Ranked 76-150 in Embedded XGBoost 76

A.5 Importances for Features Ranked 1-75 in Non-Embedded Random Forests . . 77

A.6 Importances for Features Ranked 76-150 in Non-Embedded Random Forests 78

A.7 Importances for Features Ranked 1-75 in Embedded Random Forests 79

A.8 Importances for Features Ranked 76-150 in Embedded Random Forests . . . 80

List of Tables

2.1 Comparing CBOW vs skip-gram for generating word pairings 9

5.1 Hardware-Software Configuration 35

5.2 Training Statistics for IP2Vec on CTU'13 40

5.3 Server statistics for UGR'16 43

5.4 Client statistics for UGR'16 43

5.5 Training Statistics for IP2Vec on UGR'16 44

5.6 Period of attacks chosen from days in UGR'16 [14] 47

5.7 Nonembedded features for supervised learning 49

5.8 Embedded features for supervised learning, using binned IP2Vec. 49

5.9 Evaluation metrics for XGBoost with and without IP2Vec 52

5.10 Confusion matrix of test set for XGBoost using non-embedded features . . . 52

5.11 Confusion matrix of test set for XGBoost using features embedded with IP2Vec 53

5.12 Evaluation metrics for random forests with and without IP2Vec 53

5.13 Confusion matrix of test set for random forests using non-embedded features 54

5.14 Confusion matrix of test set for random forests using features embedded with IP2Vec 54

5.15 Evaluation metrics for MLP with and without IP2Vec 55

5.16 Confusion matrix of test set for MLP using non-embedded features 55

5.17 Confusion matrix of test set for MLP using features embedded with IP2Vec . 56

List of Abbreviations

CART Classification and Regression Tree

CBOW Continuous Bag-of-Words

CNN Convolutional Neural Network

LSTM Long Short-Term Memory

MLP Multi-Layer Perceptron

NIDS Network Intrusion Detection System

NLL Negative Log-Likelihood

NLP Natural Language Processing

PCA Principal Component Analysis

RNN Recurrent Neural Networks

SMOTE Synthetic Minority Oversampling Technique

SVM Support Vector Machine

t-SNE t-Distributed Stochastic Neighbor Embedding

Chapter 1

Introduction

As networks continue to grow in complexity and traffic throughput, the tools used to monitor them for malicious behavior have struggled to keep the pace. These systems, called Network Intrusion Detection Systems (NIDSs), are used to analyze the traffic on a network and assist network administrators in detecting inbound and outbound attacks.

Most NIDSs in use today, such as Snort [1] and Bro [2], rely on a corpus of known attack signatures in order to detect incoming malicious traffic. These signature-based NIDSs are extremely effective at detecting known attacks, but are inadequate at identifying novel attacks. Rather than looking for particular values within a piece of data, other systems use statistical analysis to determine if traffic deviates from a known 'normal' behavior. Anomaly-based NIDSs are more generalizable to new attacks, but tend to incorrectly classify benign behavior as anomalous. Anomaly-based detection tools can be configured using data collectors such as Splunk [3] and Elasticsearch [4], generating alerts when collected data deviates from the norm.

Machine learning techniques have been explored in security research as a means to improve NIDSs. Clustering algorithms such as [5] and [6] detect outliers in the well-known security datasets DARPA [7] and NSL-KDD [8]. Neural networks, including convoluted and recurrent networks, have been developed using the same datasets, showing even more accuracy in detecting outliers given a labeled dataset. However, while these algorithms may work well for detecting anomalies in datasets approaching 20 years old, fewer models have been trained

using real-life data.

Cisco's NetFlow protocol has been used in previous work for intrusion detection and traffic classification [9]–[11] due to its efficient way of describing the behavior of a network. NetFlow consists of several important pieces of information regarding a network connection, or flow, including source and destination IP addresses, ports, protocols, and flags. The majority of data within a flow is categorical, meaning while there are numbers to represent the data, the information that is inferred from the data is not easily represented in a numerical space. This makes feeding NetFlow into machine learning models difficult due to limited methods in making NetFlow interpretable to a model.

1.1 Research Problem

Although machine learning has proven to be a rich source of new research for intrusion detection systems, augmenting network data for machine learning models remains a difficult task. Due to the complexities of the network stack, with protocols like IP, TCP, and UDP, a significant amount of information is lost when interpreting addresses and ports as their integer representations, as is required for most learning models. If we choose to train models without features such as IP address or port, we lose out on any potential information that those features could give our models.

While a considerable amount of work has been done to analyze machine learning methods for network intrusion detection (see 3.2), a majority of work has been done using synthetic datasets, meaning the data was generated in an environment designed to mimic a real-world network. This is not a preferred method, particularly if we want to explore the deployment of such intrusion detection systems in a real-world environment. Crafting real-world Internet traffic is a non-trivial issue [12], especially in the complex and protocol-rich atmosphere of

today's Internet.

1.2 Proposed Solution

This thesis aims to evaluate the use of an embedding technique devised by Ring et al., named IP2Vec [13]. IP2Vec enables the encapsulation of network behavior into a machine understandable format, called an *embedding*. We implement IP2Vec, and modify it for use in larger networks than the original implementation, as well as for potential use in a streaming environment. We compare implementations to determine the effect of the modifications used.

To gauge the effectiveness of the information gathered by the embedding, we use IP2Vec to embed the network features of NetFlow data before training supervised learning models for intrusion detection. Rather than use synthetic network data, we use the UGR'16 dataset [14]. UGR'16 contains includes a collection of synthetic and live traffic captured in a working enterprise environment, recorded over several months. The models are evaluated by F1-scores and confusion matrices of the test data.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 provides background knowledge regarding the concepts utilized in the system design. Chapter 3 examines related research in the fields of security datasets and machine learning. In Chapter 4, our design is described and with the results of the evaluation in Chapter 5. We discuss future work in Chapter 6 and finally conclude the thesis in Chapter 7.

Chapter 2

Background

2.1 Categorical Data Representation

In machine learning, we refer to two types of data inputs: *continuous* and *categorical* data. Continuous data is data that has meaning when represented as a number: for example, total number of bytes or packets, and time in seconds. This type of data is easily recognizable by learning algorithms, as functions can be made to map input to output in most cases. Categorical data refers to data that exists as a finite set of values. For instance, there are only 2^{32} IPv4 addresses, so an IP would be categorical. Transport layer protocols are also categorical (TCP, UDP, ICMP). Categorical data tends to be represented as strings, while continuous data is represented as floats or integers.

A shortcoming of most machine learning techniques is their inability to handle categorical variables directly. While algorithms such as random forests can handle categorical data, other methods such as clustering or XGBoost require modifications to the data. An example of categorical data is the source port field in a flow record. While the ports are represented as numbers (SSH:22, HTTP:80), models would learn relationships between ports relative to their numerical representation, not the service offered. This introduces problems when services use alternative ports such as 8080 for HTTP. During training, we would prefer a model to learn that ports 80 and 8080 are more similar than 80 and 81.

2.1.1 Encoding

The simplest way to overcome the issue of categorical data is to use *one-hot encoding*. This converts discrete data into sparse vectors, where in an encoding of length n, there are $n-1$ zeros, and a single 1 in the vector. The '1' value represents the data that we are trying to encode. In the encoding vector, there is an index for each unique value within the range of the feature. This means that to encode a source port, our vector will have a length of 65536.

Because the vector contains a single index for each unique value, encoding becomes inefficient when there is a large set of values, especially a large number of rare and underutilized values. To combat this, we can condense the feature vector by only creating encodings for the most frequent values within the feature's unique values. For example, when encoding TCP/UDP ports, we can reserve encodings for common ports like 22, (SSH), 443 (HTTPS), and 3389 (RDP). For all other ports, a single index is reserved in the vector. When encoding the 'other' ports, this value is 1, and all other values in the vector are 0. We refer to this method of encoding as *binned one-hot encoding*.

2.1.2 Embedding

While binned one-hot encoding reduces the vector length for our port problem, it significantly reduces the amount of information that can be learned for all of the words in the 'other' category. One-hot encoding becomes infeasible for other categorical variables, particularly words for Natural Language Processing (NLP). For example, if we wanted to use binned one-not encoding on the 100 most frequently used TCP/UDP ports, we would lose context on $100/65536 = 99.85\%$ of usable ports.

Instead, the preferred technique for representing large numbers of unique values is called *embedding*. We refer to this collection of unique values as a *vocabulary* in the context of

embeddings. To create an embedding, we attempt to predict a output word given an input word, generating a dense weight matrix (meaning most attributes are non-zero) for the entire vocabulary. This results in a matrix of size $n \times m$, where n is the size of the vocabulary and m is the size of our embedding. Each row represents a single word, accessed by multiplying a one-hot encoding vector by the weight matrix. Figure 2.1 shows an example of an HTTP port being embedded. The embedding can be trained independently from other models, or integrated into larger models as a preprocessing layer.

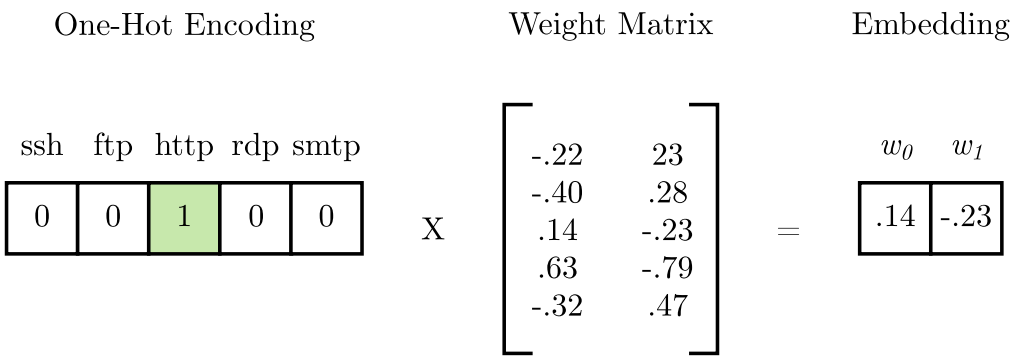


Figure 2.1: Example of an embedding of a TCP port

The advantage of using an embedding over an encoding is the ability to predict *semantic similarity*, meaning inputs are similar within the same context. This similarity is represented within the floating point values in each input's vector, and can be calculated using a *cosine similarity*. The cosine similarity measures the angle between two vectors, where the smaller the angle, the more similar two vectors. Not only does this similarity provide more context to the relationship between two words, the embeddings are also created in a vector space orders of magnitude smaller than a one-hot encoding. This creates more rich features for a classification problem, while also creating a smaller feature space. We describe a comparison of our categorical data representation techniques in Figure 2.2.

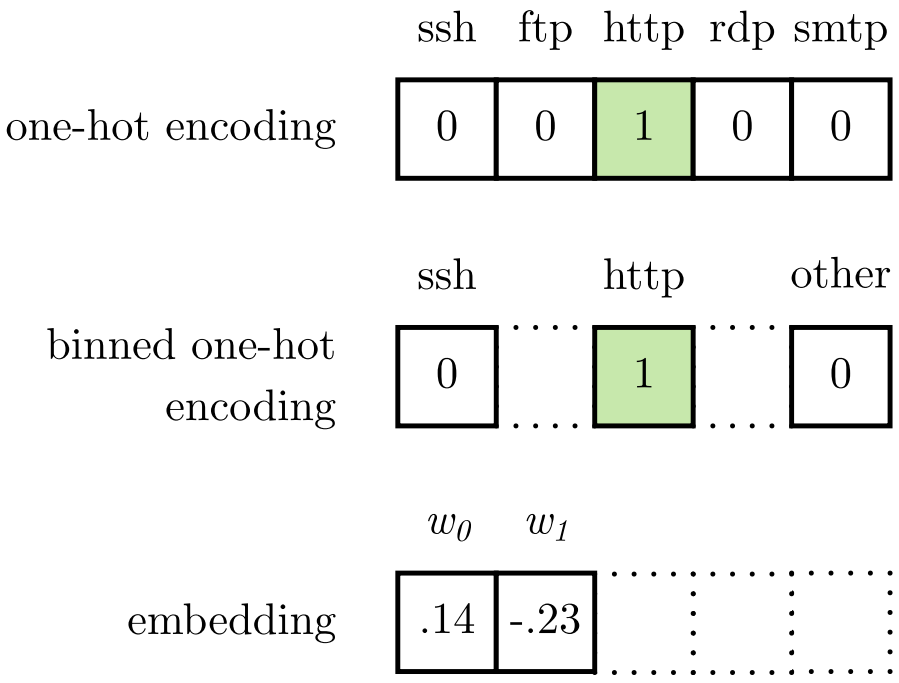


Figure 2.2: Comparison between categorical data representations
Using the methods described in this section, we can model a representation of port 80 (HTTP). Given 5 ports, a one-hot encoding would creating a vector of length 5, while binned encoding would return a slightly smaller vector, given SSH and HTTP are the most frequent ports. An embedding results in the smallest vector, with $\|v\| = 2$.

The example given creates vectors for a single feature (port number), but we can build a vocabulary using a combination of features, such as IP addresses and protocols. By combining features into a single embedding, a model can find similarities between a source IP address and a port number, or a protocol and destination IP address. Training using many different features to create an embedding allows more context to be included, such as how related a port is to a transport layer protocol (e.g. TCP/UDP). This technique is described in Section 3.2.4.

The main drawback with an embedding is its dependence on a separate model to train; a new embedding would have to be created any time a new word is added to the vocabulary. This

makes embeddings difficult to utilize for online learning scenarios as retraining an embedding could take several hours each time a new word appear. A simple fix is to add an 'other' category similar to encoding, though this minimizes the benefits of embedding for any word in that category. However, if the initial training vocabulary is large enough (spans enough time), then new words should be few and far between, leaving the 'other' category as a worthy placeholder for infrequent words.

2.2 Word2Vec

A basic word embedding can be trained with a single layer in a neural network by using a target word to predict a context word. Algorithms such as Word2Vec by Google researchers [15] create more robust embeddings by adding more operations to their model than simply predicting an output. Word2Vec uses one of two different means of creating target-context word pairs: *continuous bag-of-words* (CBOW) and *skip-gram* In a CBOW architecture, the model takes the words surrounding a target word and uses them to predict the current word. In essence, the model will attempt to predict a word given its context. In Table 2.1a, we use w_{i-2}, w_{i-1}, w_{i+1}, and w_{i+2} to predict w_i. The opposite of CBOW, a skip-gram architecture will use a target word to predict the context word. Table 2.1b shows that we will use w_i to predict w_{i-2}, w_{i-1}, w_{i+1}, and w_{i+2}.

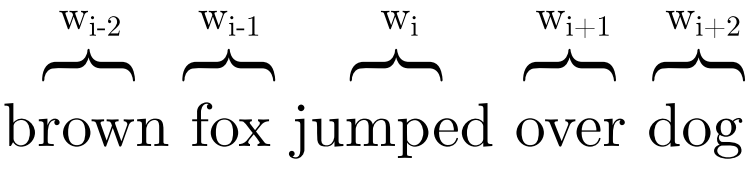


Figure 2.3: Sliding window for determining target and context words in word embedding

Input	Output
brown	jumped
fox	jumped
over	jumped
dog	jumped

(a) CBOW

Input	Output
jumped	brown
jumped	fox
jumped	over
jumped	dog

(b) Skip-Gram

Table 2.1: Comparing CBOW vs skip-gram for generating word pairings

Word2Vec includes other functions to its embedding model that set it apart from standard word embeddings:

Negative Sampling. Training embeddings for large vocabularies requires many updates during backpropagation: for example, using the Google News dataset, there are 3,000,000 unique words, which in [16] is embedded to 300 dimensions. Updating all $3,000,000 \cdot 300 = 900,000,000$ weights within the embedding matrix during backpropagation is expensive and would take a large amount of time. Instead, negative sampling causes a model to only update the weights for t words. This is done by using binary classification, learning when two words are similar (1) or dissimilar (0). Mikolov et al. propose $t = [5 - 20]$ for small datasets and $t = [2 - 5]$ for larger datasets, though the definitions for 'small' and 'large' are not clear.

Subsampling of Frequent Words. In creating word pairings using CBOW or skip-gram, many words in a vocabulary will carry a significantly heavier weight in the frequency distribution, though less weight in a specific window. For example, common English words such as "the" and "a" will make up a large portion of a training set, but do not provide much semantic knowledge in a sentence. To overcome this, Mikolov et al. subsampled their training set to reduce the number of word pairings to train on, based on their frequency within the vocabulary. The probability that a word would not be trained on was determined using the formula [15, eq. (5)]:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{2.1}$$

where t is a threshold around 10^{-5} and $f(w_i)$ being the frequency of word w_i.

Additive Compositionality The authors of [15] show that because the target and context word weights share a linear relationship in their model, vectors created by Word2Vec can be added and subtracted to create meaningful vectors. For example, when W is the embedding operation:

$$W(man) - W(woman) + W(king) \approx W(queen) \tag{2.2}$$

Given the embeddings for the words *man*, *woman*, and *king*, we can reasonably approximate the vector that would represent the word *queen* within that vocabulary.

2.3 Visualization

In order to visualize the results of trained machine learning models and embeddings, algorithms are needed to condense an n-dimensional space into a 2- or 3-dimensional space. This section explains two separate dimensionality reduction methods, called Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE).

2.3.1 PCA — Principal Component Analysis

Principal Component Analysis is an algebraic method of producing m dimensions for an n-dimensional space. To do so, PCA calculates the *principal components* of a dataset, which are new features that attempt to capture the most variance from the original dataset. These

components are linear combinations of the n features, meaning relationships between features are captured within each of the individual components. However, principal components are independent from one another, which is useful for calculating linear regressions on reduced datasets.

2.3.2 t-SNE — t-Distributed Stochastic Neighbor Embedding

While PCA attempts to create m features that represent the most variance in an n-dimensional vector, we focus more on global similarities using PCA. In some cases, this is not a preferred method to visualize a high-dimensional dataset because similarities between distant points should not greatly effect the similarities between closer points. Instead, we would prefer to preserve the local similarities between all points across all features, and use those similarities in a new dimensional space. This is the goal of t-SNE: to maintain similarities of points in two different spaces, while minimizing the error between the two sets of similarities.

