CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Mandlík Šimon** Personal ID number: **457029**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Inference on a Graph Representing a Computer Network

Bachelor’s thesis title in Czech:

Inference na grafu reprezentující počítačovou síť

Guidelines:

1. Create a survey of a literature about learning complicated inference rules over graphs.
2. Study the prior art about multi-instance learning.
3. Learn frameworks for describing graphs and multiple-instacne learning in in Julia programming language.
4. Represent the problem of interacting nodes in computer network as a graph inference problem and identify which rules do we want to learn.
5. Implement the learning of the inference rules in the chosen framework.
6. Experimentally compare the quality of inferred rules to the probability threat propagation algorithm.

Bibliography / sources:

[1] Kevin M Carter, Nwokedi Idika, and William W Streilein. Probabilistic threat propagation for malicious activity detection. In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pages 2940-2944. IEEE, 2013.
[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016.
[3] Tomáš Pevný and Petr Somol. Using neural network formalism to solve multiple-instance problems. In International Symposium on Neural Networks, pages 135-142. Springer, 2017.
[4] Jan Stiborek, Tomáš Pevný, and Martin Rehák. Multiple instance learning for malware classification. arXiv preprint arXiv:1705.02268, 2017.

Name and workplace of bachelor’s thesis supervisor:

Ing. Tomáš Pevný, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **04.01.2018** Deadline for bachelor thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

Ing. Tomáš Pevný, Ph.D. Supervisor’s signature	doc. Ing. Tomáš Svoboda, Ph.D. Head of department’s signature	prof. Ing. Pavel Ripka, CSc. Dean’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

This page is intentionally left blank.

Inference on a Graph Representing a Computer Network

Bachelor's thesis

Šimon Mandlík
Supervisor: Ing. Tomáš Pevný, Ph.D

2018

FACULTY
OF ELECTRICAL
ENGINEERING
CTU IN PRAGUE

Department of Cybernetics

This page is intentionally left blank.

First of all, I would like to express my deepest gratitude to my supervisor Tomáš Pevný, because without his guidance and persistent help this thesis would not have been possible. I appreciated insightful discussions with his colleagues, particularly Jan Jusko, who helped me with a dataset collection. I am also much obliged to Veronika Mandlíková and Jan Franců, who assisted with proofreading. Last but not least, I want to thank my girlfriend and my family for moral support during writing.

This page is intentionally left blank.

Author statement for undergraduate thesis

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date . . . .

. . . .
Author's signature

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne . . . .

. . . .
Podpis autora

This page is intentionally left blank.

Abstract

The task of inferring unknown random variables on a graph is a widely researched topic. Representing a system of random variables as a graphical model is less constraining as the graphs allow the capture of dependencies between random variables. The more general model comes at the cost of limited assumption set, for instance, i.i.d. assumption used in the classical statistical machine learning setting becomes invalid. As a result, the size of the knowledge base of algorithms for performing inference on graphical models is small, no guarantees are usually provided, and the number of inferential algorithms with learnable parameters is even lower. In this thesis, we propose a novel formalism for performing graph inference on graphical models. We show that the formalism embraces many inference procedures and is very general compared to the inflexible models of prior art. To demonstrate our claims we attempt to solve the problem of classifying unknown computer servers as malicious or benign based only on the network traffic and a blacklist of known malicious domains in the network. To accomplish this, we formulate the task in terms of the proposed formalism and implement the solution. Early experiments indicate that our method performs superiorly to the patented state-of-the-art Probabilistic threat propagation algorithm and at the same time is more general and offers greater flexibility.

Keywords: **graph inference**, **Markov random fields**, **message passing inference**, **multiple instance learning**, **neural networks**, **Probabilistic threat propagation**, **computer security**, **malicious server classification**

Abstrakt

Inference hodnot neznámých náhodných proměnných v grafu je široce studovaný problém. Grafová reprezentace systému náhodných proměnných je méně svazující, protože do grafu můžou být zakódovány vztahy mezi proměnnými. Tento obecnější model ale získáváme za cenu toho, že některé běžné předpoklady přestávají platit, jako například podmínka nezávislých a stejně rozdělených vzorků, která se uplatňuje ve statistickém strojovém učení. Toto má za následek, že databáze použitelných metod a algoritmů je omezená a zpravidla nejsou poskytovány žádné záruky. Metod, které by umožňovaly jakoukoliv formu učení svých parametrů, je ještě méně. V této práci představíme nový formalismus pro úlohu inference nad grafickými modely a demonstrujeme jeho flexibilitu a obecnost v porovnání se staršími modely. Pro experimentální ověření našich tvrzení se pokusíme vyřešit úlohu klasifikace počítačových serverů jako škodlivých anebo neškodných, to vše pouze pomocí sledování sít'ového provozu a dodaného blacklistu známých hrozeb. Tuto úlohu jsme zformulovali pomocí nového formalismu a vzniklou metodu řešení implementovali. Rané experimenty ukazují, že tato metoda překonává patentovaný state-of-the-art algoritmus 'Probabilistic threat propagation' a zároveň je více obecná.

Klíčová slova: **inference na grafu**, **'Markov random fields'**, **'message passing inference'**, **multi-instanční učení**, **neuronové sítě**, **'Probabilistic threat propagation'**, **počítačová bezpečnost**, **klasifikace škodlivých serverů**

Contents

List of Figures J

List of Tables J

List of Algorithms J

1 Introduction 1
1.0.1 Computer Security Domain . . . 2
1.0.2 Related work . . . 4
1.0.3 Thesis structure . . . 5

2 Background 6
2.1 Graphical models . . . 6
2.1.1 Bayesian networks . . . 7
2.1.2 Markov random fields . . . 8
2.1.3 Conditional random fields . . . 9
2.2 Inference on graphical models . . . 10
2.2.1 Message passing algorithms . . . 10
2.2.2 Sum-product algorithm . . . 11
2.2.3 Max-sum algorithm . . . 13

3 Method description 15
3.1 Task formulation . . . 15
3.2 MINI formalism . . . 16
3.2.1 Generative model . . . 16
3.2.2 Learning the messages . . . 18
3.2.3 Multiple instance learning predictor . . . 19
3.2.4 Neural networks function approximators . . . 21
3.2.5 Recapitulation . . . 23
3.2.6 Multi-step algorithm . . . 24

4 Specific adjustments and techniques 26
4.1 Dealing with bipartite graphs . . . 26
4.1.1 Construction of adjacency relation . . . 26
4.2 Graph pruning . . . 28
4.2.1 Complete pruning . . . 30
4.2.2 Simple pruning . . . 30
4.3 A mapping from instance relations to feature vectors . . . 31
4.4 Sampling . . . 32
4.4.1 Bag subsampling . . . 32
4.4.2 High-level graph sampling . . . 33
4.5 Dealing with imbalanced classes . . . 34

5 Experiments 37
5.1 Data . . . 37
5.1.1 Graphs . . . 37
5.1.2 Ground truth . . . 37
5.2 Evaluation pipeline . . . 38
5.2.1 Leave one domain out . . . 39
5.2.2 Fold validation . . . 39
5.3 Hyperparameters . . . 40
5.3.1 Model hyperparameters . . . 40
5.3.2 Training hyperparameters . . . 41
5.4 Results . . . 43
5.4.1 Baseline model . . . 44
5.4.2 Further experiments and improvements . . . 44
5.4.3 Comparison to Probabilistic Threat Propagation . . . 47

6 Conclusion 50
6.0.1 Future work . . . 50

A Probabilistic threat propagation 53
A.0.1 Approximate Inference . . . 53
A.0.2 Weights . . . 53

B Dataset details 54

C DVD Content 57

Acronyms 58

Nomenclature 59

Bibliography 61

List of Figures

2.1 Bayesian network . . . 8
2.2 Markov random field . . . 9
2.3 Conditional random field . . . 9
2.4 Markov random field and its corresponding factor graph . . . 11
2.5 Factorization of a tree graph . . . 12
3.1 A picture of the whole model . . . 20
3.2 A picture of the specific model . . . 23
3.3 Multi-step layered model . . . 25
4.1 Construction of a graph of servers and an adjacency relation on its vertices . . . 27
4.2 Degree distributions on a bipartite graph . . . 28
4.3 Influence of the graph pruning on a set of positive servers in the graph . . . 29
4.4 A characteristical loss function curve of learning with priors of classes. . . . 35
5.1 An illustration of K-fold validation methods . . . 39
5.2 Linear interpolation on PR curves . . . 43
5.3 PR curves of the baseline model . . . 44
5.4 PR curves of the baseline model with different numbers of negative instances in the bag 44
5.5 PR curves of the models with layers of different widths . . . 45
5.6 PR curves of the models of different depth . . . 46
5.7 PR curves of the models trained with different batch sizes . . . 46
5.8 PR curves of the models with different activation functions . . . 47
5.9 Comparison of selected models with the Probabilistic threat propagation method . . . 48

List of Tables

4.2 Table of features used in experiments . . . 31
5.1 Benefits and drawbacks of both compared methods . . . 49
B.1 The most and the least frequently visited domains in the dataset. . . . 54
B.2 Properties of the datasets used in experiments before pruning . . . 55
B.3 Properties of the datasets used in experiments after pruning . . . 55
B.4 Examples of low-level labels on the blacklist . . . 56

List of Algorithms

1 Construction procedure of an adjacency relation in a graph of servers . . . 27
2 Complete pruning technique . . . 30
3 Simple pruning technique . . . 30
4 Sampling procedure used to sample bags from a graph of servers . . . 34

This page is intentionally left blank.

Chapter 1

Introduction

The field of *Machine learning* has been on the rise in recent years, especially its offspring *Deep learning* [Goodfellow et al., 2016], which has repeatedly drawn considerable attention with its success in image recognition [Hu et al., 2017], machine translation [Wu et al., 2016], audio synthesis [van den Oord et al., 2016] or mastering two-player games [Silver et al., 2017]. Close to human or even superhuman performance was achieved in many tasks due to increasingly more accessible amounts of data and computing power. In the field of statistical machine learning in the classical supervised setting data is obtained in the form of a vector $\boldsymbol{x}$ of fixed size and its corresponding label y. Moreover, it is usually assumed that datapoints drawn from an unknown underlying distribution are independently and identically distributed (i.i.d. assumption). The goal is to devise a decision strategy performing well on unseen data. Unfortunately, it is not possible to frame every task in this manner.

Firstly, the data may be structured in such a way that it is impossible or counter-intuitive to transform samples into a vector of fixed size. However, the same data can be more than often naturally represented in a different way. *Multiple instance learning (MIL)* [Babenko, 2008] is a paradigm that instead of single instances considers higher-level entities called *bags*. Each bag contains one or more instances, and labels are only known on the level of bags. Instances are not explicitly labeled although the existence of a class distribution on instances is usually assumed. One can imagine image classification task where each picture contains multiple objects, all of them crucial for making a decision. Even though every object surely belongs to an unknown class of objects, we are interested only in classifying the picture as a whole. This requires lossless capture and efficient utilization of knowledge about objects on the higher level of abstraction than single instances. Instead of representing a bag as one vector $\boldsymbol{x}$, we group vectors describing every instance from the bag into a set $\{\boldsymbol{x}_i\}_{i=1}^{n}$ where n denotes the bag's size. For binary classification, a standard presumption is often employed that bag is labeled positive if at least one of its instances has a positive label and labeled negative otherwise. By taking a whole set of instances into account, MIL facilitates modeling of structured data, but this rather new direction has not yet been studied as extensively compared to other research areas. As a result, opting for MIL comes with a price of limited supply of algorithms and datasets.

Secondly, in some real-world examples, i.i.d. assumption is incorrect. We often observe a somehow structured system of objects that influence significantly each other. Reasoning about a single datapoint while neglecting its dependencies on others results into information loss, nevertheless, the exact opposite approach of considering every possible relation in the data is infeasible for larger datasets due to the quadratic complexity. On the other hand, it is often the case that these dependencies are mostly local to small groups of objects and thus sparse.

Lastly, there are many domains where labels are expensive or difficult to obtain, therefore it is very hard to acquire sufficiently large and representative datasets. For instance, medical investigation of a patient may be costly, and as a result, big datasets from the healthcare domain are rare. Another difficulty is that the data is sometimes not labeled with absolute confidence as some tasks are very difficult or almost impossible for a human annotator to perform without mistakes. Such type of labels is called *weak labels* and the setting *weak supervision*. We still seek an efficient method that would predict labels for previously unseen examples with sufficient accuracy, however, in the case of

incomplete or weak data annotation it is unclear how to evaluate model's performance. The classic example comes from the Computer security (CS) domain, where new malware or viruses appear every day and even though an experienced network analyst can recognize suspicious behavior, it is not possible to label every object of interest with high precision. *Semi-supervised learning* [Chapelle Olivier, 2006] is a paradigm that allows a part of training data to be unlabeled.

In this work, we address a task of (probabilistic) inference on a graph in which each object of interest is represented as a vertex and relations between objects as edges. This problem poses the same challenges that were just described – it is hard to describe an instance by a fixed size vector, there may be dependencies between objects and not every label is known and a hundred percent correct. We propose a novel *Multiple instance neural inference (MINI)* formalism for the inference in graphical models and derive a method which given only a graph and a small subset of its nodes with positive labels can infer labels for the rest of nodes in the graph. In comparison to standard models, the method is effective, much more flexible and more general. MINI formalism poses a task as a MIL problem, therefore we are able to model complex relations between nodes accommodating any degree distribution in a graph. By using a specific neural network formalism, we train models with high capacity successfully avoiding overfitting. Our method generalizes well to other graphs from a domain, where training data comes from. Furthermore, it is possible to include domain-specific features of nodes, edges or both in the graph to improve accuracy. The method can be readily extended to work in the multi-class environment, but we will consider only the binary case. Additionally, it outputs a real number between zero and one which expresses the confidence of the decision. We measure the performance of this approach by solving the problem of domain classification and outperform *Probabilistic threat propagation (PTP)*, the state-of-the-art algorithm commonly used in the CS domain. The powerful iterative message passing inference algorithm serves as a foundation for the formalism and positive results were obtained, although the specific implementation reported here uses only one iterative step. This shows a great promise for the design of multi-step iterative method in the future.