Van der Maaten and Hinton proposed t-SNE [17], a widely used method [13], [18] to map high-dimensional data to low-dimensional (but still related) manifolds. To do so, t-SNE first uses a probability as a representation of similarity. The algorithm calculates the probability $p_{j|i}$ for any point x_i, that another point x_j will be chosen from its neighbors. Points are chosen in proportion to a Gaussian distribution centered around x_i. This calculation is done across all points in the set. A similar method is applied to a low-dimensional representation of the data.

With these probabilities in hand, t-SNE uses Kullback-Leibler divergence as a means to calculate the error between two probabilities. The cost C between the high-dimensional probabilities P and low-dimensional probabilities Q can be defined as [17, eq. (2)]

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{j|i} log \frac{p_{j|i}}{q_{j|i}} \tag{2.3}$$

In order to minimize this error, t-SNE will perform gradient descent, much like how neural networks or gradient-boosted machines such as XGBoost decrease the loss after classification. However, this gradient descent is weighted to preserve the structure of local similarities. To do this, t-SNE will penalize (increase the weight of) points that are similar in the high-dimensional space, but dissimilar in the low-dimensional space. Nothing is done to the weights if originally dissimilar points are similar in the new dimensions. This method of 'tailing' the probability distribution prevents points that are further apart from one another from causing large changes to the overall representation.

We can configure the algorithm in many ways, with the main focus being on the idea of *perplexity*. The perplexity of t-SNE is used to adjust the effective number of neighbors that are within the Gaussian around x_i. When determining neighbors, the algorithm utilizes a random walk of a neighbor graph, which can be seeded so researchers can reproduce reductions. The metric used for distance calculations between points can also be adjusted to fit the dataset, such as using Minkowski or cosine distance metrics.

2.3.3 Comparison

To compare these methods of dimensionality reduction, we can reduce the feature space of the MNIST handwriting dataset. This dataset is one of the most prominent datasets for machine learning, as it small and interpretable. It represents thousands of handwritten numbers, from 0-9, and are stored as 28x28 pixel images. Examples are shown in Figure 2.4 below. For the purposes of our comparison, we spread the 2-D matrix of pixels into a single 784-dimensional vector, where each dimension represents a single pixel. We then reduce the

784 dimensions into two dimensions for graphing.

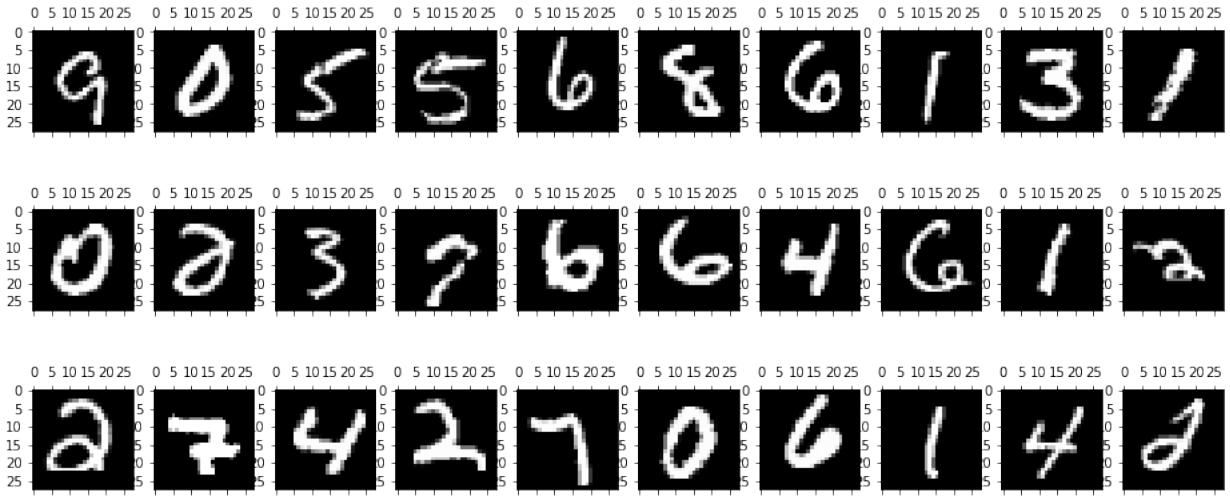


Figure 2.4: Samples of the MNIST handwriting dataset

We first use the PCA dimensionality reduction method to find the two most effective principal components for the 784-dimensional space. These two components represent about 17% of the variance in our data, meaning that all of our points are fairly similar to one another. This graph is shown in Figure 2.5. As we can see, all points are clustered together in a single blob, showing the PCA does not do well in distinguishing different values within our data.

We can compare this to the visualization using t-SNE with a perplexity of 40. The algorithm iterates 300 times to decrease the error between the conditional probabilities of the high and low dimensional data. As seen in Figure 2.6, handwritten digits that are visually similar are grouped together, and those that are dissimilar are further apart. For example, 4's (purple) and 9's (navy) are intertwined in their clusters, but 0's (red) are away from other digits.

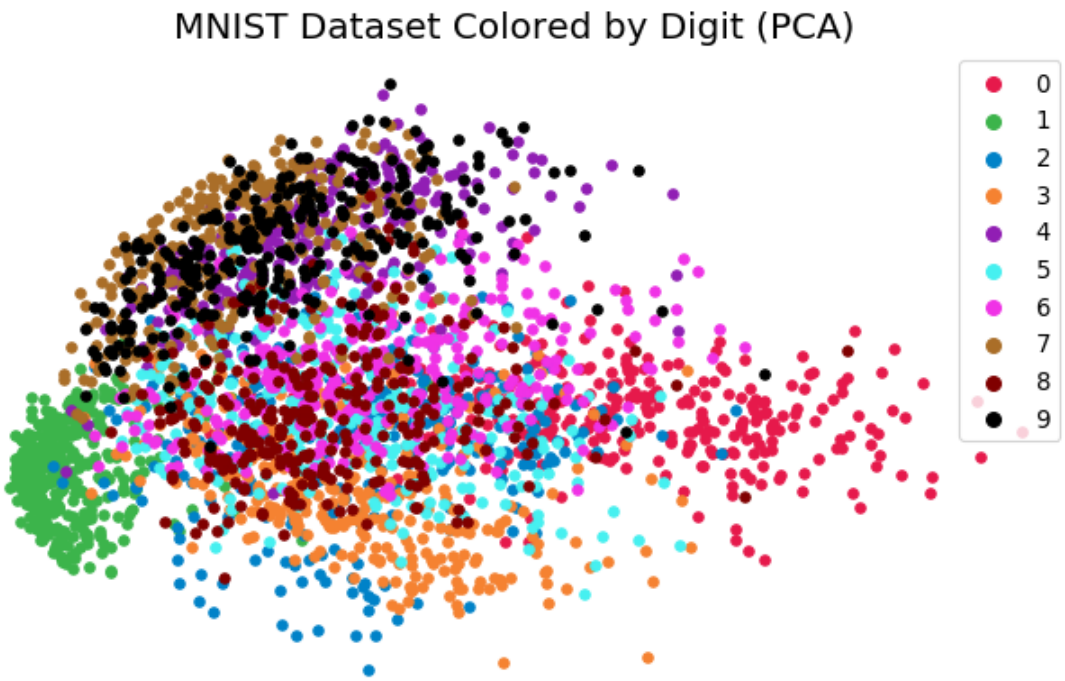


Figure 2.5: PCA reduction on the MNIST dataset

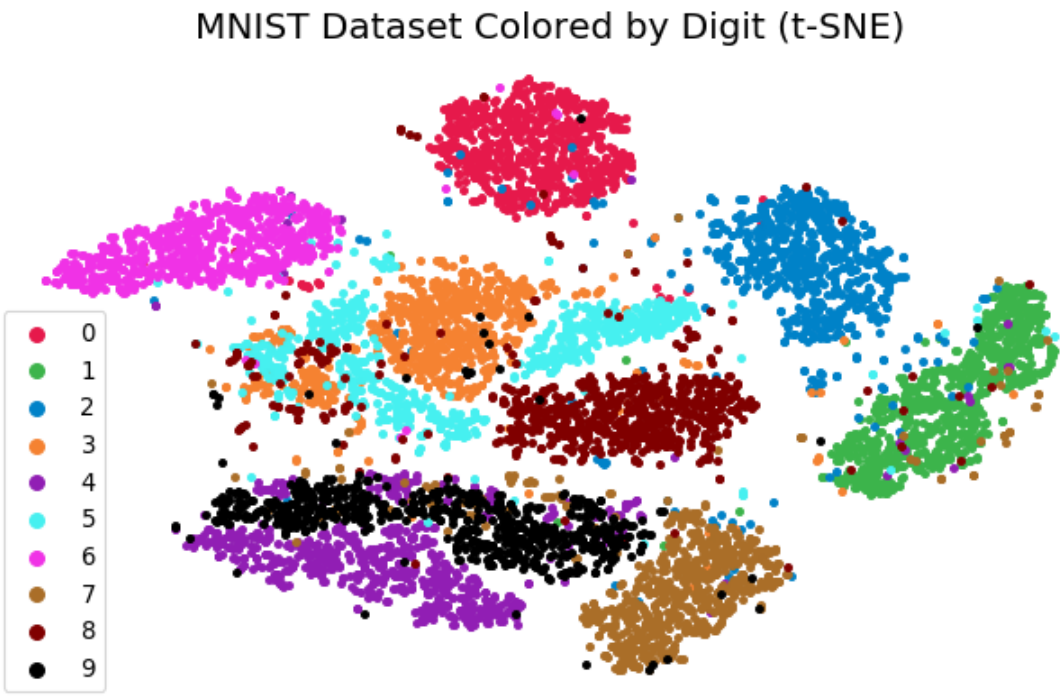


Figure 2.6: t-SNE reduction on the MNIST dataset

Chapter 3

Review of Literature

The purpose of this chapter is to explore the research related to the content of this thesis. We will analyze the viability of widely-used security datasets, and the ways that different machine learning methods have been utilized for intrusion detection.

3.1 Security Datasets

In order to properly develop and test security tools, researchers require meaningful datasets to train models and algorithms. A number of datasets have been produced over the last few decades, many of which are still being used and verified using novel security techniques.

3.1.1 DARPA

The 1999 DARPA dataset is a tcpdump dataset created by DARPA and MIT Lincoln Lab for an intrusion detection competition [7]. It is a modification of the 1998 DARPA dataset, made to include more attacks and an augmented network configuration. The data was collected over a five-week period on a simulated military network. with synthetic attacks running during Week 2.

One of the first notable criticisms of using this dataset for modern research is its age. Being that the dataset is 20 years old, NIDSs developed with DARPA may not be generalizable

to network traffic generated today. The size and complexity of current networks is several degrees removed from the simulated environment used by DARPA. Even as the data was generated, researchers were observing an increase in network pollution [19]. This pollution consists of strange-looking traffic, but is nonetheless benign.

Mahoney [20] creates a detailed analysis of the shortcomings of the DARPA dataset. By developing purposefully inaccurate detection systems, the author is able to demonstrate the existence of artifacts in the data that can skew the results of other detection systems. In his paper, Mahoney discusses the implications of simulated traffic not aligning properly with real-traffic. For example, there are attacks in the test data that contain Time-to-Live values that do not exist in the training data.

3.1.2 KDD'99

Created as part of a competition at the Fifth International Conference on Knowledge Discovery and Data Mining in 1999 [21], the KDD Cup 1999 dataset - commonly referred to as KDD'99 - is one of the most utilized datasets in computer security. A Google Scholar query for 'KDD'99' results in thousands of articles, hundreds of which have been published within the last year.

The features in this dataset were generated using the DARPA tcpdump data. Some, such as `duration` and `protocol_type`, are taken directly from TCP connections. Other features are extracted using domain knowledge of the payloads in the tcpdump files, such as `logged_in`, where a boolean describes if a connection successfully logged into a service. More features were calculated within a two-second time window, particularly error rates and counters. Overall, KDD'99 offers 41 features, including labels for attacks.

Despite the overwhelming popularity of KDD'99, it is not without its weaknesses: specifically

its foundation, DARPA. Because KDD'99 was developed from the DARPA data, flaws found in DARPA could find their into the KDD data. As noted in [8], there is no formal definition for attacks in either DARPA or KDD'99. Probes and scans should only be considered under a specific threshold, which are not defined in either research.

When Portnoy et al. [22] performed a clustering analysis of the KDD'99 dataset, they partitioned the data into 10 equal groups and found that partitions 4-7 contained only one type of attack. Similarly, [23] found that denial-of-service attacks make up over 70% of the testing data. This imbalanced distribution in the dataset makes evaluation methods severely biased towards the most frequent attacks.

3.1.3 NSL-KDD

To combat many of the flaws in the KDD'99 data, researchers in [8] augment the dataset to produce NSL-KDD. In NSL-KDD, there are no redundant records, meaning there will be no frequency bias when training models. This also reduces the the size of the training set by nearly 90%, and the test set by about 33%. For each record, a difficulty level is assigned, and the number of records selected for NSL-KDD is inversely proportional to the percentage of that difficulty within the original dataset. As a result, classification techniques will have a better baseline for comparison during evaluation. The work presented by Tavallaee et al. demonstrates that the NSL-KDD data improves the testing accuracy of a range of anomaly detection algorithms.

Even though model training improved in comparison to the original KDD'99 set, it is worth reminding that the NSL-KDD is still relying on the synthetic DARPA data from 1999. The age and generation method of the dataset detract from the improvements made, as real-world networks are still very different than those generated by DARPA.

3.1.4 CTU'13

García et al. [24] recognized the lack of properly generated datasets, particularly when evaluating the accuracy of botnet detection systems. To that end, they created a dataset known as CTU'13 [24]. Within 13 scenarios, the authors created a network with live traffic, and monitored hosts that were purposefully infected several pieces of malware. NetFlow files were generated through tcpdump captures, and labeled as either *Background*, *Normal*, or *Botnet*. Botnets were labeled through IP addresses, and Normal traffic was determined by certain filters. All other traffic was labeled as Background.

This dataset is a well-purposed effort to create a new baseline for botnet evaluation methods. Including real traffic mixed in with synthetic connections is a significant improvement over fully simulated network such as in DARPA and KDD. However, the lengths of the captures limit the dataset for the purposes of long-term temporal analysis, such as training models across days of traffic to utilize time as a feature in detection.

3.1.5 UNSW-NB15

In order to remedy the weakness of the DARPA-based datasets, another synthetic dataset was generated using similar techniques by Moustafa et al. [25]. Using the PerfectStorm traffic generator by IXIA [26], researchers created tcpdump files, and extracted features with Bro and Argus. The network has 45 IP addresses, with 49 individual features per flow, spanning 41 hours over two days.

Nine different attack families exist in the dataset, including denial-of-service and backdoor attacks. A shortcoming of this dataset is it is entirely synthetic, with attacks being generated at fixed intervals. These fixed intervals could introduce unintentional affects during training of a learning model, making learning using with temporal features difficult. It is also a short

dataset, so we would be unable to generate broader temporal factors like day of the week.

3.1.6 UGR'16

A relatively recent dataset published by Maciá-Fernández et al. shows promise as an extensive NetFlow dataset. Created and designed to be used for cyclostationarity research, the UGR'16 dataset was collected at a tier 3 Internet service and cloud provider in Spain [14]. Data was collected from March-June as a ground-truth set, and from the end of July through August as a test set. It includes 12 of the most common NetFlow features, including IP addresses, ports, and timestamps.

In the test set, the authors generated traffic from denial-of-service attacks, port scans, and botnets using virtual machines that mimic many of the functions seen in the ISP's network. Botnet traffic was modified from Scenario 42 of the CTU'13 dataset [24] to fit the timestamps and IP addresses to the ISP network. Flows are labeled base on the type of attack, or labeled as background. The authors decided to include synthetic attacks so the dataset was not dependent on inconsistent labeling of background data. Relying on purely live traffic would require inspection of every flow to confirm if they are benign or malicious. Instead, we assume that all traffic labeled as 'background' is benign.

The data was analyzed using a series of tools to label attacks that may have occurred in the live capture. First, a combination of blacklists was used to label potentially malicious flows, but it should be noted that these may not all be attacks. Attack signatures were also analyzed and labeled. Using anomaly detection methods developed in [27], other scanning and spam attacks were found through the training and test data. The authors of [14] provide a thorough analysis of the anomalies detected, visualizing the sources and destinations of the scans.

The UGR'16 dataset stands out from other data not only because it is new and untested, but also due to the structure of the data. In total, there are over 230GB of compressed and anonymized NetFlow data that is intended to be used for long-term statistical analysis. This structure is useful for creating NIDSs that utilize time-based features as context in their decision process. These time features could expand to weeks, months, or seasons using this dataset. But it should be noted that due to the novelty of this data, its practical usefulness for machine learning and NIDS research has yet to be explored.

3.2 Machine Learning

Research in the field of intrusion detection has increasingly been exploring the use of machine learning to classify malicious behavior or detect anomalies [28]. There are two ways that we can utilize machine learning to solve a research problem. *Classification* techniques attempt to place connections into bins to define a type of attack. On the other hand, we can use *regression* to predict continuous values rather than classes, such as predicting the number of flows that will occur within a time period.

We can evaluate the effectiveness of an intrusion detections system with several metrics, and [29] argues that the *false alarm* rate is one of the most critical measures. IDSs should try to minimize the number of inputs that are incorrectly labeled as 'anomalous'. By limiting the number of false positives, we can minimize the amount of time an analyst takes to process all alarms. Mahoney and Chan [30] analyzed the results of an intrusion detection competition. They found that limiting the number of alerts to 10 false alarms a day also limits the detection rate to between 40% and 55%.

3.2.1 Clustering

Clustering is a form of unsupervised learning, meaning there are no labels included in the data. The typical methods of clustering include k-means clustering [31], DBSCAN [32], and BIRCH [33]. Because there are no labels, clustering is applied to outlier detection problems. During algorithms such as DBSCAN or k-Nearest Neighbors, we split points into different cluster using distance metrics. Points that are outside of a distance threshold from other points are labeled as outliers. Otherwise, we set a threshold for the number of points in a cluster and attempt to fit cluster labels (k-means). Points within a cluster that are far enough away from the center of the cluster can be labeled as anomalous [34]. We can label clusters by hand, and use those labels for supervised learning problems. This requires domain knowledge of the data being clustered, and can lead to inaccurate training if data is mislabeled.