1.0.1 Computer Security Domain

Even though the proposed method can be effortlessly employed in an arbitrary domain, in this work we focus on the CS domain. The arrival of the Internet and its swift growth gave rise to an entirely new sort of crime, sometimes called *cybercrime*. Nowadays, malware programs are being created faster than before, and protection against such threats becomes increasingly difficult. In the past, a traditional method of *signature-based detection* compared pre-defined pieces of information extracted from the inspected file or its communication, such as the hash of the file, various (sub)sequences in compiled binary files or a special structure of data packets the program sends. After that, the extracted signature was compared to an existing database of known threats. These databases are called blacklists and form the core of the majority of antivirus software. This approach requires a careful manual or semi-automatic examination of every newly-discovered malware, which results in blacklists being temporarily outdated, sometimes even for several days. Furthermore, modern, sophisticated malware makes use of polymorphic or encrypted code segments, therefore devising a corresponding signature may be very hard or nearly impossible. The birth of code generation or code mutation automated tools made this situation even more desperate, as this has allowed malware creators to produce new threats with less effort. Fighting this losing battle, in which we are destined to be one step behind all the time, is not the right way to go.

As a consequence, a radically different *behavior-based* approach is gaining popularity among CS companies. The main idea is not to inspect the structure of the program but to observe its behavior. This can be done by monitoring its network activity or system calls. Contrary to a signature of a program, the behavior is really tough to conceal. To give an example, there are various types of

malware, each type with a different purpose, but the overwhelming majority of them aim to make money for their creators. These types include ransomware, which encrypts the whole hard disk of the user and demands payment in exchange for the decryption, adware, which generates revenue by displaying additional advertisement on user's screen, or spyware, which tracks user's activity on the internet without their consent and may steal important passwords. To obtain instructions on which action to take next, the malware must connect to a controlling server, called Command and control (C&C) server. This is a typical example of unavoidable behavior which can be used to detect malware. Similarly, a program suddenly opening many files may be ransomware, many computations being carried out indicate a presence of cryptocurrency mining malware or a lot of connections to distant countries may be a consequence of the device being used for Distributed denial of service (DDoS) attack. Tools of statistical analysis and machine learning proved themselves successful in classifying programs based on their behavior because the explosion of the volume of data makes learning bigger high-capacity models without overfitting feasible.

There is plenty of concrete tasks to address in the CS domain. One can for example attempt to identify infected computers connected to the network, classify computer programs by inspecting their code or binary representation or detect suspicious network flows[1]. In this thesis, we look into the task of classifying dangerous servers, which may contain malicious code or give instructions to malware installed on client's computer. If we are accurate enough, the detected servers may be persecuted, effectively dealing with the puppeteer himself instead of single puppets in the form of malware programs. As a result, the malicious activity would be greatly reduced. The task is hard because the CS domain suffers from exactly the same problems as described at the beginning of this chapter, that is, manual data annotation is hard and servers influence each other and thus can be by no means considered independent samples. This led to new terminology for weak annotations – instead of reliable ground truth, we have only *tips* at our disposal and one needs to bear in mind that many servers with malicious content may not be detected by network analysts. It is not, therefore, clear how to even define a bulletproof performance measure given incomplete and unreliable information. On top of that, computer networks are fastly evolving environment and datasets quickly become obsolete. As more and more devices connect to the network and throughput of a network grows, a vast amount of data is produced posing a computational and representational challenge for anyone trying to harness it.

There are many types of dangerous servers and to simplify things, we split servers into two groups and call them *malware* or *positive* servers and *benign* or *negative* servers. We perform classification by examining records of communication between users connected to the network and domains residing in it. To be able to employ the proposed formalism, we construct a *graph of servers* and then test models by classifying vertices in this graph. Even though a *domain* is perhaps a more appropriate name for the objects we classify, we are going to use the term interchangeably with a *server* in order not to confuse the reader, when we speak about the CS domain. Moreover, previous work [Yu et al., 2010] showed that malicious servers tend to form local *communities* in network graphs. This result means that relations between considered objects are sparse and we will refer to it in the rest of the text. Lastly, by *users* we mean all devices connected to the network, not only personal computers and mobile phones but also printers, televisions and other objects able to communicate.

[1] `https://en.wikipedia.org/wiki/NetFlow`

1.0.2 Related work

Graphical models and inference

To study graphical probabilistic models' fundamentals and existing methods for inference in them, we consulted [R. Kindermann, 1980, Lafferty et al., 2001] and [Bishop, 2006]. This thesis relies on an important proof that every probability distribution can be factorized and thus represented as an undirected graph from [Hammersley, 1971].

Probabilistic graph inference was solved exactly for some subclasses of graphs [Baum and Petrie, 1966, Pearl, 1988], but for general graphs, approximate methods need to be employed. Such methods include Loopy belief propagation [Frey and MacKay, 1998], Markov chain monte carlo [Wang et al., 2000] or Graph cuts [Kohli and Torr, 2005]. However, the most relevant method for our work is the *sum-product algorithm* introduced in [Kschischang et al., 2001], which served as our starting point. This algorithm belongs to the class of so-called *Message Passing* algorithms, where inference is considered an iterative process of sending 'messages' between nodes in a graph.

An example of utilization of the sum-product algorithm is demonstrated for instance in [Horng Polo Chau et al., 2011], but it requires manually crafted definitions of potential functions in the graph model. Learning the potential functions involves performing inference procedure at every iteration of gradient learning, which slows down the procedure significantly. In [Yu et al., 2017] neural networks were used for the task, but only on binary Markov random fields. A suggestion to predict messages instead of computing them directly (which is both computationally expensive and also proved to be only an approximation) appeared in [Ross et al., 2011]. Previous work in computer vision domain [Lin et al., 2015b] uses Convolutional neural networks to learn messages in message passing inference for the task of semantic image segmentation.

Our main focus was to perform graph inference with the assistance of neural networks and develop a method that would be efficient and accurate for any degree distribution of the graph. We followed a Multiple instance learning [Babenko, 2008] learning framework and leveraged work proposed in [Pevný and Somol, 2017, Stiborek et al., 2017] to solve graph inference task framed as a MIL problem.

Probabilistic threat propagation

We test our method on the specific task of inferring malicious servers from the computer security domain. The state-of-the-art approach is to use Probabilistic threat propagation [Carter et al., 2013, Kazato et al., 2016], an iterative method, which despite offering limited possibilities performs well in practice. We refer the reader to Appendix A, where we sketch out how the Probabilistic threat propagation algorithm works. It was adapted and successfully deployed in *Cisco Systems, Inc.* [Jusko et al., 2016], and the method was patented[2] afterward. Probabilistic threat propagation demands hand-crafted parameters and offers no intuitive way to take external knowledge about objects into account compared to our method. What's more, it appears to be superior to the Probabilistic threat propagation approach according to early experiments.

[2] https://patents.google.com/patent/US9596321B2/en

1.0.3 Thesis structure

In the second chapter, we familiarize the reader with previous work on graphical models that capture relations between objects and briefly describe the message passing algorithm, which inspired us. We describe the MINI formalism and the graph inference method derived from it in the third chapter and in the fourth chapter discuss specific techniques and methods, which may be useful to implement to decrease computational complexity of tasks we solve with the formalism. The same chapter also presents the details about the server classification problem and our approach to solving it. The description of experiments carried out is in the fifth chapter together with a comparison to PTP method and a conclusion with several ideas for future work is presented in the last, sixth chapter. Several appendices, a list of acronyms, a nomenclature and a bibliography close the thesis.

Chapter 2

Background

In this chapter, we briefly introduce *graphical models*, which naturally encode dependencies and interactions between objects, and the class of *message passing inference algorithms*, a standard method for performing inference in graphical models.