Intrusion detection research using clustering has revolved around the KDD'99 and NSL-KDD datasets. Iglesias and Zseby [35] use the improved NSL-KDD dataset to analyze the importance of feature selection during k-means clustering. In [36], the authors show that clustering techniques produce high false-positive rates on the NSL-KDD data. Kumar et al. improve on the BIRCH algorithm in [37] by utilizing intra-cluster distances to evaluate cluster quality. Their results show, however, that their clustering technique decrease the classification rate of 'normal' connections in the KDD'99 as the number of clusters increase.

Other efforts to improve the results of k-means clustering on network intrusion data include x-means [38] and y-means [39]. In [38], Ahmed and Mahmood modify the fixed number of clusters, k, in the k-means method to use a range of x values. This allows the clustering on KDD'99 and DARPA to produce more dynamic and effective clusters. Guan et al. use y-means to remove outliers that lie outside a threshold in a k-means cluster, and generate

more clusters from those outliers. They also use the KDD'99 dataset.

While clustering has been researched for use on static data, online learning for clustering algorithms requires different approaches. In order to add a new point to a cluster, most algorithms will recalculate all neighbors and clusters, which can take a significant amount of time for large datasets. This is not feasible for streaming environments like live networks. To this end, researchers extended the capabilities of clustering to operate in an incremental manner. For instance, [40] proposes an incremental clustering algorithm for log data that calculates distance from an input point to all clusters, rather than all neighbors. [41] maintains a history of all previous clusters in order to improve the recalculation and accuracy of future clusters.

3.2.2 Neural Networks

Neural networks are a concept that has been around for several decades [42], but have slowly grown in popularity in recent years due to the increase in computational power and solutions to past problems [43]. Neural networks are a supervised learning model, that train *hidden layers* of functions to predict an output from an input. These functions are saved for later training or for production use. Stacking multiple hidden layers in a neural network is usually referred to as *deep learning*.

Neural networks exist in several different flavors, including a multi-layer perceptron (MLP) [42], recurrent neural networks (RNN, formerly named *Hopfield networks* after creator John Hopfield), and convolutional neural networks (CNN) [44], [45]. Hochreiter and Schmidhuber proposed the long short-term memory (LSTM) [46] to counter the vanishing and exploding gradient problems described in [47].

Most of the intrusion detection systems that are based on neural networks rely on RNNs and

LSTMs. This is due to the fact that we can utilize the sequential nature of the RNN to build temporal context during training. As we input data into an RNN, the network maintains a memory of previous inputs (within a specific limit) to make a decision on the current input. This is useful for intrusion detection so that our systems can take information from previous flows to determine the intent of the input flow.

In 1999, Ghosh et al. proposed using Elman networks (a style of RNN) for modeling anomaly behavior in the DARPA dataset [48]. The authors were able to detect 77.3% of intrusions with no false positives. They showed an improvement of this model over an MLP and a signature-based table lookup.

Staudenmeyer uses the KDD'99 dataset in [49] to evaluate the effectiveness of an RNN-LSTM for intrusion detection. He finds that while 60% of networks trained had performance better than random guessing (meaning that prediction rate was 50% or better), the models trained did considerably better than the solutions to the original KDD competition. Kim et al. ran a similar evaluation in [50] using KDD'99 and an RNN-LSTM. Their model averaged a false alarm rate of 10%, which is considerably higher than would be usable in practice. We can review [51] for an analysis of the use of different optimizers on an IDS trained on KDD'99. Optimizers are used to calculate how to adjust the internal functions of neural networks, and have varying degrees of speeds and accuracy.

Yin et al. evaluated using an RNN as a NIDS on the NSL-KDD dataset in [52]. They compared different machine learning models including random forests and MLP, and demonstrate that the RNN performs better than all other models in their experiment. Using the CTU'13 dataset to analyze the effectiveness of their models, Torres et al. use behavioral models to encode their date to feed into an RNN [53]. Their behavioral model uses a combination of the continuous and categorical features to encode the data into a 50-dimensional vector for input into the RNN.

3.2.3 Decision Trees

Another type of supervised learning method is a *decision tree*. Decision trees learn a series of decisions that partition data into the labeled classes. Because these decisions are based on the nature of the data and not mathematical functions of the data, we can interpret a tree to understand why inputs give a particular output. We can combine trees using *ensembling* methods to potentially improve performances.

Decision trees algorithms started with ID3 [54], which was later improved with the C4.5 tree [55]. The C4.5 tree was only usable for classification problems, which prevent it from being used for regression problems. Therefore, the algorithm was again improved, and became known as Classification and Regression Tree (CART) [56].

C4.5 and CART have been applied to intrusion detection in several works. In [57], the authors compared the C4.5 algorithm to a Support Vector Machine (SVM) , another standard machine learning algorithm. They demonstrate high accuracy for C4.5 in the KDD99 problem. However, not much is given about other metrics of effectiveness, such as false alarm rate, which is important when considering alerts for analysts. Similarly, [58] use C4.5 as an algorithm to test their feature selection methods, showing high accuracy on KDD99, but do not provide the other important metrics. Many other works [59]–[61], utilize a C4.5 tree for intrusion detection research, though [60] uses live network data to train their model. Revathi and Malathi evaluate several machine learning algorithms such as SVM and random forests in [62], where we can see that the CART algorithms is severely outperformed by random forests, a bagged tree approach. *Bagging* collects the outputs from many deep trees and combines them using averaging or majority vote.

Random forests has been shown as an effective method of classification for intrusion detection systems. Resende and Drummond perform an excellent survey of the use of random forests

for intrusion detection in [63], evaluating the performance metrics and content of numerous works that utilize random forests. [64] tests the use of different sized forests using the NSL-KDD dataset. Because a random forest is a combination of decision from many trees, the number of trees (with different structures) has a noticeable effect on accuracy in this work. Hota and Shrivas compare random forests with a simple neural network and C4.5 decision tree using the NSL-KDD dataset, with better performance across all metrics, including true and false positive rates [65]. In [66], we can see the important of feature selection for random forests. Using Synthetic Minority Oversampling Technique (SMOTE) , random forests performs better with 19 out of 41 features in NSL-KDD, but best with 22 out of 41 features.

A more recent approach to decision trees was introduced with XGBoost [67], a *boosting* method of ensembling decision trees. Rather than combining the decisions of multiple large trees, XGBoost trains many trees with the error of the previous tree. While this sequential process takes time, it is improved by allowing the parallelized training of a single tree. XGBoost has been used recently for many Kaggle competitions, but has not been evaluated well for network intrusion detection problems. [68] analyzes the algorithm on the NSL-KDD dataset. They provide a thorough discussion on their methodology for tuning the model and the results of their training, but do not use the NSL-KDD dataset as provided. Instead, the authors combine the training and test sets, which does not provide a means of testing how well the model will operate on unseen data. XGBoost has been utilized for a thorough analysis of a malware Kaggle competition [69], and was shown to perform well for this problem. The authors extracted many features from hex dumps and disassembled malware samples, and argue that while they did not win the competition, the tradeoff between the complexity of their model and their performance sets them apart from the winner.

3.2.4 Categorical Data Representation

In Section 2.1.2, we discussed encoding and embedding, and how we can use them to convert the categorical representation of TCP ports into a continuous value that machine learning algorithms can understand. Works such as [70] show the use of binned one-hot encoding for representing ports for neural networks.

In [15], Mikolov et al. proposed a novel word embedding scheme called 'Word2Vec.' Word2Vec includes additional functionality over a basic word embedding scheme including (1) subsampling of frequent words to prevent overfitting, and (2) negative sampling on the weights that are updated after each round of training. Word2Vec has shown to be useful during Natural Language Processing (NLP), particularly for sentiment analysis [71] and text classification [72].

The Word2Vec algorithm was adapted by Ring et al. in [13] to apply to network flows, aptly named IP2Vec. Using a similar embedding scheme, the authors were able to create embeddings for IP addresses and protocols in Scenario 50 in the CTU'13 dataset. By running the DBSCAN clustering algorithm on a subnet of the IP addresses, the authors demonstrated the ability to cluster botnet hosts together. This embedding was shown as a useful way to visualize the similarities behaviors between hosts. Because the embeddings were trained on a vocabulary of IP addresses, ports, and protocols, the embeddings for IP addresses contain contexts of what hosts a machine was communicating, and the services offered and accessed by these hosts. This offers a more efficient and richer means of converting the categorical data of a flow record into continuous values that machine learning algorithms can understand.

Chapter 4

Experimental Design

Current machine learning research for network security revolves around the use of aging and synthetic datasets such as KDD'99 [21]. While these works are relevant in demonstrating the potential of these techniques, many datasets fail to include network features such as IP and port. These features are readily available for any network with a NetFlow capture, and contain considerable amounts of information regarding the relationships and behaviors of hosts seen on a network.

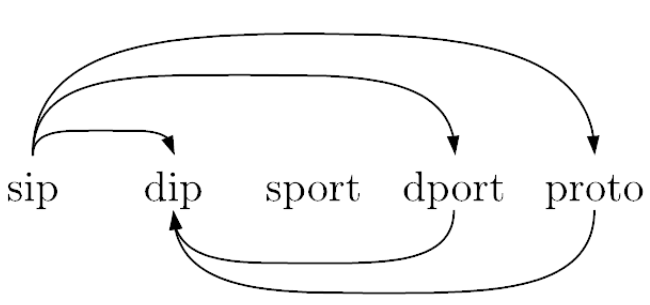
NetFlow features are available in a dataset like CTU'13 [24]; however, they are difficult to convert into continuous values for input into machine learning models. IP addresses are categorical representations of 32-bits of information, and while those 32-bits are capable of being used as integers, they do not reveal any information about the behavior of the host they are tied to. When determining similarities between IP addresses, models will calculate that two addresses are similar if they have similar numerical values, such as 192.168.1.1 and 192.168.1.2. These addresses are only one integer or one bit distant from one another, making them very similar in some machine learning models, but may exhibit entirely different behaviors.

4.1 Binned IP2Vec

Instead, we choose to capture similarities in IP addresses based on their behavior in our network. We achieve this using the technique developed by Ring et al., named IP2Vec [13]. IP2Vec creates an embedding for IP addresses and ports in a NetFlow dataset, and learns network behavior by predicting NetFlow features given other NetFlow features. The technique is similar to that of Word2Vec, but uses a custom method of determining input/output pairs. We design and implement a modified version of Ring's IP2Vec, described below.

4.1.1 Choosing Word Pairs

Word2Vec [15] utilizes continuous bag-of-words or skipgram (2.2) to determine the pairs of words that will be used to train the embedding. In calculating word pairs, Word2Vec will slide a window across each sentence in a document to decide the context around a given word. The same technique is not be applicable when analyzing a network flow, as we are not limited in reading a NetFlow record from left to right. When reading a flow record, an analyst can infer information about the transaction given multiple associations of features, but not necessarily all combinations of features.



input	output
sip	dip
sip	dport
sip	proto
dport	dip
proto	dip

Figure 4.1: Choosing word pairs in IP2Vec

Given a unidirectional NetFlow record like the one shown in Figure 4.1, we can extract several pairs of inputs and outputs to train our embedding. As we decide these pairs, we can ask several questions about what information we are trying to learn in order to choose word pairs:

- What machines does the Source IP talk to?
  Source IP to *Destination IP*
- Which services does the Source IP request?
  Source IP to *Destination Port*
- Which protocols does the Source IP use to communicate?
  Source IP to *Protocol*
- What machines host a given service?
  Destination Port to *Destination IP*
- What machines operate using a given protocol?
  Protocol to *Destination IP*

In the 5-tuple flow, the only feature not being used for learning an embedding is the Source Port. We choose to omit this feature in our embedding because we are learning with unidirectional flows. If we were to use Source Port as an input or output in our embedding process, we would be attempting to learn unidirectional flows as bidirectional. In other words, given only one side of a conversation, our embedding would attempt to infer information about the other side of a conversation (which may not even exist in the cases of unidirectional protocols like UDP). Instead, we expect to learn the behavior of a response from a second flow record, and calculate the input/output pairs with the same process.

4.1.2 Embedding Model Design

After calculating word pairs, each pair is used as the input and target for a single layer neural network. Figure 4.2 describes the behavior during training. An embedding layer is used as a lookup table between each 'word' and its respective floating-point embedding; mathematically, we are multiplying a one-hot encoding vector by a matrix, accessing the row in the matrix that represents our word. The row is then passed through a linear layer, the output of which is put into a final activation function. We use a logsigmoid function to squash the output from the linear layer to between 0 and 1.

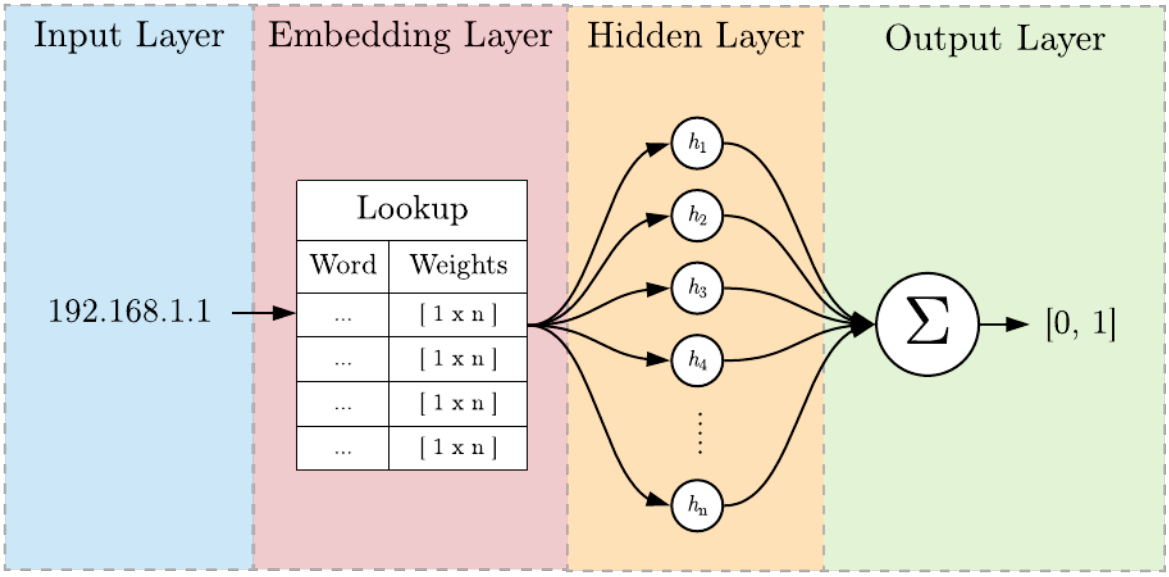


Figure 4.2: Network design for IP2Vec

Negative Sampling

Negative sampling is a loss function described by Mikolov et al. that is used to reduce the effort needed to update embeddings in a large dataset [15]. Instead of updating all weights

in a matrix during backpropagation, negative sampling allows a select set of weights to be updated after each input. For each input, the row containing the embedding for the input is moved closer to the output, and more distant from a configurable `neg_samples` rows of other data.

We choose to use negative sampling as it changes a multiclass classification problem into a binary classification problem. Rather than classifying an input into one of many different classes, where the number of possible classes is the size of our vocabulary, the embedding learns similarity between input and output. In other words, the binary problem allows our model to determine if an input/output pair is similar (1) or not (0). Multiclass classification become difficult

The negative samples are chosen at random from the vocabulary. In designing Word2Vec, Mikolov et al. [15] stated that their models chose negative samples using a modified unigram distribution:

$$P(w_i) = \frac{f(w_i)^{\frac{3}{4}}}{\sum_{n=0}^{N} f(w_n)}$$

This distribution works well in natural language processing problems, as words with high frequencies like 'and', 'a', and 'the' are removed from the vocabulary because they do not provide a significant amount of context. However, in NetFlow, removing the most common features would cause us to lose information about important parts of our network. For instance, removing highly frequency words in our NetFlow vocabulary would require removing words like TCP, UDP, HTTP/S, etc. But choosing from a unigram distribution would push the model to primarily select those common words as negative samples, rather than selecting from a smoother distribution of the average frequency words.

Instead, we sample randomly from a uniform distribution for our negative samples, such

that each word has an equal probability of being selected. In our testing, this gave better results compared to both a unigram distribution and the distribution suggested by Mikolov. In doing so, we choose a middle ground between selecting the most frequent words too often, and removing those words entirely.

IP Address Binning

To improve the training time of IP2Vec, we propose binning IP addresses that do not meet some criteria. For our purposes, we choose to bin all IP addresses that are seen in less than `min_samples` flows. This reduces the size of the dataset by recognizing the lack of information that we can train on when there are a minimal number of flows for an IP address. This bin is defined as an 'OTHER' word in the vocabulary.

While this design choice may lead to a lack of context regarding how machines infrequently operate with other machines, we believe it is appropriate for the embedding to learn how each IP address interacts with a generalized 'unknown' category for other machines. Rather than attempting to extract many individual definitions from very little information, we expect the sum of the information from all flows to be more effective in producing meaningful embeddings for the common words in the NetFlow vocabulary.

Hyperparameters

Other tunable parameters for the neural network are *embedding dimension*, *batch size*, and *epochs*. The embedding dimension is the size of the vector that our embedding will output for each word. Google developers working on the TensorFlow library recommend using the 4th root of the vocabulary size [73], and the original researchers working on IP2Vec chose 32 as their embedding size. This value is adjusted given the dataset we work with.

The batch size determines the number of word pairs that is trained on before backpropagation, while the number of epochs is the number of times the entire dataset is trained over. A combination of these two hyperparameters affects both the training time and the accuracy of the model. A smaller batch size allows fewer word pairs to be affected each time the model backpropagates, meaning the embeddings for those words are updated more precisely, rather than being influenced by the error from other words. However, as batch size decreases, training time is significantly increased as the model will need to update weights more frequently.

With more epochs comes more opportunities to update weights, but introduces overfitting to our model. Overfitting reduces the embedding's ability to generalize relationships, and will cause the information contained in the embedding to be less flexible to less common relationships. More epochs also means more training time, so for our purposes, we choose to train initial models for 10 epochs, as was done in the original IP2Vec work [13].

Chapter 5

Evaluation

In this chapter, we describe the results of our evaluation of IP2Vec. The embeddings and supervised learning models were trained on a machine with the following specs and packages:

Tool	Version	Package	Version
OS	Ubuntu 16.04.5 LTS	pandas	0.23.0
Python	3.6.5	numpy	1.14.3
Server	Dell PowerEdge R630	scikit-learn	0.19.1
CPU	2x Xeon E5-2670	pytorch	0.4.1
RAM	72GB DDR3	xgboost	0.80

Table 5.1: Hardware-Software Configuration

5.1 Binned IP2Vec

We use the PyTorch modules for Python to implement our embedding with same techniques found in [13]. However, in order to prepare for larger datasets, we include a means to bin uncommon or unknown IP addresses. This value is referred to as the `min_samples` for our vocabulary.

We train embeddings for both the CTU'13 and UGR'16 datasets, and evaluate the quality of the embeddings through dimension reduction to a 2D space. The dimensionality reduction is done through t-Distributed Stochastic Neighbor Embedding (t-SNE) [17], where we attempt

to map an n-dimensional vector into an m-dimensional space. This process is discussed in Section 2.3.2.

5.1.1 CTU'13

The first dataset we embed is the CTU'13 Scenario 50 dataset. This data includes $\approx$ 6 hours of NetFlow traffic, with 10 unique IP addresses infected with the Neris botnet, which spams HTTP and SMTP traffic to other addresses. Botnet traffic is an undemanding test case for a technique to capture host behavior, as hosts infected with the same botnet malware tend to behave in similar ways [74]. The hosts are distinguishable from other hosts due to their repeated and cyclical behavior, particularly if the infected hosts behavior deviates from the hosts normal behavior. Therefore, for each of the 10 infected hosts, we look to generate embeddings that are more similar to one another than to embeddings of other IP addresses in the dataset.

We compare the embedding with several values for `min_samples`, to evaluate what effect binning IP addresses has on the embedding. For this dataset, we choose `min_samples={1,2,5}`. A value of 1 means that the vocabulary contains all IP addresses, regardless of the number of times the IP address is seen in the dataset. We also test the embedding when the set of IP addresses seen only once are binned together into a single word (`min_samples=2`). Such a value is useful for conserving space in the vocabulary by not learning about IP addresses that may be related to simple reconnaissance or web crawlers. Instead, the binned word is used to store a general definition for this category of hosts. A larger bin of 5 is used to determine the effects of the binning process on a broader and more substantial set of the data.

The graphing pipeline is shown in Figure 5.1. We condense a vocabulary of network features

into an embedding using IP2Vec, and transform those embeddings using t-SNE to create `x` and `y` coordinates. Difficulties in interpretation become apparent after transforming non-linear embeddings into a linear, 2D plane. Linear analysis techniques such as clustering does not reveal significant information about the nature of the relationships made in the embedding.

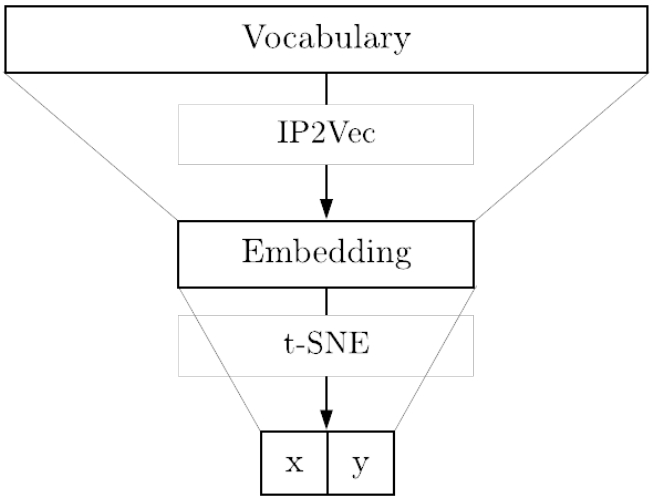
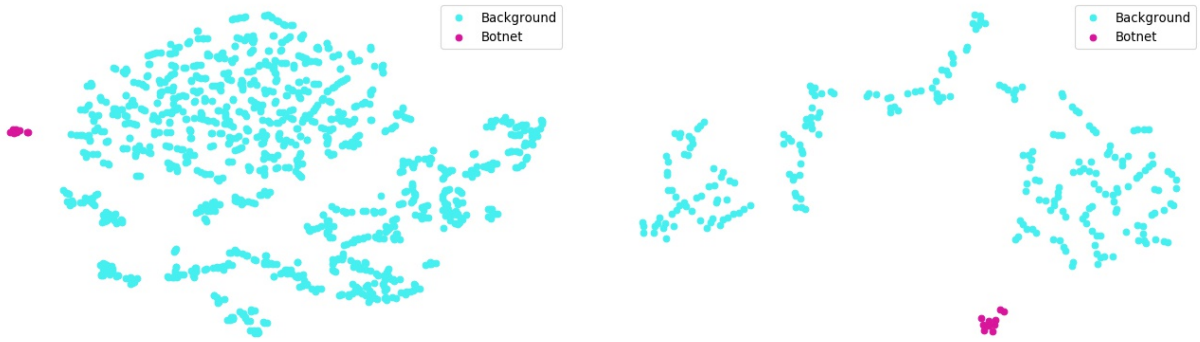


Figure 5.1: Graphing pipeline for IP2Vec

Ring et al. [13] uses DBSCAN [32] to cluster a t-SNE reduced embedding. This provides a quantitative analysis of whether clusters can be separated and labeled into their appropriate classes. Using DBSCAN, Ring et al. calculated the *accuracy* (whether a class was accurately clustered), *homogeneity* (whether a cluster contains only one class), and *completeness* (whether all points of a class are in the same cluster). However, these metrics are difficult to accurately interpret through the t-SNE embedding due to the many hyperparameters that effect the final output t-SNE. To that end, we choose to qualitatively analyze the t-SNE embeddings, as the global structure of the data does not provide meaningful information, particularly the metrics calculated by DBSCAN.

The first embedding produced is used the verify the ability of IP2Vec to capture similarity between machines with a known behavior. In the CTU'13 dataset, there are 10 IP addresses of machines known to be infected with the Neris botnet. As these machines spam HTTP and SMTP traffic in a similar manner, sharing network behavior, we expect a t-SNE representation of the embedding to graph these IP addresses close to one another. t-SNE is run using cosine similarity metric, as it is the same similarity metric used in Word2Vec [15]. Cosine similarity takes into account the direction and magnitude of vectors to determine a $[0, 1]$ range of likeness between vectors.



(a) /16 subnet (b) /24 subnet

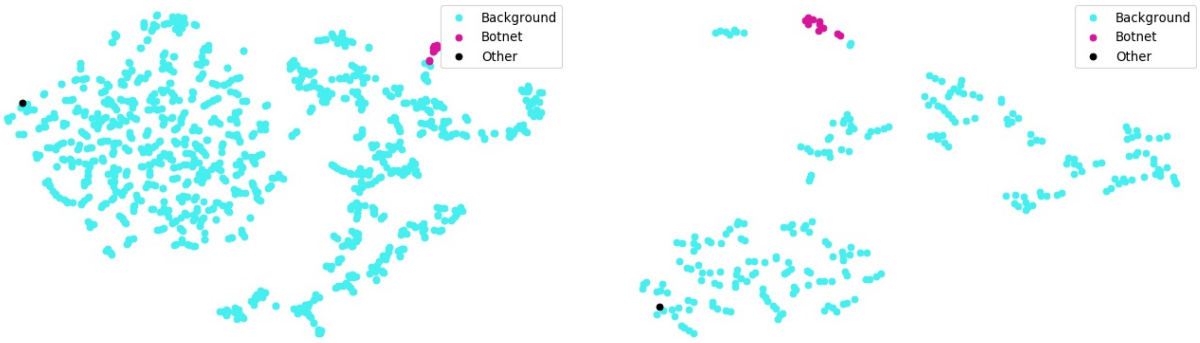
Figure 5.2: 2D t-SNE of 32-dimensional IP2Vec embedding with *min_samples* = 1

In Figure 5.2, we train an embedding with IP2Vec without binning. Each IP address in the CTU'13 dataset has a word in the vocabulary, regardless of its frequency. This produces a vocabulary size of 431,845 words, which includes IP addresses, ports, and protocols. We graph IP addresses within the 147.32.1.1/16 subnet (1181 addresses), as well as the 147.32.84.1/24 subnet (256 addresses), as these are subnets that contain the botnet. As seen in the graphs above, the infected hosts (red) are clustered together, meaning t-SNE is able to clearly learn similarity between the embeddings of the hosts. Additionally, the dissimilarity between the botnet hosts and other machines on the network is apparent in the relative distance between the cluster of infected hosts and other points in the subnet.

(a) /16 subnet (b) /24 subnet

Figure 5.3: 2D t-SNE of 32-dimensional IP2Vec embedding with *min_samples* = 2

Reducing the vocabulary of IP2Vec with the proposed `min_samples=2` technique reveals similar results after reduction through t-SNE. By choosing `min_samples=2`, the embedding trains all IP addresses that are only ever seen in one flow into a single vocabulary word, shown as "Other" in the figures. This value is intended to capture flows of common network traffic from simple scans, web crawling, or benign flows that are never repeated. This reduces the vocabulary size from 431,845 to 417,452 (96.6%). As seen in Figure 5.3, the infected hosts are clustered similarly as in `min_samples = 1`.



(a) /16 subnet (b) /24 subnet

Figure 5.4: 2D t-SNE of 32-dimensional IP2Vec embedding with *min_samples* = 5

To exaggerate the effects of binning infrequent IP addresses, we train a separate IP2vec

embedding using `min_samples=5`. In requiring an IP to be seen in more flows, we reduce the vocabulary down to 209,632 words (48.5%). In doing so, we see the similarities between the botnet hosts and other IP addresses in the dataset, as the botnet cluster is closer to the bulk of the dataset. We can also see several points being graphed in the vicinity of the botnet cluster. This shows that IP2Vec loses the ability to learn the unique behavior of the botnet when drastically reducing the size of the vocabulary. Instead, the botnet vocabulary becomes similar to other benign behavior. This can happen due to sharing frequent destination addresses or services, such as DNS servers, as well as the relationship between these points and the "Other" class.

The overall statistics for training the three IP2Vec embeddings is shown in Table 5.2.

	`ms=1`	`ms=2`	`ms=5`
Vocabulary Size	431,845	417,452	209,632
Vocabulary %	100.0	96.6	48.5
Training Time (s)	5402	5414	5394
/16 KL-Divergence	1.317425	1.281992	1.325652
/24 KL-Divergence	0.713150	0.714128	0.731464

Table 5.2: Training Statistics for IP2Vec on CTU'13

5.1.2 UGR'16

Next, we embed the network information on a larger network than in CTU'13, using the UGR'16 dataset [14]. This dataset contains gigabytes of NetFlow data from an Internet and cloud service provider, making it a desirable target to analyze host similarities due to the heterogeneous nature of the network traffic. However, it is not computationally feasible to train an IP2Vec embedding on the entire dataset.

We choose a single day, August 1, as a target for our embedding, to narrow the focus of the embedding. This produces about 135 million unidirectional NetFlow records. Because we

choose to train our models on CPU only (with no GPU acceleration), we choose to narrow the dataset again into a single subnet: 42.219.152.1/21. This subnet was chosen because it resulted in a moderately smaller dataset, 87 million flows.

For the IP2Vec embedding, we use only background flows in the August 1 subnet. All attacks are removed from the data, as this embedding is used to represent a network at its normal state. Training an embedding with the attacks will leak information about attacks from the embedding into future models, overfitting any learning done with IP2Vec.

We take care to analyze the performance of the model as it is being trained. This done by monitoring the negative log-likelihood (NLL) loss after each batch of the training process. The average loss should decrease over time, as shown in Figure LOSS. The figure shows a rolling average of the NLL loss for an embedding with `min_samples = 1`. It should be noted that due to the size of the dataset, we choose to decrease the number of epochs from 10 to 5. This also helps prevent overfitting, as continuing to train after the loss has approached its minimum will cause the embedding to loss the ability to generalize to the less frequent relationships in the dataset. As seen in Figure 5.5, the loss decreases significantly in the first 2 million batches, but loses momentum slowly afterwards.

For this dataset, we choose to continue using a 32-dimensional embedding. There are 817,439 unique words in the data; using the Google suggested rule of thumb [73], taking the 4th root of the vocabulary gives ≈ 30, but we round to 32 to use a power of 2.

Because attacks are removed from the dataset, we analyze the August 1 data for the most active web servers and clients to evaluate the effectiveness of the embedding. We are still limited by the abilities of t-SNE to reduce the embeddings into a 2D plane, so complex relationships are difficult to infer from the graphs. We expect web servers to be close together in the graph, while clients with a more dynamic behavior ought to be more widespread.

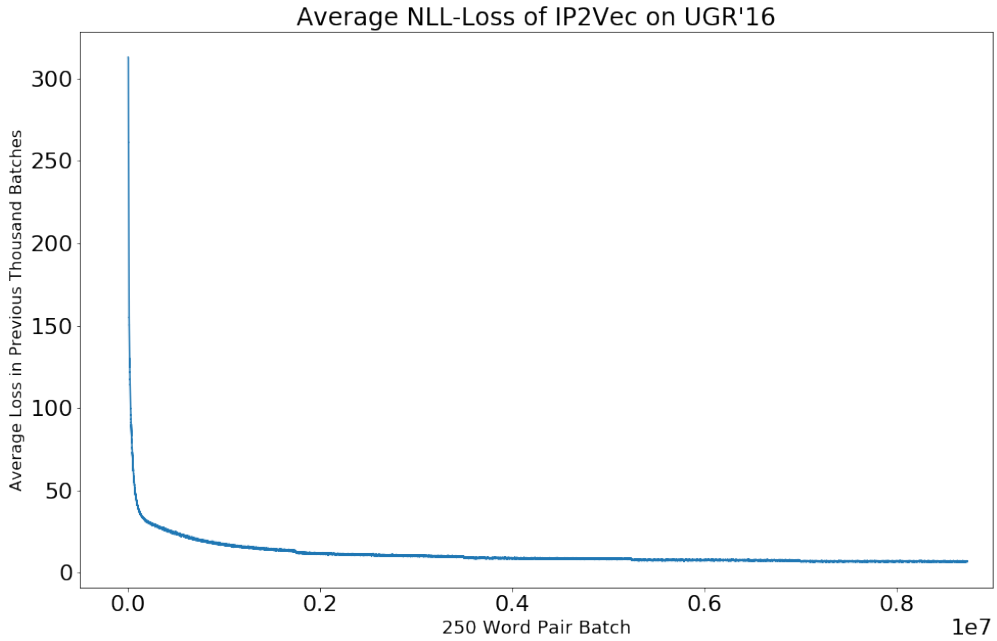


Figure 5.5: Rolling average loss of IP2Vec on UGR'16

We choose four web servers that are primarily active on ports 80 and 443, and which often communicate to ephemeral ports (49152–65535). These are represented by the red squares in Figures 5.6 and 5.7. We also choose 20 IP addresses that exhibit client-like behavior. This hosts tend to use ephemeral source ports, and have a low number of unique destination IP addresses. The clients are represented as red circles in the t-SNE graphs. Due to the structure of the network in the UGR'16 data, the IP addresses chosen may exist behind a NAT; however, we choose to interpret each IP addresses as a single host. As seen in Table 5.4, several IP addresses were chosen even though they had nearly equal numbers of unique source and destination ports. These were chosen to round out the group to 20, as we needed many active connections throughout the day to establish strong embeddings.

Figure 5.6 shows UGR'16, embedded and reduced, with `min_samples=1`. As expected, the web servers exist within the same cluster. The clients are also clustered in several positions, showing that different groups of clients can be represented as similar in the 2D plane.

Src IP	Common Src Ports	# Src Port	# Dst Port
42.219.156.211	80, 443	190	64515
42.219.155.56	80	4	64119
42.219.155.28	443, 80	56	62665
42.219.158.156	443, 22, 80	7370	59406

Table 5.3: Server statistics for UGR'16

Src IP	# Src Port	# Dst Port
42.219.157.222	49399	17
42.219.156.198	20322	2525
42.219.155.11	18190	9
42.219.156.197	16366	201
42.219.156.196	15760	245
42.219.156.194	15384	335
42.219.156.199	15373	190
42.219.153.154	14919	1911
42.219.157.58	13310	1345
42.219.153.149	12859	125

Src IP	# Src Port	# Dst Port
42.219.157.8	12284	149
42.219.153.8	9897	732
42.219.153.234	9489	138
42.219.155.113	8882	8771
42.219.157.145	8829	8909
42.219.157.6	7469	1496
42.219.155.131	6070	11
42.219.157.12	4367	421
42.219.159.182	1345	209
42.219.158.181	1219	337

Table 5.4: Client statistics for UGR'16

TODO: TALK ABOUT WHY EACH OF THE CLIENT CLUSTERS EXIST AS THEY DO.

Next, we train another embedding for UGR'16 with `min_samples=2`. This value is determined in the same way as with CTU'13: binning words that are only ever seen once gives a representation for machines such as simple probes or web crawlers. Using this value reduces the size of the vocabulary from 817,439 to 724,220 (88.6%).