2.1 Graphical models

As we shall see in the sections to come, it is possible to convert (joint) probability distributions composed of many random variables into a graphical representation, where each node represents one random variable and edges represent probabilistic relations between variables. These models are called *graphical models* or *structured probabilistic models* and bring several advantages:

- Describing the entire joint probability distribution (for example in a table) may be very inefficient. Typically, there are no interactions between most pairs of random variables involved in the distribution and the CS domain is no exception. Graphical models take advantage of this and encode relations efficiently into a graph.
- Graph representation makes it easier to gain an insight into properties of probability distribution and interactions between variables comprising this distribution.
- Graph as a mathematical structure is widely studied, and the graph theory provides many useful algorithms and methods.

We define a graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, \dots, v_n\}$ is the set of n vertices associated with random variables and $\mathcal{E} \subseteq \{\{u, v\}\, ;\, u, v \in \mathcal{V}, u \neq v\}$ is the set of edges in G representing functional dependencies between variables. We shall call the elements of $\mathcal{V}$ as 'vertices', 'nodes', 'random variables' or just 'variables' and the set $\mathcal{E}$ as an *adjacency relation* since it in reality is a symmetric, irreflexive binary relation. Note that we do not allow loops as it does not make sense in the context of random variables.

One of the most important characteristics of graph models is *conditional independence*. Two random variables X and Y are said to be conditionally independent given a set of other random variables $\mathbf{Z} = Z_1, \dots, Z_k$, denoted $X \perp\!\!\!\perp Y \mid \mathbf{Z}$, if the following equation holds:

$$p(X \mid Y, \mathbf{Z}) = p(X \mid \mathbf{Z}) \tag{2.1}$$

or equivalently:

$$p(X, Y \mid \mathbf{Z}) = p(X \mid \mathbf{Z}) p(Y \mid \mathbf{Z}) \tag{2.2}$$

for all possible values of X, Y and $\mathbf{Z}$.

We shall show that the conditional independence makes both structure of graphical models and inference in them simpler and easier to grasp. Graphical models build on the fact, that every probability distribution can be broken into a product of functions of its variables, which we call *factors*:

$$p(\boldsymbol{X}) = \prod f(\boldsymbol{X}) \tag{2.3}$$

The definition of a factor varies from one type of graphical model to another, but usually, it describes a form of dependency between variables or sets of variables. We distinguish two main categories of graphical models – directed graphical models, also known as *Bayesian networks*, and undirected graphical models, sometimes referred to as *Markov random fields* or *Markov networks*. We also describe the specific subclass of Markov random fields, undirected graphical models known as *Conditional random fields*, where some random variables in the graph are observed, due to their importance later. By describing a joint probability as a graph we can define factors as functions representing local conditional distributions on the variables. Indeed, in practice, we often work with probability distributions defined on the set of random variables, where many variables are conditionally independent of each other given some set of other nodes. This enables us to capture important relations between variables while retaining favorable computational complexity.

All graphical models we work with are defined to exhibit so-called *Markov property*, which means that two random variables represented by nodes X_i and X_j are conditionally independent given set of nodes $\boldsymbol{S}$ if $\boldsymbol{S}$ forms a *separating subset*[1] in the graph between X_i and X_j. As a result, every two variables not connected in a graph must be conditionally independent given all other nodes in the graph. We can consequently define the *Markov blanket (MB)* of the node X_i, denoted $\mathbf{MB}(X_i)$ as a minimal set of vertices in the graph, such that every other vertex $X_j \notin \mathbf{MB}(X_i)$ is conditionally independent of X_i given its Markov blanket $\mathbf{MB}(X_i)$:

$$X_i \perp\!\!\!\perp X_j \mid \mathbf{MB}(X_i) \tag{2.4}$$

As a result, the inference of the probability of X_i given every other vertex in the graphs turns into:

$$\begin{aligned} p(X_i \mid \{X_j\,;\, X_i \neq X_j\}) &= p(X_i \mid \{X_j\,;\, X_j \notin \mathbf{MB}(X_i) \cup \{X_i\}\}\,, \mathbf{MB}(X_i)) \\ &= p(X_i \mid \mathbf{MB}(X_i)) \end{aligned} \tag{2.5}$$

Here we used the definition of Markov blanket and the conditional independence in (2.1). This result demonstrates the important role the concepts of conditional independence and Markov blanket play in the modeling of structured data.

2.1.1 Bayesian networks

Bayesian network (BN) is a type of graphical model, which models joint probability distribution as a directed acyclic graph. Factorization of joint probability is defined in this case according to sum and product rule of probability into marginal and conditional probabilities:

$$p(X_1, X_2, \ldots, X_n, Y) = p(X_1, X_2, \ldots, X_n \mid Y)p(Y) \tag{2.6}$$

[1] Separating subset between two vertices u and v in the graph is a minimal set of other vertices such that u becomes unreachable from v after the removal of the separating subset from the graph.

Edge incidence relation represents conditional probability between two random variables and relationship between a factorization of the joint probability and its graphical representation stems from the properties of the joint distribution. By implication, every node is conditionally independent of its non-descendants given all its parents. A trivial example of a Bayesian network can be seen in Figure 2.1. As we do not directly work with Bayesian networks in this thesis and every important aspect of these models for our work have already been discussed, we end their description here.

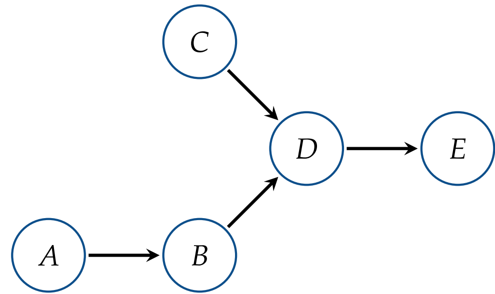


Figure 2.1: Example of factorization of probability distribution $p(A, B, C, D, E) = p(A)p(B \mid A)p(C)p(D \mid B, C)p(E \mid D)$ into factors. It can be clearly seen that B is dependent on A, D is dependent on C, B and E is dependent on D. Additionally, E is conditionally independent of all other nodes given its parent D and D is conditionally independent of A given B and C.

2.1.2 Markov random fields

Markov random field (MRF) [R. Kindermann, 1980] is an undirected graphical model, where the joint probability distribution on a set of random variables is represented as an undirected graph. Factorization of joint probability distribution into a product of functions defined over variables which are local to the graph is not as straightforward as in the case of Bayesian Networks. Intuitively two random variables not connected by an edge in the graph should be conditionally independent of each other given every other vertex in the graph. This calls for the use of *maximal cliques*[2] for encoding dependence between variables into connectivity properties of a graph. The factorization of the probability distribution is then proportional to the product of *potential functions* ψ_K:

$$p(\boldsymbol{X}) \propto \prod_K \psi_K(\boldsymbol{X}_K) \tag{2.7}$$

where $\boldsymbol{X}_K$ is a set of variables which are represented as the nodes in clique K.

The definition of the Markov blanket of a vertex X_i is rather straightforward as it is the set of all its neighbors in the graph $\mathcal{N}(X_i) = \{X_j \in \mathcal{V}\,;\, \{X_i, X_j\} \in \mathcal{E}\}$, where:

$$X_i \perp\!\!\!\perp X_j \mid \mathcal{N}(X_i) \tag{2.8}$$

for every $X_j \notin \mathcal{N}(X_i)$. The Hammersley-Clifford theorem [Hammersley, 1971] states, that every positive probability distribution can be factorized into a product of potential functions defined over maximal cliques in the graph and therefore represented as Markov random field. The theorem also holds in the reverse direction, that is, every Markov random field, whose density can be factorized over its cliques, represents a probability distribution. This theorem is one of the foundation stones of our method.