As seen in Figure 5.7, the "Other" point is graphed as more related to the servers than the clients. This is due to number of times that the word "Other" is used to predict one of the server IP addresses, and vice versa. This situation is more common than "Other" being used to predict the same port; therefore, the embedding for "Other" will be updated to be more similar to the embeddings of the servers. This demonstrates that interpreting the t-SNE

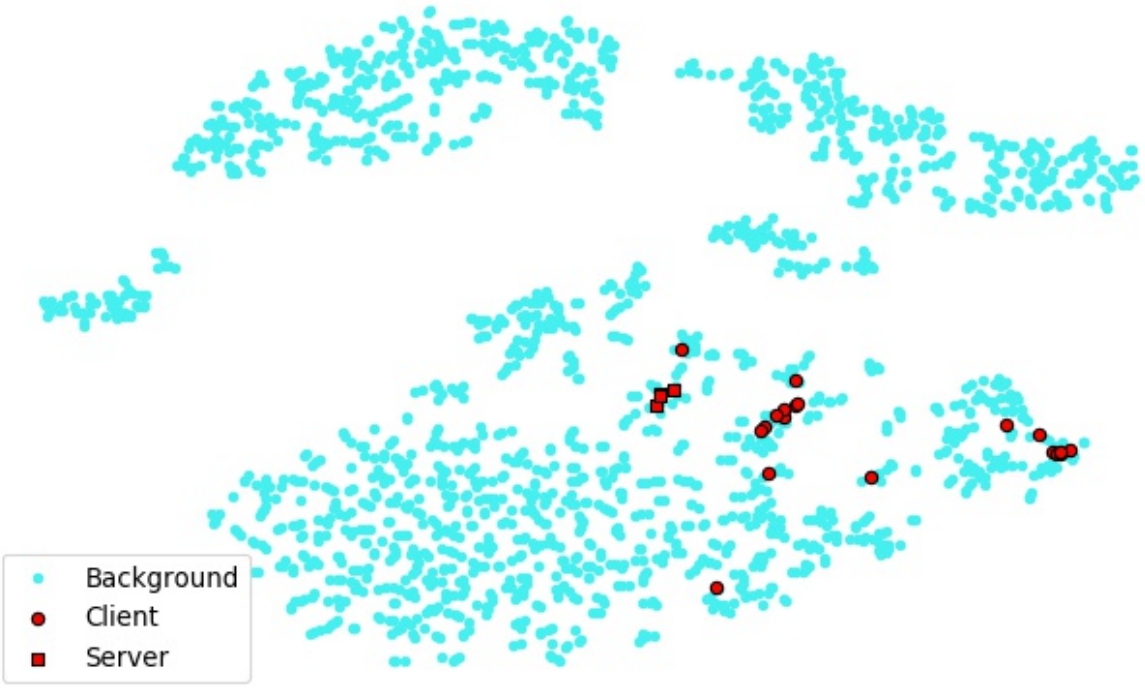


Figure 5.6: t-SNE reduction of 32-dimensional IP2Vec embedding on UGR'16, with `min_samples = 1`

graphs reveals more about the SIP/DIP word pairs than the effects of ports or protocols on the embeddings of the IP addresses.

The overall statistics for training the two IP2Vec embeddings for UGR'16 is shown in Table 5.5.

	`ms=1`	`ms=2`
Vocabulary Size	817,439	724,220
Vocabulary %	100.0	88.6
Training Time (s)	17549	17619
/21 KL-Divergence	1.351012	1.359721

Table 5.5: Training Statistics for IP2Vec on UGR'16

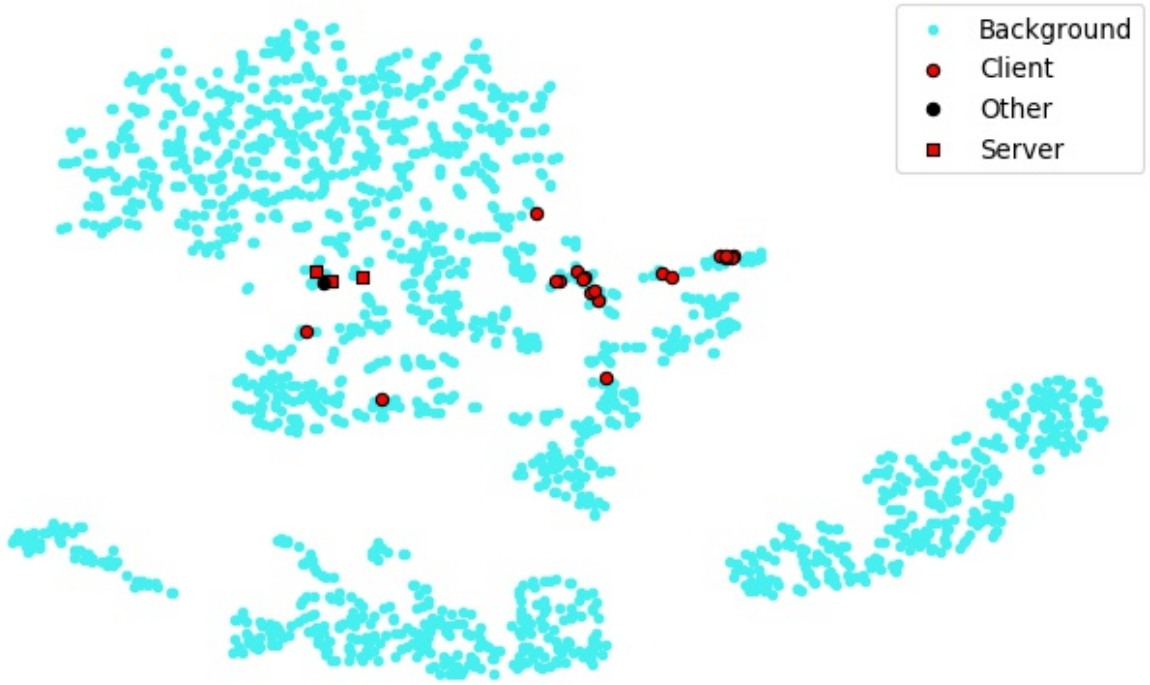


Figure 5.7: t-SNE reduction of 32-dimensional IP2Vec embedding on UGR'16, with `min_samples = 2`

5.2 Intrusion Detection

In order to evaluate the effectiveness of IP2Vec in detecting intrusions in a network, we embed the network information in UGR'16 using the binned IP2Vec model trained previously, and define a multiclassification problem in the dataset. As described in [14], there are 6 unique labels for attacks that were generated from virtual machines on the network: `dos11`, `dos53s`, `dos53a`, `scan11`, `scan44`, and `nerisbotnet`. Including the `background` label, there are 7 unique classes that can be assigned to a datapoint. Because we are given class labels, we are able to use supervised learning techniques to train models to predict those classes.

We look to show that the addition of the IP2Vec data will improve the classification performance of several supervised learning models: Random Forests, XGBoost, and a single-layer

Multi-Layer Perceptron (MLP). Each model uses the F-beta scoring metric to compare performances, instead of metrics such as accuracy. In doing so, we can decrease the effect of class imbalance in the dataset, and determine the models' abilities to classify attacks, rather than their abilities to correctly classify negative classes (in our case, `background`).

5.2.1 Data Engineering

Because of the limitations of training on CPU, as well as a large class imbalance, we take a subnet of traffic from August 2nd. This is the same subnet used to extract data for IP2Vec: 43.219.152.1/21. However, this still leaves over 80 million flows, which becomes an issue when training many models to tune hyperparameters. To that end, we take a time slice of eight hours on August 2nd, from 8am-4pm, to bring the number of background flows to 30 million. We choose this timespan with the assumption that attacks would occur within a normal work day, to disrupt the operation of services offered in the ISP network. Attacks occurring in time periods with lower traffic throughput are not analyzed.

The UGR'16 dataset is structured such that each day has two two-hour of periods of attack, spaced 12 hours apart. The first period is 'planned', meaning each attack type is scheduled for a particular time within the two hours of attack. The second period of attacks is labeled 'random', where each attack is given a random time to begin, with the possibility of overlapping with other attacks. For our evaluation, we choose the 'planned' period of attacks to inject into our eight-hour timeslice. This is primarily due to the absence of the botnet traffic in 'random', as well as the desire for clear periods of time when attacks occur.

For each two-hour time period in the eight-hour timeslice, we inject the attacks from the corresponding day that contains attacks in that time period. Table 5.6 describes the day and time periods from which the attacks are extracted. We combine different days of data with

the assumption that the attacks have no effect on the behavior of the rest of the network [14]. The combination of eight hours of attacks and eight hours of background data from August 2nd results in 37 million flows.

Day	Time
Mon., 08/01/2016	8am-10am
Tues., 08/02/2016	10am-12pm
Wed., 08/03/2016	12pm-2pm
Thu., 08/04/2016	2pm-4pm

Table 5.6: Period of attacks chosen from days in UGR'16 [14]

Virtual Machine Substitution

In the UGR'16 dataset, each of the attackers and victims are virtual machines operated by the authors, running similar resources as the rest of the network, such as HTTP and FTP servers [14]. However, an analysis of the data reveals that there is no interaction between the virtual machines and the outside network. This makes evaluating IP2Vec difficult, as there is no embedding for the victims that does not contain attack traffic.

To remedy this, we choose four servers to replace the four victims of the network-based attacks like `dos11` and `scan44`. By analyzing the dataset to determine a profile for each of the servers, we can modify the dataset to mimic the correct responses that the servers would give during each of the attacks.

We also choose 20 machines that exhibit client-like behavior, as described in Section 5.1.2, to replace the IP addresses of the hosts infected with the Neris botnet. No modifications of responses is needed, as we assume the malware causes similar behavior regardless of IP address.

Time Window Expansion

With attacks added and modified to the eight-hour timeslice, we unfold every 100 flows into a single data point, to create 371,770 100-flow datapoints for classification. This changes the classification from 'Does a single flow belong to an attack?' to 'Is an attack occurring in this period of traffic?'. We choose 100 as the size of the window as an analysis of the August 1st data shows an average of 101 flows per 100 milliseconds. A window is labeled `background` if it contains no attacks in the 100 flows; otherwise, it is labeled as the attack that the window contains.

5.2.2 Feature Space

We first train a baseline for each learning method we evaluate, using the set of features described in Table 5.7. After the data has been expanded using the sliding window, each row in the unembedded dataset contains 1703 features; 17 features taken from each of the 100 flows in the window, and 3 aggregate features extracted from continuous values in the original flows. We choose to one-hot encode the protocols instead of using the embedded values, as the number of unique protocols is small enough to encode. The embedding could provide needed information, but also increases the size of our features space by another 3200 features.

The second feature space we evaluate includes the embedded network features: `sip`, `dip`, `sport`, `dport`, as seen in Table 5.8. Each feature is embedded into 32 dimensions using the IP2Vec model trained in Section 5.1.2. This means that each flow is given an additional 128 columns of information, and each window input is given 12800 new features, for a total of 14503 features when combined with the unembedded values.

Feature Name	Type	Feature Name	Type
proto_TCP	int	rst	int
proto_UDP	int	syn	int
proto_ICMP	int	ack	int
proto_GRE	int	urg	int
proto_IPIP	int	fin	int
proto_ESP	int	psh	int
tos	int	forward	int
dur	float32	span*	float32
pkts	float32	avg_pkts*	float32
bytes	float32	avg_bytes*	float32

Table 5.7: Nonembedded features for supervised learning
* denotes extracted features, calculated per window.

Feature Name	Type
sip_{0-31}	float32
dip_{0-31}	float32
sport_{0-31}	float32
dport_{0-31}	float32

Table 5.8: Embedded features for supervised learning, using binned IP2Vec.

5.2.3 Supervised Learning

The 371,770 samples are split into training, validation, and test sets, at a ratio of 60%, 20%, and 20%, respectively. Each set is chosen at random, with at least 1% of the test set being attack samples, chosen at random. The training and validation sets are split such that all classes are balanced between the two sets.

For each type of learning method we implement, we perform a random search of values to find the relative ranges for several hyperparameters. Once a smaller range of values are determined, we perform a validation step to determine the effectiveness of each set of hyperparameters in the selected range. The best performing set of parameters is then tested with the holdout set. We report this score as the performance of each of the models.

Prediction results in four types of labels; in binary classification, negative samples are usually the background data, while positive samples are in the class that we are interested in detecting.

- True Positive (TP) - predicted positive, was positive
- False Positive (FP) - predicted positive, was negative
- True Negative (TN) - predicted negative, was negative
- False Negative (FN) - predicted negative, was positive

A *confusion matrix* can be used to visualize these values. Rows in the confusion matrices shown represent all true values for a class, while the columns represent the class it was predicted to be. True positives exist along the diagonal. Basic scores can be calculated using these types of labels:

- *Accuracy* - ability to correctly classify data

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

- *Precision* - ability to avoid classifying negatives as positives

$$precision = \frac{TP}{TP + FP}$$

- *Recall* - ability to correctly classify positives as positives

$$recall = \frac{TP}{TP + FN}$$

We choose to use a combination of *precision* and *recall* as our metrics for evaluation, as accuracy is easily skewed by the number of TN points (in our case, `background`). Instead, we use an F1 score, calculated by taking the harmonic mean of the *precision* and *recall* of the prediction results, as described in Equation 5.2.3.

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

XGBoost

The first model we evaluate our feature spaces on is called XGBoost [67]. XGBoost is a method of combining decision trees through boosting, an ensembling method where the error from one tree is used to improve the performance of the next tree. As with many decision trees, XGBoost provides a means of interpreting the decision making process through feature importances, making it a desirable model for determining the effectiveness of features on classification.

To optimize the models, we run a search of values for the parameters `n_estimators`, `max_depth`, and `min_child_weight`. These values determine the number of trees to build, the maximum depth of the trees, and the minimum error in a node before splitting, respectively. Other parameters are set to default, or are constant and proportional to the feature spaces.

The non-embedded model produces an optimal validation score with parameters (203, 5, 3) for the search parameters, and an F1 score of 70.16% on the test set. The embedded model is tested with parameters (197, 5, 3), and resulted in a score of 78.69%. Looking at the confusion matrices for the final test set, we can see the embedded features reduce the number of false positives, while also reducing the number of attacks classified as `background`.

The prediction scores are shown in Table 5.9.

	Non-Embedded	Embedded
Training Score (%)	91.77	**96.37**
Validation Score (%)	78.32	**85.83**
Test Score (%)	70.16	**78.69**
Training Time	**6m24s**	1h29m29s

Table 5.9: Evaluation metrics for XGBoost with and without IP2Vec

Actual Values	Predicted Values						
	background	dos11	dos53s	dos53a	scan11	scan44	nerisbotnet
background	73433	2	38	2	120	8	8
dos11	3	10	55	0	1	0	0
dos53s	5	6	206	7	0	0	0
dos53a	2	0	5	88	0	0	0
scan11	86	0	0	0	123	0	0
scan44	1	0	0	0	1	18	14
nerisbotnet	5	0	1	0	0	3	103

Table 5.10: Confusion matrix of test set for XGBoost using non-embedded features

Random Forests

Random forests is a machine learning technique that stands as the converse of XGBoost. While XGBoost utilizes boosting to improve sequential trees, random forests averages the results of many decision trees to produce a prediction, an ensembling technique called *bagging*. Just as with XGBoost, we select random forests as a method of evaluation due to its ability to analyze which features were most important for classification, as well as its ease of use in the `scikit-learn` libraries.

We perform a search of values for the `n_estimators` to determine the optimal number of trees to average together, and `max_depth`, the maximum depth that a single tree can grow, and `max_features`, the amount of features used when building a tree. Unlike in XGBoost,

Actual Values	Predicted Values						
	background	dos11	dos53s	dos53a	scan11	scan44	nerisbotnet
background	73476	0	17	4	113	0	1
dos11	5	44	20	0	0	0	0
dos53s	4	5	208	7	0	0	0
dos53a	2	0	6	86	1	0	0
scan11	74	0	0	0	135	0	0
scan44	3	0	0	0	0	26	5
nerisbotnet	3	0	1	0	0	5	103

Table 5.11: Confusion matrix of test set for XGBoost using features embedded with IP2Vec

we choose different ranges of feature percentages for random forests, as training times were greatly increased when the maximum features was increased.

The non-embedded random forests was trained with `n_estimators=100`, `max_depth=15`, and `max_features=0.3` while the embedded random forests was trained with parameters 100, 15, and 0.02, respectfully. The confusion matrices of the test data for each are shown in Tables 5.13 and 5.14, with the final scores in Table 5.12. As shown, the embedded models produce higher F1 scores due to the reduced number of false positives generated.

	Non-Embedded	Embedded
Training Score (%)	76.07	**81.08**
Validation Score (%)	65.77	**70.09**
Test Score (%)	57.55	**71.66**
Training Time	**3m48s**	7m07s

Table 5.12: Evaluation metrics for random forests with and without IP2Vec

Multi-Layer Perceptron

The third model we choose to evaluate the feature spaces on is a multi-layer perceptron (MLP). Word embeddings such as IP2Vec can be used as a preprocessing step for other neural networks such as an MLP; however, in our case, we choose to use pretrained embeddings.

Actual Values	Predicted Values						
	background	dos11	dos53s	dos53a	scan11	scan44	nerisbotnet
background	73435	0	3	4	96	3	70
dos11	44	0	25	0	0	0	0
dos53s	43	0	167	13	0	0	1
dos53a	6	0	9	80	0	0	0
scan11	162	0	0	0	47	0	0
scan44	5	0	0	0	0	2	27
nerisbotnet	5	0	0	0	0	1	106

Table 5.13: Confusion matrix of test set for random forests using non-embedded features

Actual Values	Predicted Values						
	background	dos11	dos53s	dos53a	scan11	scan44	nerisbotnet
background	73608	0	2	1	0	0	0
dos11	54	0	15	0	0	0	0
dos53s	33	1	184	6	0	0	0
dos53a	4	0	11	80	0	0	0
scan11	164	0	0	0	45	0	0
scan44	1	0	0	0	0	31	2
nerisbotnet	5	0	0	0	0	7	100

Table 5.14: Confusion matrix of test set for random forests using features embedded with IP2Vec

This is to prevent including information about attacks into the embeddings of IP addresses and ports. We use the `scikit-learn` implementation of an MLP, which limits us to a single activation function type for all hidden layers we use. Each layer of a network multiplies its inputs by *weights*, sums the products, then passes the sums on to *activation functions*. The number of activation functions per layer is a configurable parameter called `hidden_dim`.

We search over hidden layer sizes in factors of 2 between 2 and the size of the input, meaning 1024 for the non-embedded data and 8196 for the embedded data. We choose to use a Rectified Linear Unit (ReLU) as the activation function, as it is the default function and provided better results when compared to logistic or tangent functions. The final networks

for the non-embedded and embedded data use `hidden_dim` values of 32 and 128, respectively.

The MLP performs the most poorly out of the test models, primarily due to the increased number of false positives. The non-embedded model produced an F1-score of 31.51%, while the embedded model performs slightly better at 46.61%.