[2]Maximal clique is a maximal complete subgraph (adding another node from graph breaks this completeness).

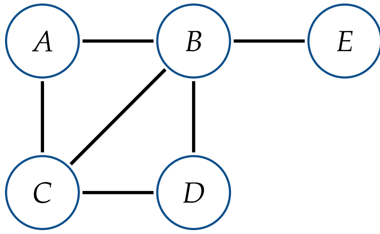


Figure 2.2: Example of representing probability distribution $p(A, B, C, D, E)$ as an undirected graph. This particular graph depicts factorization into a form $p(A, B, C, D, E) = \frac{1}{Z}\psi(A, B, C)\psi(B, C, D)\psi(B, E)$ as $\{A, B, C\}$, $\{B, C, D\}$ and $\{B, E\}$ are maximal cliques in the graph. For example, E is conditionally independent of every other node given B, A and D are conditionally independent given their separating subset $\boldsymbol{S} = \{B, C\}$ and $\mathbf{MB}(C) = \{A, B, C\}$ is the Markov blanket of variable C.

Potential functions need not satisfy the properties of a probability function, particularly they do not have to sum or integrate to one. However, they are required to be non-negative. Taking this into account we can normalize the factorization in (2.7):

$$p(\boldsymbol{X}) = \frac{1}{Z} \prod_{K} \psi_K(\boldsymbol{X}_K) \tag{2.9}$$

where Z is a normalizing constant, sometimes called *partition function*, defined as:

$$Z = \int_{\boldsymbol{X}} \prod_{K} \psi_K(\boldsymbol{X}_K)\, d\boldsymbol{X} \tag{2.10}$$

where summation can be used in the case of discrete distributions. Exact evaluation of this constant may pose a significant computational challenge as the complexity of computation scales exponentially with the number of random variables. Fortunately, in many applications, this term cancels out. A simple instance of a Markov random field and its factorization can be seen in Figure 2.2. As opposed to Bayesian networks where each factor has a clear meaning, potential functions have no immediate interpretation. One particular point of view comes from the minimum total potential energy principle in physics, which states that a system always approaches the state with the lowest potential energy. Because potential functions are defined as non-negative, they are often interpreted as:

$$\psi_C(\boldsymbol{X}_C) = \exp(-E(\boldsymbol{X}_C)) \tag{2.11}$$

where E is an energy of the variables in clique. This is favorable since now multiplication of potential functions turns into addition and the state with the lowest energy corresponds to the state with the highest probability.

2.1.3 Conditional random fields

If some random variables in a Markov random field are observed, the model turns into *Conditional random field (CRF)*, therefore CRFs are just a special case of MRFs. CRFs might come in handy when solving different inference tasks and the task of classifying computer servers, which we attempt to solve later on, is in reality best interpreted as the inference in CRF. Indeed, we can consider domains on the blacklist observed with known value. An example of a CRF model is in Figure 2.3.

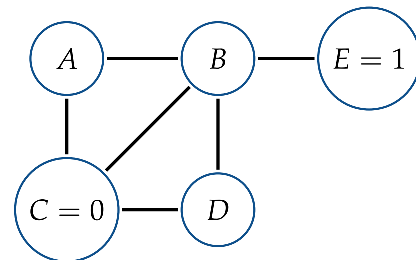


Figure 2.3: Conditional random field representing the same model as the model in Figure 2.2, however, this time values of variables C and E have been observed. A typical task now would be to infer $p(A, B, D \mid C = 0, E = 1)$ using observed information and relations between variables encoded into the graph.

2.2 Inference on graphical models

Once the system of random variables is represented as a graphical model where vertices correspond to single random variables, we can perform inference on this graph. The first classical use case is inferring a conditional distribution:

$$p(Y_1, Y_2, \ldots, Y_n, \mid X_1, X_2, \ldots, X_m) \tag{2.12}$$

of one or more random variables $Y_1, Y_2, \ldots, Y_n$ given some observations $X_1, X_2, \ldots, X_m$. This task is usually solved with Conditional random fields, but can be in fact generalized to finding a marginal probability $p(Y_1, Y_2, \ldots, Y_n, X_1, X_2, \ldots, X_m)$ if we fix probability distribution of the observed variables to Dirac delta functions. This task corresponds to finding the marginal probability in Markov random field, confirming that CRFs are only a special case of MRFs.

The second use case is finding a state in which the system is most likely to reside. This corresponds to solving following optimization task:

$$X_1^*, X_2^*, \ldots, X_n^* = \underset{X_1, X_2, \ldots, X_n}{\text{argmax}} \; p(X_1, X_2, \ldots, X_n) \tag{2.13}$$

For some subclasses of models, exact methods were developed, for instance in [Baum and Petrie, 1966] for *Hidden Markov models*, which are special instances of Bayesian networks, where some variables are observed and some are latent. However, in general graphs solving either of the aforementioned tasks is often computationally infeasible or even proved to be NP-complete and iterative approximation methods have to be employed.

Approximation methods used to infer unknown probabilities include *message-passing inference* [Pearl, 1988, Frey and MacKay, 1998, Kschischang et al., 2001], *graph cuts* [Kohli and Torr, 2005] and *Monte carlo Markov chains* [Wang et al., 2000]. In the next section, we describe message passing inference as it is the foundation of our method.

2.2.1 Message passing algorithms

Message passing algorithms is a family of algorithms which aims to infer unknown quantities on graphs by sending *messages* between graph nodes. Messages capture information local to parts of the graph and by propagating them through the graph it is possible to transfer this information to distant nodes as well. Adjusting local messages accordingly to incoming messages from other parts of the graph and repeating this process iteratively may lead to system converging to a stable state. We will briefly introduce the main algorithms that Message passing inference (MPI) consists of.

In order to explain the inner-workings of the algorithms and simplify the further discussion, we first need to introduce the concept of *factor graph* structure. Recalling back to the Section 2.1, every probability distribution can be broken into factors, from whose product the joint probability is computed. The joint probability distribution can be then computed as a product of these factors. the factor graph is constructed by removing all edges of the original graph, adding nodes corresponding to each factor to nodes already present in the graph and adding new edges according to the following rule. A new edge is added between a node representing a random variable and a factor node, whenever the factor as a function is parametrized by the random variable. This ensures that resulting graph is bipartite, first partite formed by random variable nodes and second partite by factor nodes. Markov random field can be easily turned into a factor graph because every potential function

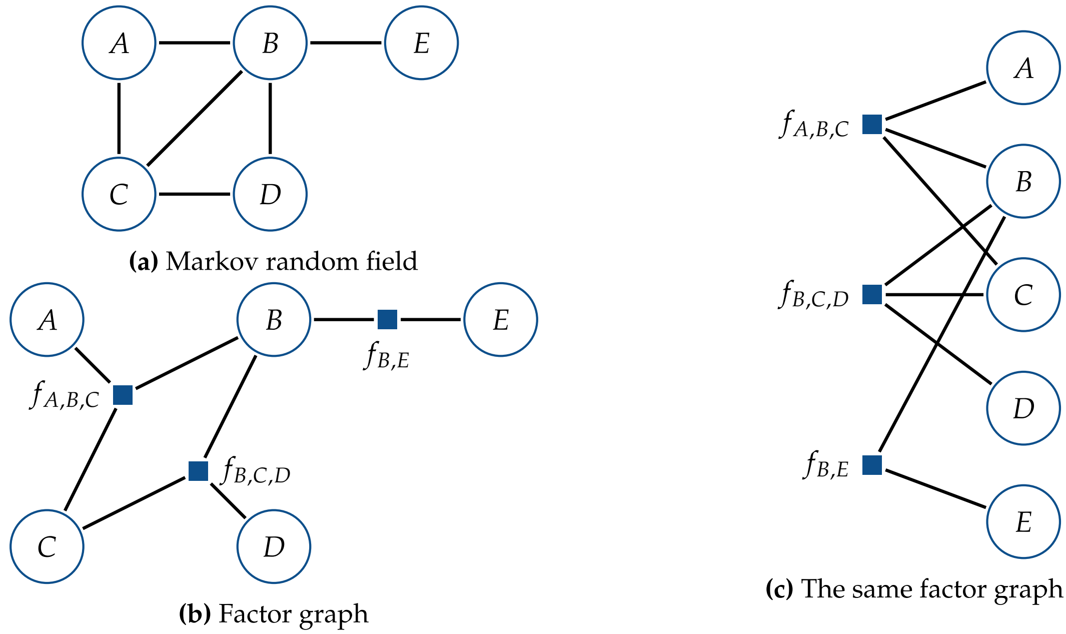


Figure 2.4: Transformation of Markov random field model from Figure 2.2 into factor graph. The same graph is drawn on the right hand side to demonstrate bipartitness of any factor graph. Comparing to (2.7) it can be seen that every factor and its neighbors corresponds to potential function of the neighbors.

operating on a clique corresponds to one factor. An example of such transformation can be seen in Figure 2.4. We will discuss one particular algorithm originally proposed in [Pearl, 1988] called *Belief propagation*. The work addressed only Bayesian networks, but we will describe its generalization to MRFs called the sum-product algorithm. This algorithm adopted the name of its predecessor and therefore is sometimes called Belief propagation as well.

2.2.2 Sum-product algorithm

The sum-product algorithm [Kschischang et al., 2001] tackles the problem of inferring marginal probabilities from the whole joint probability or conditional probability described (2.12) while maintaining efficiency. Every marginal probability of random variable X can be obtained by marginalizing over other random variables:

$$p(X_i) = \int_{(\mathbf{X} \setminus X_i)} p(\mathbf{X}) d(\mathbf{X} \setminus X_i) \tag{2.14}$$

$$= p(X_i) = \int_{X_1} \dots \int_{X_{i-1}} \int_{X_{i+1}} \dots \int_{X_n} p(\mathbf{X}) \, dX_n \dots dX_{i+1} \, dX_{i-1} \dots dX_1 \tag{2.15}$$

where $\mathbf{X} = (X_1, X_2, \dots, X_n)$ is a vector of all n random variables in the system. Also, we slightly abuse the notation used for the set difference to keep brevity. Therefore, $\mathbf{X} \setminus X_i$ represents a vector of random variables with the variable X_i left out. If Markov random field is a tree, it contains only cliques of cardinality two and therefore every potential function ψ is a function of two arguments and every factor in the factor graph has only two neighbors.

As a result, the joint probability the graph represents can be written as:

$$p(\boldsymbol{X}) = \prod_{f \in \mathcal{N}(X_i)} F_f(X_i, \boldsymbol{X}(f)) \tag{2.16}$$

Here, $\mathcal{N}(X_i)$ denotes the set of factors neigboring some random variable X_i in the factor graph, $\boldsymbol{X}(f)$ the set of random variables present in the subtree formed by the factor node f and $F_f(X_i, \boldsymbol{X}(f))$ is the product of all the factors in the subtree formed by the factor f. To illustrate this factorization we refer to Figure 2.5, which also gives a basic intuition about the notion of messages. Due to the conditional independence between some variables, the joint probability can be computed locally and the result can be 'sent' over to other parts of the graph.

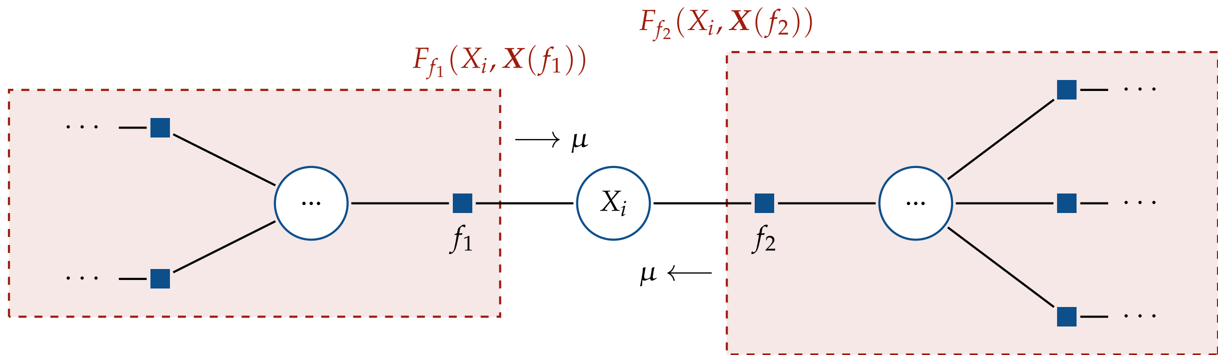


Figure 2.5: Factorization of a tree graph into a product of two terms. The values μ representing functions F_f evaluated at different subtrees can be viewed as messages sent from other parts of the tree computed locally and expressing belief about values of X_i.

Identity (2.14) can be then rewritten using (2.16) as:

$$p(X_i) = \int_{\boldsymbol{X} \setminus X_i} \prod_{f \in N(X_i)} F_f(X_i, \boldsymbol{X}(f))\, d(\boldsymbol{X} \setminus X_i) \tag{2.17}$$

for every random variable X_i. This factorization into functions of different part of graphs may not be possible in general graphs. *Belief propagation* [Pearl, 1988] was shown to work exactly in tree graphs and the algorithm can be directly derived from (2.17) and (2.16). In the case of graphs with loops, its variant called *Loopy belief propagation* [Frey and MacKay, 1998] can be used. Exact inference is not guaranteed, but this method performed reasonably well on various tasks.

As it turns out, integrands in (2.17) have many factors in common (this is given by the fact that probabilistic dependencies are only local in the graph). For the summation of many terms containing repeated members, we can use the identity $xy + xz = x(y + z)$ and thus save one operation by factoring out appropriate expressions. The sum-product algorithm leverages this observation and takes an approach similar to dynamic programming.

Messages passed between the factor node f and the random variable X_i and vice versa are recursively defined as:

$$\mu_{f\to X_i}(X_i) = \int_{\mathbf{X}_f} f(X_i, \mathbf{X}_f) \prod_{X_n \in \mathcal{N}(f)\setminus X_i} \mu_{X_n\to f}(X_n)\, d\mathbf{X}_f \tag{2.18}$$

$$\mu_{X_i\to f}(X_i) = \prod_{f_n \in \mathcal{N}(X_i)\setminus f} \mu_{f_n\to X_i}(X_i) \tag{2.19}$$

where $\mathbf{X}_f$ is a set of variables on which the factor f depends with the variable X_i left out. Here, we assumed that we work with an undirected graphical model and directly substituted potential functions ψ into factor functions f, however, the equations hold for any functions that appropriately factorize the joint distribution. In the case of discrete distributions, we apply summations instead of integrals. From these equations, it can be seen that for inferring the marginal probability of a random variable, the sum-product algorithm does nothing else than a marginalization over all possible values of all other variables and rearranging the summation in an efficient way. Let us emphasize that every message depends only on the value of the random variable it is sent to or sent from. Furthermore, provided all incoming messages are nonnegative, resulting message is nonnegative as well.