	Non-Embedded	Embedded
Training Score (%)	64.19	**84.78**
Validation Score (%)	62.35	**77.64**
Test Score (%)	31.51	**46.61**
Training Time	**3m07s**	18m32s

Table 5.15: Evaluation metrics for MLP with and without IP2Vec

Actual Values	Predicted Values						
	background	dos11	dos53s	dos53a	scan11	scan44	nerisbotnet
background	72236	16	569	15	597	14	164
dos11	14	1	39	0	0	0	0
dos53s	15	2	189	9	2	1	1
dos53a	0	0	12	100	1	0	0
scan11	179	0	3	0	55	0	5
scan44	1	0	1	2	4	2	20
nerisbotnet	5	0	0	0	9	1	70

Table 5.16: Confusion matrix of test set for MLP using non-embedded features

5.2.4 Analysis

The primary means of interpreting the effects of the new features created from IP2Vec on the intrusion detection models is comparing the importances of features for the tree based methods: XGBoost and random forests. These feature importances are found in the appendix, with Figures A.1-A.4 for XGBoost and Figures A.5-A.8 for random forests.

In each of the decision trees, the feature representing the amount of time elapsed within a sample (`span`), had the most effect on the decision making process. This is expected, as many

Actual Values	Predicted Values						
	background	dos11	dos53s	dos53a	scan11	scan44	nerisbotnet
background	72551	113	177	2	724	19	25
dos11	6	50	15	0	2	0	0
dos53s	4	6	195	4	1	0	0
dos53a	1	1	5	93	0	0	0
scan11	124	0	1	0	115	0	0
scan44	2	0	0	0	0	30	2
nerisbotnet	2	0	0	0	0	6	78

Table 5.17: Confusion matrix of test set for MLP using features embedded with IP2Vec

of the attacks generate large amounts of flows within a short period of time. However, past this, XGBoost and random forests distinguish themselves from one another with the features they choose to use. XGBoost utilizes the number of bytes in each of flow and the duration of each flow. On the other hand, random forests tended to use flags to make decisions, with a few samples of duration seen in the top 150.

Including the embedded flow information shifts the feature importances, particularly in XGBoost. In XGBoost, we see features generated from embedding IP addresses take the place of `dur_i` in the top 150, where `i` is the index of the flow in the input. This shows that XGBoost was able to use host information to classify attacks. And because the F1-scores improved with the addition of this information, we can infer that this features provided more information about how to classify a flow than did other features like `dur_i`.

Random forests exhibits the same behavior, though not as exaggerated as XGBoost. Embedded information from IP2Vec begins appearing in the bottom of the top 150 features, taking the place of flag features and the duration of each flow. Again, `span` is the most important feature, as it is the easiest of method distinguishing between an attack like denial-of-service and a period of background flows. Other features are then used to determine the type of attack that is occurring.

Chapter 6

Discussion

As shown, IP2Vec provides a meaningful contribution to the classification performance of an intrusion detection system. The use of a static embedding as used in this work introduces issues when in use for a long period of time. As IP addresses can change behavior over time, using an embedding trained with past data will use old behavior to attempt to predict future attacks. As noted in [13], training the embedding over time would fit in new behaviors, updating the definition of an IP address as it changes in a method called *online learning*. However, care should be taken with online learning, as attackers could potentially poison embeddings with specially crafted traffic, changing embeddings enough to allow future attacks to remain undetected [75], [76].

6.1 Future Work

Approaching network intrusion detection and NetFlow data as an NLP problem introduces many avenues for new work, particularly for determining the quality of embeddings created from NetFlow traffic. In an NLP setting, word embeddings are validated through the use of *analogies*, which takes advantage of the additive compositionality of Word2Vec (Section 2.2). In this work, we rely on t-SNE for determining similarities between IP addresses. Developing analogies for IP2Vec would give a more quantitative evaluation for the quality of the embeddings.

While we utilize decision trees for their interpretability and ease of use, more complicated machine learning methods such as recurrent or convolutional neural networks may prove to be a more appropriate technique to use embeddings in classification. These methods provide a better means of giving temporal relationships between flows in a period of time, compared to the static window generated in this work. IP2Vec is implemented in this work to be a pregenerated set of definitions, but when combined with a neural network, the embeddings can be trained while also training the network for classification. The combination of IP2Vec with other deep learning methods may prove to be more useful for intrusion detection.

This work primarily focused on detecting attacks whose signatures are detectable from features that are not related to IP2Vec. The most important feature for XGBoost and random forests was the `span` feature, that describes the amount of time that passes within a window. Future works that would like to test the embeddings more richly should focus on host-based attacks, rather than network attacks like denial of service. These could include a variety of botnet attacks such as in CTU'13 [24].

Chapter 7

Conclusion

In this thesis, we provide an analysis of several datasets for network security, and their uses in intrusion detection with machine learning. Through this analysis, we find a need for utilizing modern and more realistic datasets, as well as a means of representing flow information like IP addresses. We implement and evaluate IP2Vec [13], and modify to allow unknown values. We verify IP2Vec's ability to capture similarities between hosts through the CTU'13 dataset, by qualitatively comparing clusters containing botnet-infected hosts. We demonstrate IP2Vec even further using the UGR'16 dataset, where we visualize the similarities between web servers and clients.

We then evaluate the ability of IP2Vec to contribute to the classification of attacks that occur in the UGR'16 dataset. Our evaluation shows that the inclusion of embedded information improves F1-scores for XGBoost, random forests, and an MLP, at the cost of training time in most cases.

In the future, focus should go primarily to developing evaluations for embedded network information. Future work would also contribute to developing NetFlow datasets containing host-based attacks with background information, allowing for better testing of IP2Vec.

Bibliography

[1] Cisco, *Snort - Network Intrusion Detection & Prevention System*, 2018. [Online]. Available: https://www.snort.org/.

[2] The Bro Project, *The Bro Network Security Monitor*, 2014. [Online]. Available: https://www.bro.org/.

[3] Splunk Inc., *Splunk*, 2018. [Online]. Available: https://www.splunk.com/.

[4] Elasticsearch BV, *Open Source Search & Analytics - Elasticsearch*, 2018.

[5] Z. Chen and Y. F. Li, "Anomaly detection based on enhanced DBScan algorithm", *Procedia Engineering*, vol. 15, pp. 178–182, 2011, ISSN: 18777058. DOI: 10.1016/j.proeng.2011.08.036. [Online]. Available: http://dx.doi.org/10.1016/j.proeng.2011.08.036.

[6] W. Chen, F. Kong, F. Mei, G. Yuan, and B. Li, "A Novel Unsupervised Anomaly Detection Approach for Intrusion Detection System", *Proceedings - 3rd IEEE International Conference on Big Data Security on Cloud, BigDataSecurity 2017, 3rd IEEE International Conference on High Performance and Smart Computing, HPSC 2017 and 2nd IEEE International Conference on Intelligent Data and Securit*, pp. 69–73, 2017. DOI: 10.1109/BigDataSecurity.2017.56.

[7] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, "1999 DARPA off-line intrusion detection evaluation", *Computer Networks*, vol. 34, no. 4, pp. 579–595, 2000, ISSN: 13891286. DOI: 10.1016/S1389-1286(00)00139-0.

[8] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set", *IEEE Symposium on Computational Intelligence for Security and*

Defense Applications, CISDA 2009, no. Cisda, pp. 1–6, 2009, ISSN: 2329-6267. DOI: 10.1109/CISDA.2009.5356528.

[9] K. Flanagan and E. Fallon, "Network Anomaly Detection in Time Series using Distance Based Outlier Detection with Cluster Density Analysis", pp. 116–121,

[10] K. Flanagan, E. Fallon, A. Awad, and P. Connolly, "Self-Configuring NetFlow Anomaly Detection using Cluster Density Analysis", pp. 421–427, 2017.

[11] R. Miao, R. Potharaju, M. Yu, and N. Jain, "The Dark Menace : Characterizing Network-based Attacks in the Cloud",

[12] V. Paxson and S. Floyd, "Why We Dont Know How To Simulate the Internet", in *Proceedings of the 29th conference on Winter simulation*, 1997, pp. 1037–104.

[13] M. Ring, A. Dallmann, D. Landes, and A. Hotho, "IP2Vec: Learning similarities between IP addresses", *IEEE International Conference on Data Mining Workshops, ICDMW*, pp. 657–666, 2017, ISSN: 23759259. DOI: 10.1109/ICDMW.2017.93.

[14] G. Maciá-Fernández, J. Camacho, R. Magán-Carrión, P. García-Teodoro, and R. Therón, "UGR'16: A new dataset for the evaluation of cyclostationarity-based network IDSs", *Computers and Security*, vol. 73, pp. 411–424, 2018, ISSN: 01674048. DOI: 10.1016/j.cose.2017.11.004. [Online]. Available: https://doi.org/10.1016/j.cose.2017.11.004.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and Their Compositionality", pp. 1–9, ISSN: 10495258. DOI: 10.1162/jmlr.2003.3.4-5.951.

[16] M. Mihaltz, *word2vec-GoogleNews-vectors*, 2016. [Online]. Available: https://github.com/mmihaltz/word2vec-GoogleNews-vectors.

[17] L. J. P. Van Der Maaten and G. E. Hinton, "Visualizing high-dimensional data using t-sne", *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008, ISSN: 1532-4435. DOI: 10.1007/s10479-011-0841-3. [Online]. Available: https://lvdmaaten.github.io/publications/papers/JMLR_2008.pdf%0Ahttp://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=7911431479148734548related:VOiAgwMNy20J.

[18] M. S. Alsadi and A. H. Hadi, "Visualizing clustered botnet traffic using T-SNE on aggregated NetFlows", *Proceedings - 2017 International Conference on New Trends in Computing Sciences, ICTCS 2017*, vol. 2018-Janua, pp. 179–184, 2018. DOI: 10.1109/ICTCS.2017.30.

[19] V. Paxson, "Bro: A system for detecting network intruders in real-time", *Computer Networks*, vol. 31, no. 23, pp. 2435–2463, 1999, ISSN: 13891286. DOI: 10.1016/S1389-1286(99)00112-7.

[20] M. V. Mahoney and P. K. Chan, "An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection", *In Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection*, vol. 2820, no. Ll, pp. 220–237, 2003, ISSN: 0302-9743. DOI: 10.1007/b13476. [Online]. Available: http://www.springerlink.com/index/GF9CJ2VYY8E7QNK7.pdf.

[21] U. K. Archive, *KDD Cup 1999 Data*, 1999. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[22] L. Portnoy, E. Eskin, and S. J. Stolfo, "Intrusion detection with unlabeled data using clustering", *ACM CSS Workshop on Data Mining Applied to Security (DMSA)*, pp. 5–8, 2001. [Online]. Available: http://academiccommons.columbia.edu/catalog/ac:138734.

[23] K. Leung and C. Leckie, "Unsupervised anomaly detection in network intrusion detection using clusters", *Proceedings of the Twenty-eighth Australasian conference on Computer Science - Volume 38*, vol. 38, no. January, pp. 333–342, 2005, ISSN: 14451336. [Online]. Available: http://portal.acm.org/citation.cfm?id=1082161.1082198.

[24] S. García, M. Grill, J. Stiborek, and A. Zunino, "An empirical comparison of botnet detection methods", *Computers and Security*, vol. 45, pp. 100–123, 2014, ISSN: 01674048. DOI: 10.1016/j.cose.2014.05.011.

[25] N. Moustafa and J. Slay, "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)", *2015 Military Communications and Information Systems Conference (MilCIS)*, pp. 1–6, 2015. DOI: 10.1109/MilCIS.2015.7348942. [Online]. Available: http://ieeexplore.ieee.org/document/7348942/.

[26] IXIA, *PerfectStorm*, 2018. [Online]. Available: http://www.ixiacom.com/products/perfectstorm.

[27] J. Camacho, A. Pérez-Villegas, P. Garciá-Teodoro, and G. MacIá-Fernández, "PCA-based multivariate statistical network monitoring for anomaly detection", *Computers and Security*, vol. 59, pp. 118–137, 2016, ISSN: 01674048. DOI: 10.1016/j.cose.2016.02.008. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2016.02.008.

[28] R. Sommer and V. Paxson, "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection", *2010 IEEE Symposium on Security and Privacy*, pp. 305–316, 2010, ISSN: 10816011. DOI: 10.1109/SP.2010.25. [Online]. Available: http://ieeexplore.ieee.org/document/5504793/.

[29] C. Gates and C. Taylor, "Challenging the Anomaly Detection Paradigm: A Provocative Discussion", *Proceedings of the 2006 workshop on New security paradigms*, pp. 21–29, 2007. DOI: 10.1145/1278940.1278945.

[30] M. Mahoney and P. Chan, "Learning rules for anomaly detection of hostile network traffic", *Third IEEE International Conference on Data Mining*, pp. 601–604, 2003, ISSN: 15504786. DOI: 10.1109/ICDM.2003.1250987. [Online]. Available: http://ieeexplore.ieee.org/document/1250987/.

[31] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu, "An efficient k-means clustering algorithm: analysis and implementation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 7, pp. 881–892, 2002, ISSN: 0162-8828. DOI: 10.1109/TPAMI.2002.1017616.

[32] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise", *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, vol. 2, pp. 226–231, 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=3001460.3001507.

[33] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases", *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, vol. 1, pp. 103–114, 1996, ISSN: 01635808. DOI: 10.1145/233269.233324. [Online]. Available: http://doi.acm.org/10.1145/233269.233324.

[34] L. Ertoz, E. Eilertson, A. Lazarevic, P.-n. Tan, V. Kumar, J. Srivastava, and P. Dokas, "MINDS - Minnesota Intrusion Detection System", *Next Generation Data Mining*, pp. 199–218, 2004. [Online]. Available: http://www.it.iitb.ac.in/~deepak/deepak/courses/mtp/papers/minds-minnesota%20intrusion%20detection%20system.pdf.

[35] F. Iglesias and T. Zseby, "Analysis of network traffic features for anomaly detection", *Machine Learning*, no. October 2014, pp. 59–84, 2015, ISSN: 0885-6125. DOI: 10.1007/

s10994-014-5473-9. [Online]. Available: http://dx.doi.org/10.1007/s10994-014-5473-9.

[36] I. Syarif, A. Prugel-bennett, and G. Wills, "Unsupervised Clustering Approach for Network", pp. 135–145, 2012. DOI: 10.1007/978-3-642-30507-8{_}13.

[37] S. Kumar, S. Kumar, and S. Nandi, "Multi-density clustering algorithm for anomaly detection Using KDD'99 dataset", *Communications in Computer and Information Science*, vol. 190 CCIS, no. PART 1, pp. 619–630, 2011, ISSN: 18650929. DOI: 10.1007/978-3-642-22709-7{_}60.

[38] M. Ahmed and A. N. Mahmood, "Novel Approach for Network Traffic Pattern Analysis using Clustering-based Collective Anomaly Detection", *Annals of Data Science*, vol. 2, no. 1, pp. 111–130, 2015, ISSN: 2198-5804. DOI: 10.1007/s40745-015-0035-y. [Online]. Available: http://link.springer.com/10.1007/s40745-015-0035-y.

[39] Y. Guan and A. A. Ghorbani, "Y-Means: A Clustering Method for Intrusion Detection", no. May, pp. 1083–1086, 2003. DOI: 10.1109/CCECE.2003.1226084.

[40] M. Wurzenberger, F. Skopik, M. Landauer, P. Greitbauer, R. Fiedler, and W. Kastner, "Incremental Clustering for Semi-Supervised Anomaly Detection applied on Log Data", *Proceedings of the 12th International Conference on Availability, Reliability and Security - ARES '17*, pp. 1–6, 2017. DOI: 10.1145/3098954.3098973. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3098954.3098973.

[41] J. Dromard and P. Owezarski, "Integrating Short History for Improving Clustering Based Network Traffic Anomaly Detection", *Proceedings - 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W 2017*, pp. 227–234, 2017. DOI: 10.1109/FAS-W.2017.152.

[42] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain", *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958, ISSN: 0033295X. DOI: 10.1037/h0042519.

[43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, no. 6088, pp. 533–536, 1986, ISSN: 00280836. DOI: 10.1038/323533a0.

[44] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition", *Proceedings of the IEEE*, 1998. DOI: 10.1109/5.726791.

[45] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview", 2014.

[46] S. Hochreiter and J. Scmidhuber, "Long Short-Term Memory", vol. 9, no. 8, pp. 1735–1780, 1997.

[47] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient Flow in Recurrent Nets : the Difficulty of Learning Long-Term Dependencies", p. 464, 2001.

[48] A. K. Ghosh, A. Schwartzbard, M. Schatz, A. K. Ghosh, A. Schwartzbard, and M. Schatz, "Learning Program Behavior Profiles for Intrusion Detection", 1999.

[49] R. C. Staudemeyer, "Applying long short-term memory recurrent neural networks to intrusion detection", no. 56, 2015.

[50] J. Kim, J. Kim, T.-T.-H. Le, and H. Kim, "Long Short Term Memory Recurrent Neural Network Classifier for Intrusion Detection", 2016.

[51] T.-t.-h. Le, J. Kim, and H. Kim, "An Effective Intrusion Detection Classifier Using Long Short-Term Memory with Gradient Descent Optimization", pp. 1–5, 2017.

[52] C. Yin, Y. Zhu, J. Fei, and X. He, "A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks", vol. 5, 2017.

[53] P. Torres and C. Catania, "An Analysis of Recurrent Neural Networks for Botnet Detection Behavior",

[54] J. R. Quinlan, "Induction of Decision Trees", *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986, ISSN: 15730565. DOI: 10.1023/A:1022643204877.

[55] ——, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, ISBN: 1-55860-238-0.

[56] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. CRC Press, 1984.

[57] S. Peddabachigari, A. Abraham, and J. Thomas, "Intrusion detection systems using decision trees and support vector machines", *International Journal of Applied . . .*, no. July, pp. 1–16, 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.4079&rep=rep1&type=pdf.

[58] S. Sheen and R. Rajesh, "Network intrusion detection using feature selection and Decision tree classifier", *TENCON 2008 - 2008 IEEE Region 10 Conference*, pp. 1–4, 2008. DOI: 10.1109/TENCON.2008.4766847. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4766847.