Informally speaking, a message $\mu(X_i)$ carries a belief in random variable X_i having a particular marginal value accumulated in other parts of the graph. Intuitively, when values of messages sent to the random variable are high, the marginal probability of the value is high as well:

$$p(X_i) \propto \prod_{f \in \mathcal{N}(X_i)} \mu_{f\to X_i}(X_i) \tag{2.20}$$

In this light, the messages can be viewed as likelihood of random variable having a particular marginal value.

2.2.3 Max-sum algorithm

The max-sum algorithm solves the task of maximally probable state of distribution defined in (2.13). We seek a state $\mathbf{X}^*$ such that:

$$p(\mathbf{X}^*) = \max_{\mathbf{X}} p(\mathbf{X}) = \max_{X_1} \max_{X_2} \ldots \max_{X_n} p(X_1, X_2, \ldots, X_n) \tag{2.21}$$

Because the probability can be again expressed as a product of factors that are common for many terms being maximized, we make similar observation as in the case of the sum-product algorithm, that is $\max(xy, xz) = x \max(y, z)$ if x is a constant, positive number. Following the same reasoning we could define messages just like in (2.18) and (2.19) performing maximization instead of marginalization, but in order to avoid underflow for small probabilities, the probability is transformed into the logarithmic domain. The messages are defined as:

$$\mu_{f\to X_i}(X_i) = \max_{\mathbf{X}_f} \left(\ln f(X_i, \mathbf{X}_f) + \sum_{X_n \in \mathcal{N}(f)\setminus X_i} \mu_{X_n\to f}(X_n) \right) \tag{2.22}$$

$$\mu_{X_i\to f}(X_i) = \sum_{f_n \in \mathcal{N}(X_i)\setminus f} \mu_{f_n\to X_i}(X_i) \tag{2.23}$$

The most probable value for a single variable can be computed from:

$$p(X) = \underset{X}{\operatorname{argmax}} \sum_{f \in N(X)} \mu_{f \to X}(X) \tag{2.24}$$

If some values of random variables in MRF are known, we can take it into account when computing messages coming from and into these variables and the inference procedure remains the same. We still have not addressed many decisions that need to be taken when designing message-passing algorithm, such as how to initialize all messages, whether to send all messages synchronously or adopt a different asynchronous schedule and how to normalize messages, as without normalization, the algorithm may diverge. However, this chapter was meant to give the reader some insight into important methods and techniques that have inspired our work, most importantly the notion of variables sending messages to each other, instead of describing them in full detail. For a thorough discussion, one may consult [Bishop, 2006].

Chapter 3

Method description

In this chapter, we define rigorously the task of classifying malware servers in computer networks, introduce our *Multiple instance neural inference* formalism for performing inference in general graphs and explain how the formalism leverages findings from the fields of message passing graph inference, neural network modeling and Multiple instance learning. We propose a one-step method for solving graph inference task using MINI formalism, present the advantages of the approach in comparison with the current methods and discuss possible extension into multi-step inference algorithm.

3.1 Task formulation

To frame the task formally, we are presented with an undirected graph without loops $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a finite set of vertices and is a set of edges (adjacency relation). Additionally, we consider a set of classes $\mathcal{C}$ and a 'ground-truth' function $C \colon \mathcal{V} \to \mathcal{C}$ assigning the true class to every vertex. This function is of course unknown, but we will use it to denote the class of some vertex. Given a small subset of vertices $\mathcal{V}' \subseteq \mathcal{V}$ and their respective values $C(\mathcal{V}')$, our goal is to infer remaining labels with high degree of precision.

During learning, one or more graphs with complete labels (i.e. $\mathcal{V}' = \mathcal{V}$) serve as a training test and at the test time another graph is fed to the algorithm, this time only a small subset of vertices $\mathcal{V}'$ is labeled. Our method outputs not only a class $c \in \mathcal{C}$, but also probability distribution over classes. Function C can be therefore viewed as a mapping from $\mathcal{V}$ to $\mathbb{R}^{|\mathcal{C}|}$.

Using the graphical model framework introduced in the last chapter we can also interpret the task from the probabilistic view. A set of vertices $\mathcal{V}$ becomes a set of random variables taking on values from $\mathcal{C}$. The joint probability $p(\boldsymbol{X})$ is encoded in the corresponding Markov random field and probabilistic relations between variables in $\mathcal{C}$ define the placement of edges. Given observations of values of some variables $\boldsymbol{X}_O \in \boldsymbol{X}$ we want to infer conditional marginals:

$$p(\boldsymbol{X} \setminus \boldsymbol{X}_O \mid \boldsymbol{X}_O) \tag{3.1}$$

for the remaining variables in the graph. Because of our focus on undirected graphs, from now on we abandon the general notation for graph factors f, used for the definition of messages in Section 2.2.2, and use potential functions ψ, described in Section 2.2, instead.

Computer security domain interpretation

In order to illustrate how this problem formulation translates to a real world use case, let us consider again the CS domain and in particular the task of classifying servers as malicious or benign. We can think of $\mathcal{V}$ as a set of all servers in the dataset, each server $v \in \mathcal{V}$ belonging to a class $c \in \{\textit{malicious}, \textit{benign}\}$. Graph adjacency relation $\mathcal{E}$ can be defined in multiple ways, in our case, we define two servers to be adjacent in the graph if any user connected to both of these servers. The existence of such communication channel depends on the servers themselves and their classes. If

we are given a record of network traffic in a company during a pre-defined time window, we can build a graph of servers by observing activities of users connected to the network. Given a blacklist of malicious servers (known to be almost surely incomplete), we would like to infer which servers in the network are likely malicious as well.

3.2 MINI formalism

Now we are ready to present the formalism itself. We begin by describing a hypothetical generative model that generates graphs we will learn from given parameters θ (from the parameter space Θ). Parameters θ can be thought of as a description of the system at the time of observation. Then we show that by assuming the Markov property to be true in our graphs we can only consider Markov blanket of the current node being inferred. This allows us to employ the powerful prior-art message passing inference framework to define predictors of messages sent between nodes. By viewing the problem from the MIL perspective we gain a new insight into the problem and define the predictors using neural networks.

3.2.1 Generative model

In order to better understand the problem, let us firstly describe a hypothetical generative model producing graphs that meet our requirements. The formal definition may seem intimidating by its complexity, but we shall see that it suitably describes the situation and its all-embracing nature will simplify the next step in MINI formalism.

Consider a finite set of vertices $\mathcal{X}$ which represent all random variables in the system. We define $\mathcal{G}$ as a space of all undirected graphs without loops representing a probability distribution on its vertices. Moreover, we assume that graphs in $\mathcal{G}$ have the Markov property.

We define a probability space $(\mathcal{G}, \mathcal{A}, P_\theta)$ where $\mathcal{G}$ is a space of all graphs built on top of $\mathcal{X}$ fulfilling the aforementioned requirements, $\mathcal{A}$ is a σ-algebra on $\mathcal{G}$, and $P_\theta\colon \mathcal{A} \to [0, 1]$ is a probability measure. Next, we define a function $\mathcal{F}$ which maps parameters $\theta \in \Theta$ to every set of graphs in $\mathcal{A}$. We also require that for some fixed θ the probability measure P_θ satisfies $P_\theta(a) = 0$, whenever $\mathcal{F}(a) \neq \theta$. Because we consider $\mathcal{X}$ finite, the space of all graphs $\mathcal{G}$ is finite as well, therefore we can put $\mathcal{A}$ equal to $\mathcal{P}(\mathcal{G})$ and restrict our attention only to probabilities of single graphs. Here, $\mathcal{P}(\mathcal{G})$ is a powerset of the set $\mathcal{G}$. Sampling from this probabilistic space provides us with graphs we shall observe and the constraint put on P_θ makes sure that all sampled graphs come from the same situation θ.