[59] S. S. Sivatha Sindhu, S. Geetha, and A. Kannan, "Decision tree based light weight intrusion detection using a wrapper approach", *Expert Systems with Applications*, vol. 39, no. 1, pp. 129–141, 2012, ISSN: 09574174. DOI: 10.1016/j.eswa.2011.06.013. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2011.06.013.

[60] Z. Yanjie, "Network Intrusion Detection System Model Based on Data Mining", *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2016 17th IEEE/ACIS*, vol. 9, no. 9,

pp. 359–370, 2015. DOI: 10.1109/SNPD.2016.7515894. [Online]. Available: http://ieeexplore.ieee.org/document/7515894/.

[61] G. Stein, B. Chen, A. S. Wu, and K. A. Hua, "Decision tree classifier for network intrusion detection with GA-based feature selection", *Proceedings of the 43rd annual southeast regional conference on - ACM-SE 43*, vol. 2, p. 136, 2005. DOI: 10.1145/1167253.1167288. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1167253.1167288.

[62] S. Revathi and A. Malathi, "A Detailed Analysis on NSL-KDD Dataset Using Various Machine Learning Techniques for Intrusion Detection", *International Journal of Engineering Research & Technology (IJERT)*, vol. 2, no. 12, pp. 1848–1853, 2013. [Online]. Available: http://www.ijert.org/browse/volume-2-2013/december-2013-edition?download=7027%3Aa-detailed-analysis-on-nsl-kdd-dataset-using-various-machine-learning-techniques-for-intrusion-detection&start=280.

[63] P. A. A. Resende and A. C. Drummond, "A Survey of Random Forest Based Methods for Intrusion Detection Systems", *ACM Computing Surveys*, vol. 51, no. 3, 52:1–52:27, 2018, ISSN: 03600300. DOI: 10.1145/3178582. [Online]. Available: https://doi.org/10.1145/3178582.

[64] N. Farnaaz and M. A. Jabbar, "Random Forest Modeling for Network Intrusion Detection System", *Procedia Computer Science*, vol. 89, pp. 213–217, 2016, ISSN: 18770509. DOI: 10.1016/j.procs.2016.06.047. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2016.06.047.

[65] H. S. Hota and A. K. Shrivas, "Data Mining Approach for Developing Various Models Based on Types of Attack and Feature Selection as Intrusion Detection Systems (IDS)", *Intelligent Computing, Networking, and Informatics*, vol. 243, 2014, ISSN: 2194-5357.

DOI: 10.1007/978-81-322-1665-0. [Online]. Available: http://link.springer.com/10.1007/978-81-322-1665-0.

[66] A. Tesfahun and D. Lalitha Bhaskari, "Intrusion detection using random forests classifier with SMOTE and feature reduction", *Proceedings - 2013 International Conference on Cloud and Ubiquitous Computing and Emerging Technologies, CUBE 2013*, pp. 127–132, 2013. DOI: 10.1109/CUBE.2013.31.

[67] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System", 2016, ISSN: 0146-4833. DOI: 10.1145/2939672.2939785. [Online]. Available: http://arxiv.org/abs/1603.02754%0Ahttp://dx.doi.org/10.1145/2939672.2939785.

[68] S. Dhaliwal, A.-A. Nahid, and R. Abbas, "Effective Intrusion Detection System Using XGBoost", *Information*, vol. 9, no. 7, p. 149, 2018, ISSN: 2078-2489. DOI: 10.3390/info9070149. [Online]. Available: http://www.mdpi.com/2078-2489/9/7/149.

[69] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification", pp. 183–194, 2015. DOI: 10.1145/2857705.2857713. [Online]. Available: http://arxiv.org/abs/1511.04317.

[70] A. Kun, J. Michael, E. Valla, N. S. Neggatu, and A. W. Moore, "Network traffic classification via neural networks", no. 912, 2017, ISSN: 1476-2986.

[71] D. Zhang, H. Xu, Z. Su, and Y. Xu, "Chinese comments sentiment classification based on word2vec and SVMperf", *Expert Systems with Applications*, vol. 42, no. 4, pp. 1857–1863, 2015, ISSN: 09574174. DOI: 10.1016/j.eswa.2014.09.011. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2014.09.011.

[72] J. Lilleberg, Y. Zhu, and Y. Zhang, "Support Vector Machines and Word2vec for Text Classification with Semantic Features", *Proceedings of IEEE 14th International*

*Conference on Cognitive Informatics & Cognitive Computing (ICCI*CC)*, pp. 136–140, 2015. DOI: 10.1109/ICCI-CC.2015.7259377.

[73] TensorFlow Team, *Introducing TensorFlow Feature Columns*, 2017. [Online]. Available: https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html.

[74] M. Bailey, E. Cooke, J. Farnam, Y. Xu, and M. Karir, "A survey of botnet technology and defenses", *Cybersecurity Applications & Technology*, 2009.

[75] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, "Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning", *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2018-May, no. 1, pp. 19–35, 2018, ISSN: 10816011. DOI: 10.1109/SP.2018.00057.

[76] J. Steinhardt, P. W. Koh, and P. Liang, "Certified Defenses for Data Poisoning Attacks", no. i, 2017, ISSN: 10495258. [Online]. Available: http://arxiv.org/abs/1706.03691.

Appendices

Appendix A

Feature Importances

This section displays the top 150 features used in models trained with and without the IP2Vec data. This includes both XGBoost and random forests, but does not include MLP as neural networks do not include a means of extracting the importance of features.

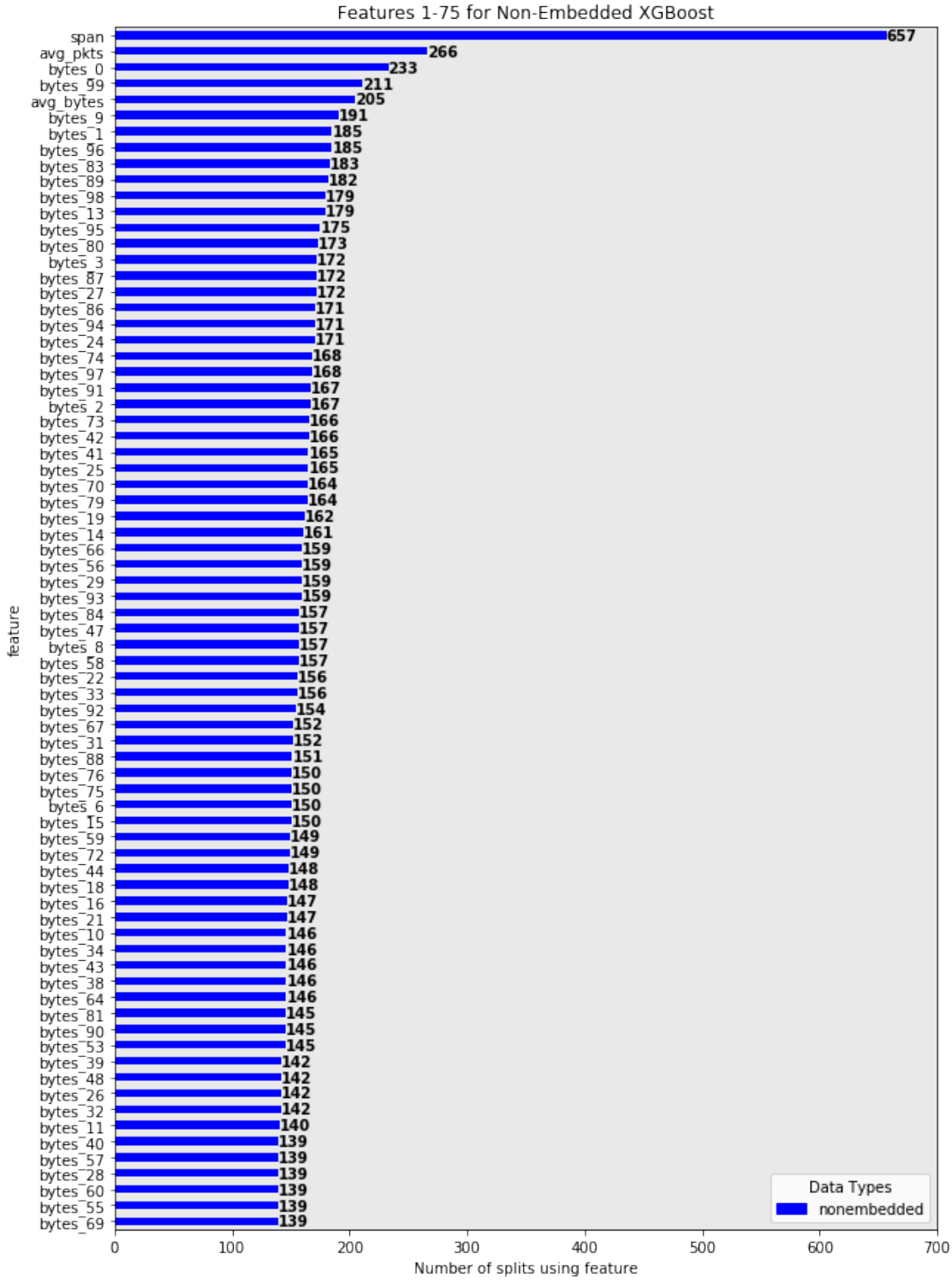


Figure A.1: Importances for Features Ranked 1-75 in Non-Embedded XGBoost

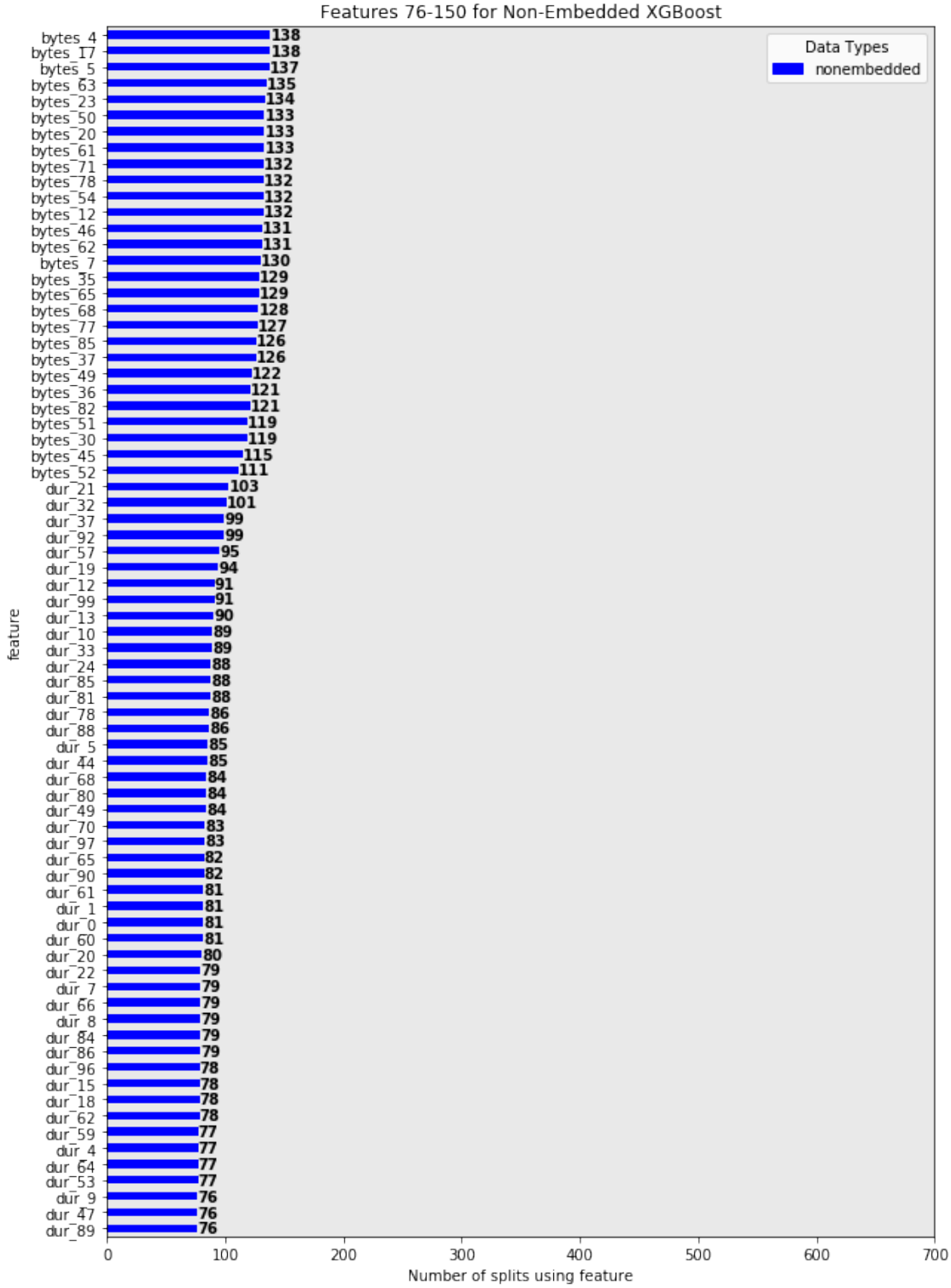


Figure A.2: Importances for Features Ranked 76-150 in Non-Embedded XGBoost

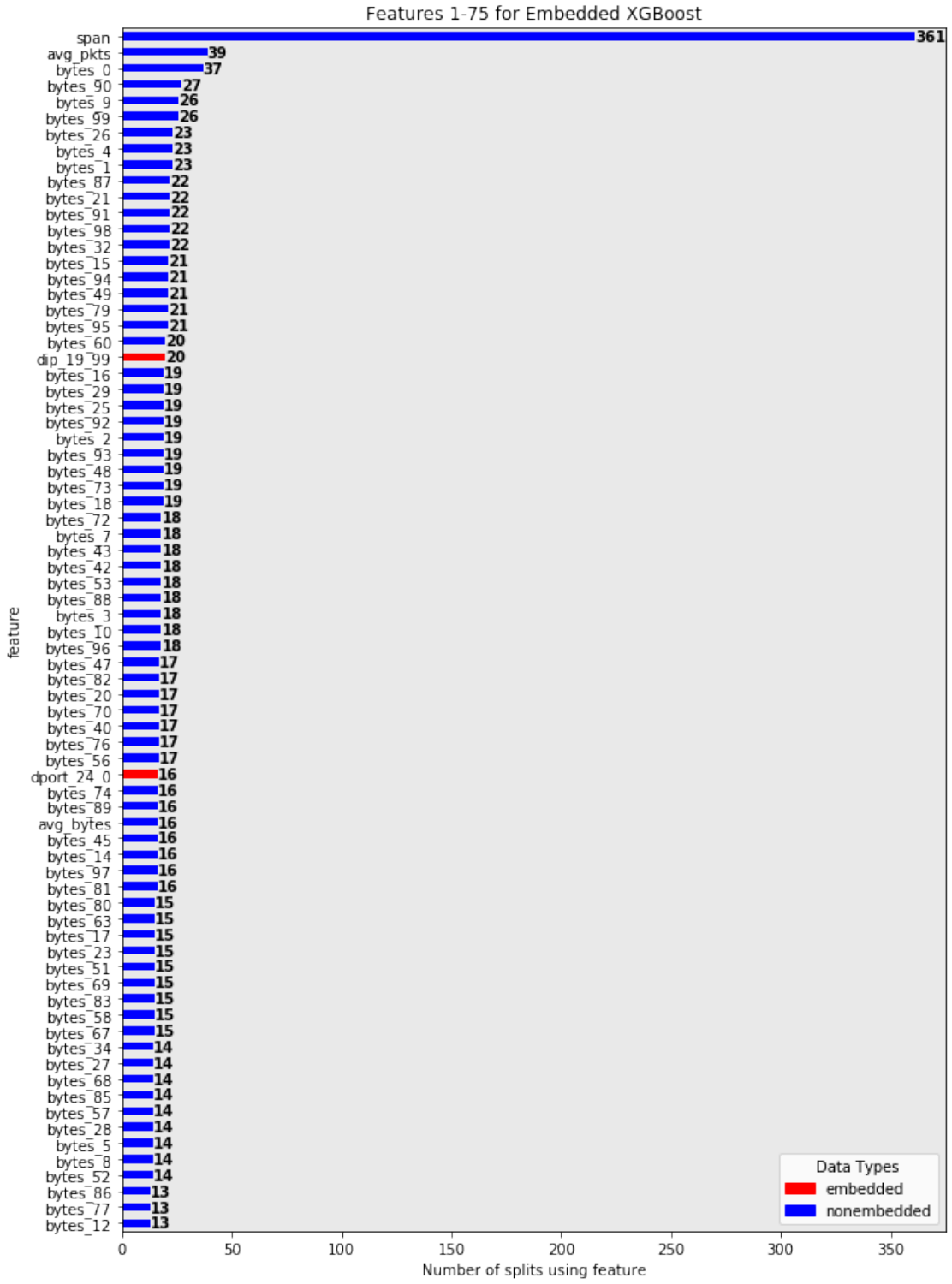


Figure A.3: Importances for Features Ranked 1-75 in Embedded XGBoost

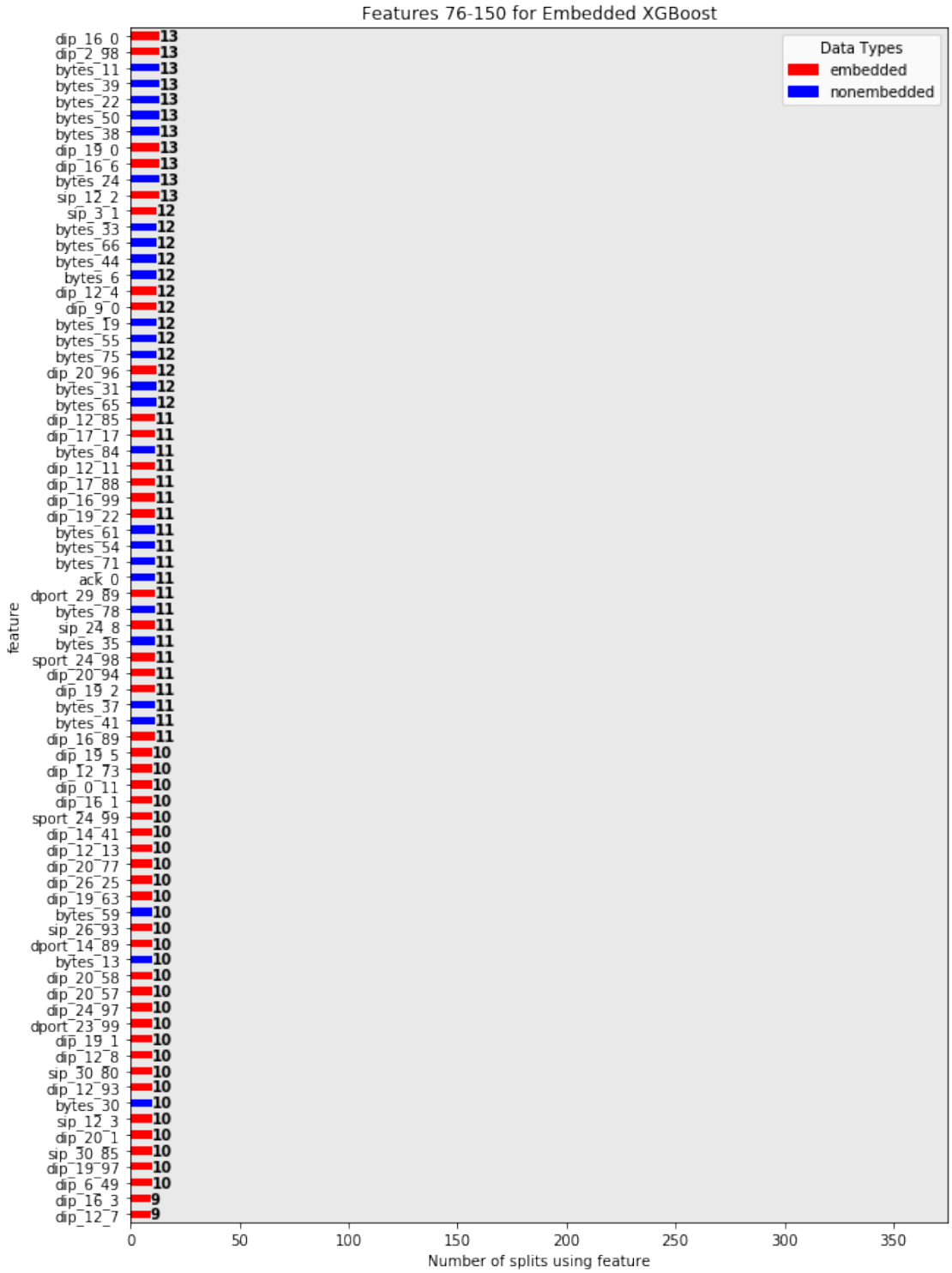


Figure A.4: Importances for Features Ranked 76-150 in Embedded XGBoost

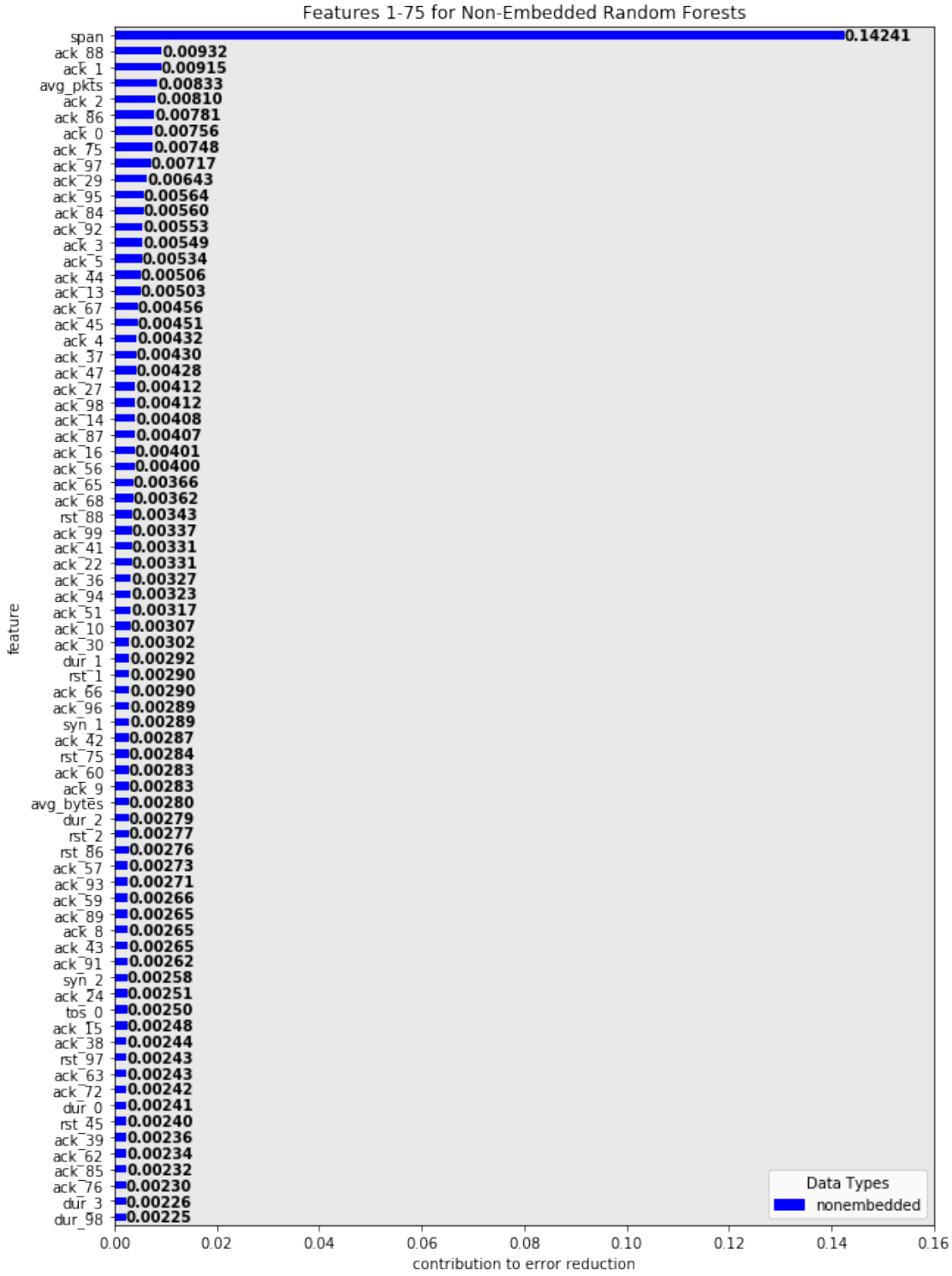


Figure A.5: Importances for Features Ranked 1-75 in Non-Embedded Random Forests

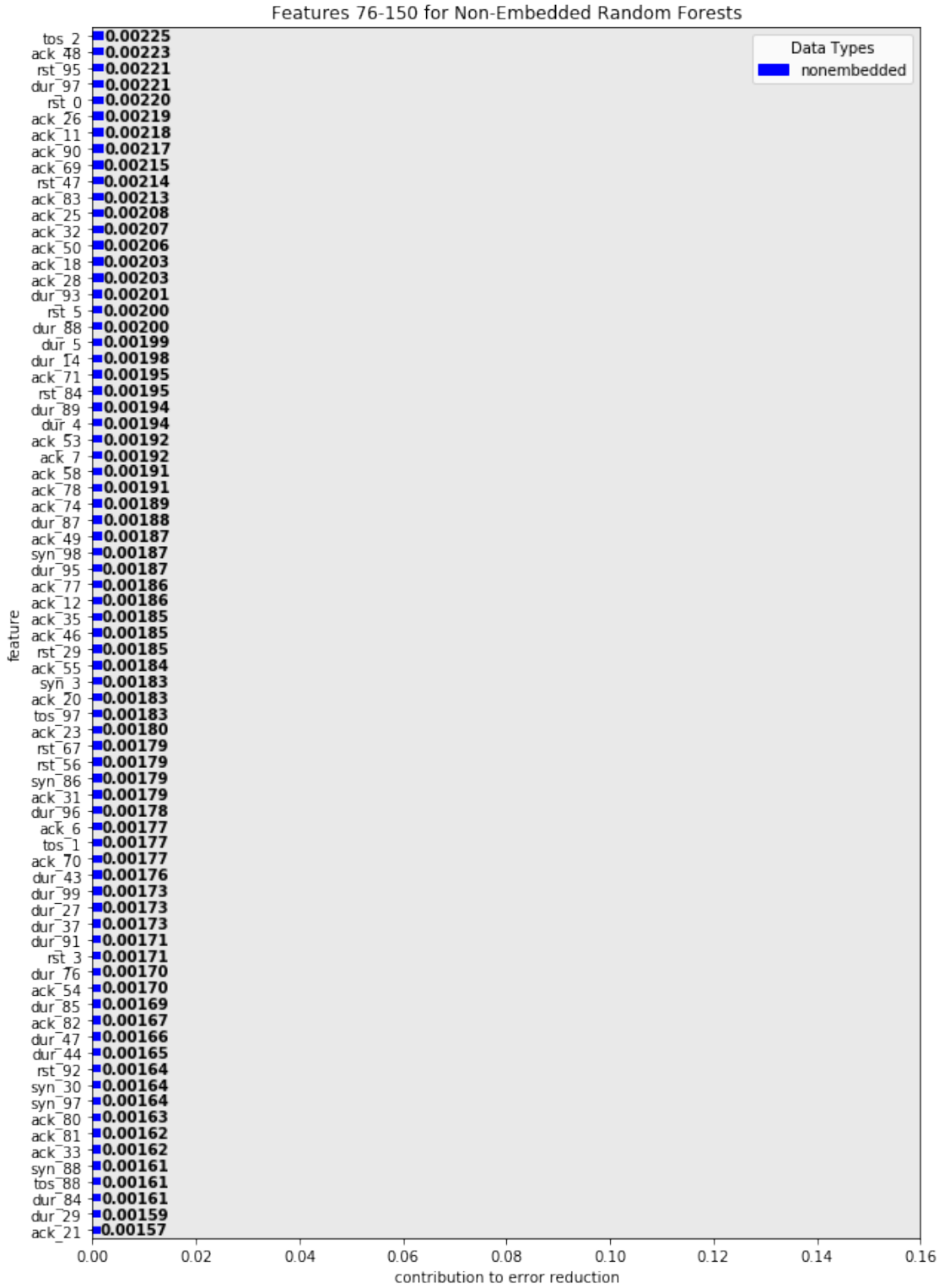


Figure A.6: Importances for Features Ranked 76-150 in Non-Embedded Random Forests

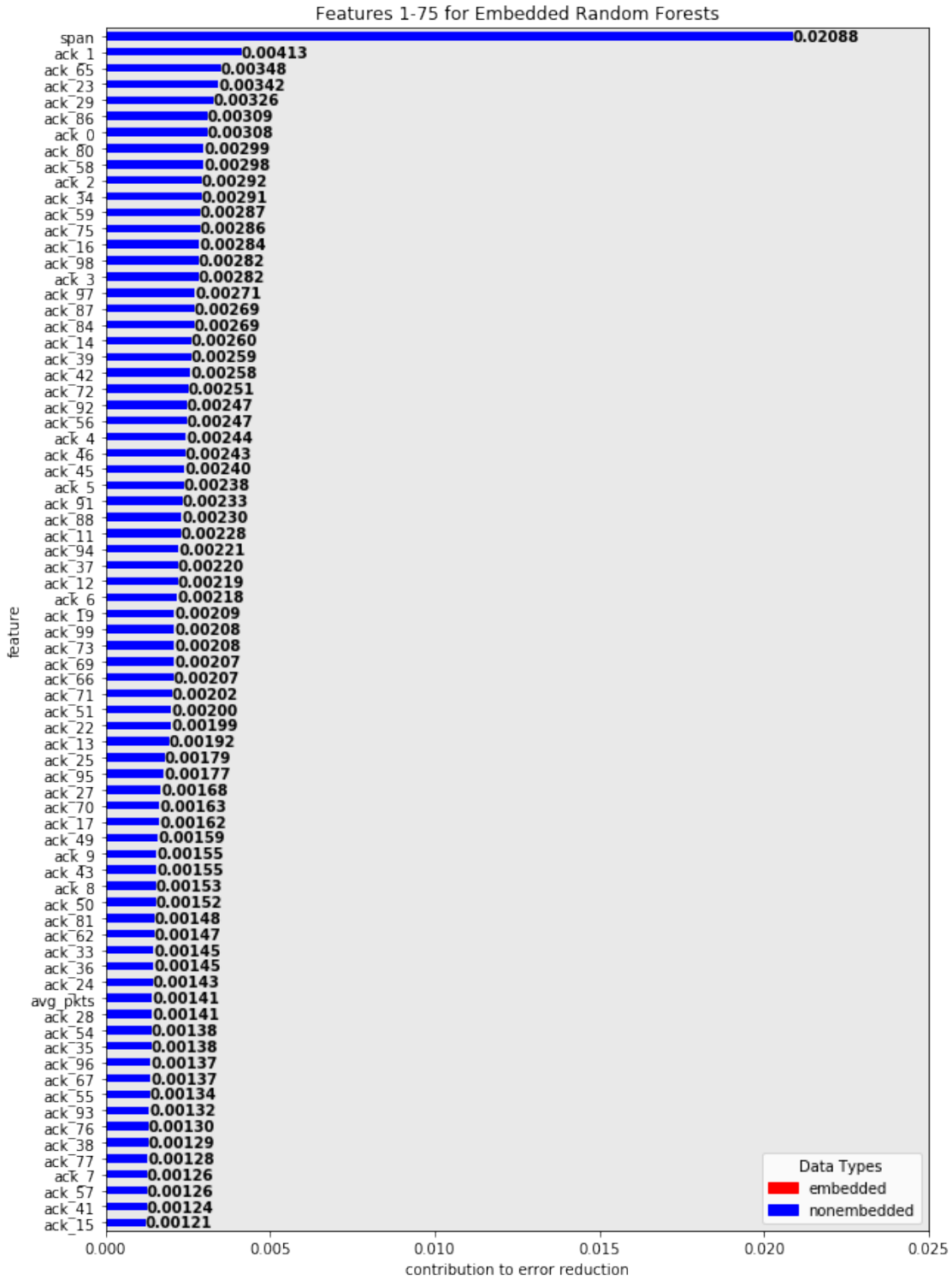


Figure A.7: Importances for Features Ranked 1-75 in Embedded Random Forests

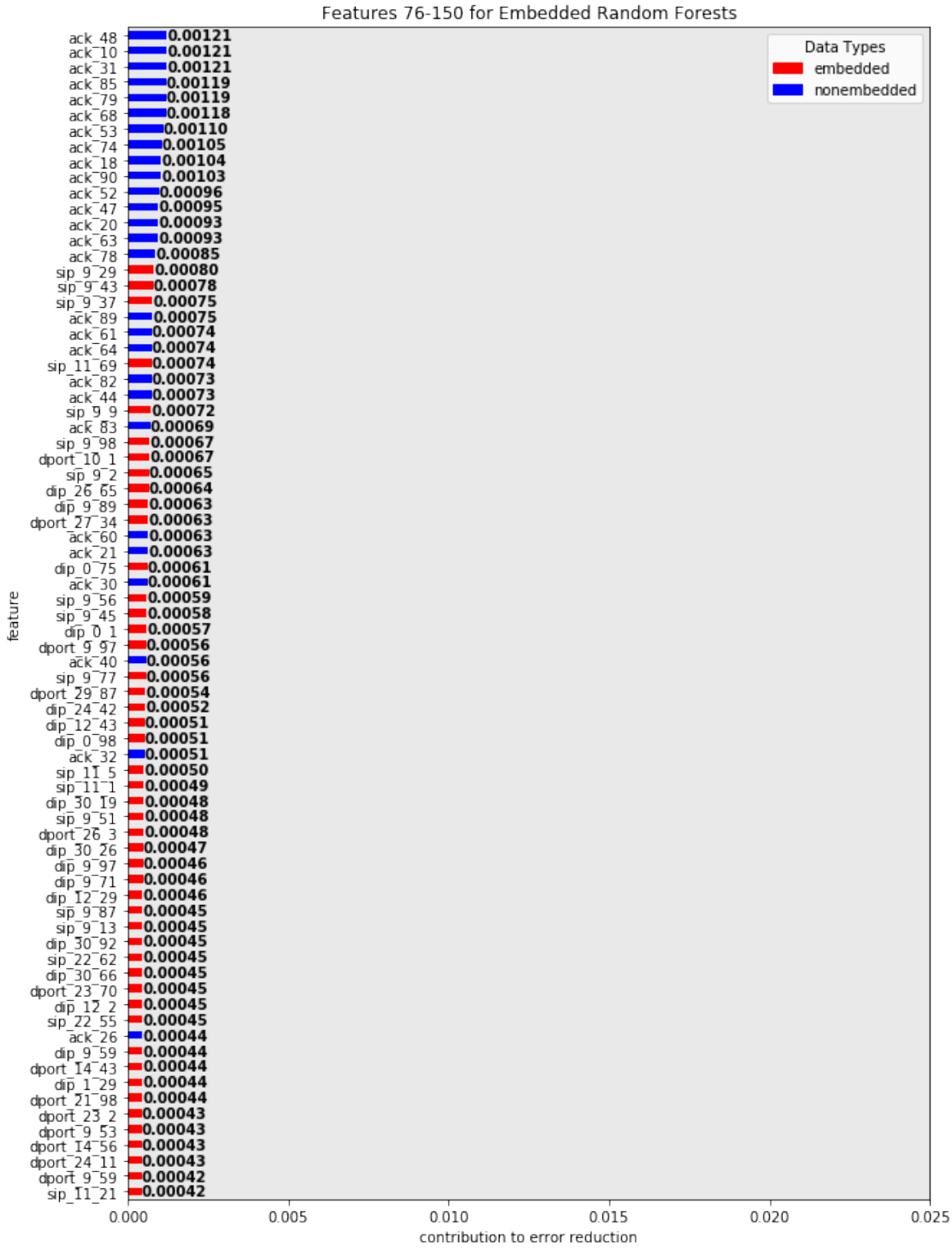


Figure A.8: Importances for Features Ranked 76-150 in Embedded Random Forests