Computer security domain interpretation

Returning to our example from the CS domain, we may interpret parameters θ as a 'situation' in the network, including influential outer factors such as time and internet trends, characteristics of users connected to the network and most importantly, properties of domains. The probability in (3.1) we aim to infer is therefore parametrized by θ as well and can be rewritten as:

$$p(\boldsymbol{X} \setminus \boldsymbol{X}_O \mid \boldsymbol{X}_O\,; \theta) \tag{3.2}$$

We are interested in the part of parameters specifying labels of servers. We assume that parameters θ are 'stationary', in other words, they do not change much during the time of observation. As a result, all snapshots of communication follow the distribution that is parametrized by θ and in the following text, we introduce a way of inferring conditional probability in (3.2) from the graphs in $\mathcal{G}$.

Markov property

By assuming that the graphs in $\mathcal{G}$ have the Markov property, we gain a way of factorizing the joint probability every graph represents. Hammersley–Clifford theorem [Hammersley, 1971] says that such graphs can be turned into Markov random fields and consequently factorized in the same way as in (2.9):

$$p(\boldsymbol{X}) = \frac{1}{Z} \prod_{K} \psi_K(\boldsymbol{X}_K\,;\,\theta) \tag{3.3}$$

or in other words, the probability the graph represents can be written as a product of some functions operating on maximal cliques $\boldsymbol{X}_K$ in the graph. We define potential functions ψ_K to be parametrized by θ, the vector of generative model parameters. Here, Z is a constant partition function.

As the next step we can derive a formula for following marginal conditional probability:

$$\begin{aligned}
p(X_i \mid \mathbf{X} \setminus X_i) &= \frac{p(\mathbf{X})}{p(\mathbf{X} \setminus X_i)} = \frac{p(\mathbf{X})}{\int_{X_i} p(\mathbf{X})\, dX_i} = \frac{\frac{1}{Z} \prod_K \psi_K(\mathbf{X}_K\,;\,\theta)}{\int_{X_i} \frac{1}{Z} \prod_K \psi_K(\mathbf{X}_K\,;\,\theta)\, dX_i} \\
&= \frac{\frac{1}{Z} \prod_{\mathbf{X}_K \not\subseteq \mathbf{MB}(X_i)} \psi_K(\mathbf{X}_K\,;\,\theta) \prod_{\mathbf{X}_K \subseteq \mathbf{MB}(X_i)} \psi_K(\mathbf{X}_K\,;\,\theta)}{\frac{1}{Z} \prod_{\mathbf{X}_K \not\subseteq \mathbf{MB}(X_i)} \psi_K(\mathbf{X}_K\,;\,\theta) \int_{X_i} \prod_{\mathbf{X}_K \subseteq \mathbf{MB}(X_i)} \psi_K(\mathbf{X}_K\,;\,\theta)\, dX_i} \\
&= \frac{\prod_{\mathbf{X}_K \subseteq \mathbf{MB}(X_i)} \psi_K(\mathbf{X}_K\,;\,\theta)}{\int_{X_i} \prod_{\mathbf{X}_K \subseteq \mathbf{MB}(X_i)} \psi_K(\mathbf{X}_K\,;\,\theta)\, dX_i} = \frac{p\left(\{X_i\} \cup \mathbf{MB}(X_i)\right)}{p(\mathbf{MB}(X_i))} \\
&= p(X_i \mid \mathbf{MB}(X_i))
\end{aligned} \tag{3.4}$$

In the derivation, we used Bayesian and marginalization rules, together with the definition of the Markov blanket in Markov random fields. This result is in fact nothing else than a specific case of identity (2.5) applied on graphs from $\mathcal{G}$. The graph inference task (3.2) is hard and suggests nothing about the way the task can be grasped, let alone devising an exact solution, which would be feasible. This result, however, motivates the use of Message passing inference framework, a prior-art method for inference in MRFs. One may notice the similarity between the probability of X_i given its Markov blanket and the formula for messages in message passing algorithms, which also contains only the close neighborhood of variables. Another argument for employing MPI algorithms for inference is the fact that the max-sum algorithm approximates the maximally probable state of the system of variables. The sought probability is exactly the same as the definition of our task in (3.2), with the only difference that in our task we observe some values, therefore the computation of messages would be modified accordingly.

We showed that message passing inference algorithms are applicable to graphs generated by the generative model. The definition of MINI formalism continues with the way one may learn the messages in MPI instead of computing them using fixed closed-form formulas.

3.2.2 Learning the messages

From our discussion in the previous sections it follows that thanks to the properties of graphs, we can reason about the problem in MPI framework. Nevertheless, the framework seems constraining because of the necessity to provide formulas for potential functions $\psi_K(\mathbf{X}_k)$ in a closed form. One reasonable solution is to attempt to learn the definition of $\psi_K(\mathbf{X}_k)$ as has been done in previous work [Lin et al., 2015a, Zheng et al., 2015, Schwing and Urtasun, 2015]. The authors attempted to learn potential functions with Stochastic gradient descent (SGD) by optimizing a global objective on the graph and predicting unknown variables by performing message-passing inference. This approach has two drawbacks – firstly, running message-passing algorithm at each iteration of SGD is expensive on large data and secondly, a strong bias is introduced by assuming message equations to take the exact form as described in (2.18) and (2.19). We mentioned in the last chapter that the sum-product algorithm is proven exact in tree graphs, however for general graphs with loops, there is no guarantee apart from good experimental results.

As discussed in [Ross et al., 2011], comparing (2.19) and (2.20), variable-to-factor message $\mu_{X\to f}$ can be thought of as a marginal value (2.20) of X when factor f is removed from a graph. By substituting (2.18) into (2.19) we obtain:

$$\mu_{X_i\to f}(X_i) = \prod_{f'\in\mathcal{N}(X_i)\setminus f} \mu_{f'\to X_i}(X_i) \tag{3.5}$$

$$= \prod_{f'\in\mathcal{N}(X_i)\setminus f} \int_{\mathbf{X}_{f'}} f'(X_i, \mathbf{X}_{f'}) \prod_{X_j\in\mathcal{N}(f')\setminus X_i} \mu_{X_j\to f'}(X_j) \tag{3.6}$$

Thus, messages $\mu_{X\to f}$ leaving a variable X and being sent to factor f can be considered a marginal values over classes (classification) of variable X, dependent only on messages from variables not connected to X by factor f. This means that the sum-product algorithm is classifying variables in the graph by computing a sequence of classifications and can be viewed as a function we recursively apply to each variable that operates on local beliefs obtained from the variable's neighborhood. The same substitution can be applied to (2.20):

$$p(X_i) \propto \prod_{f\in\mathcal{N}(X_i)} \mu_{f\to X_i}(X_i) \tag{3.7}$$

$$\propto \prod_{f\in\mathcal{N}(X_i)} \int_{\mathbf{X}_f} f(X_i, \mathbf{X}_f) \prod_{X_j\in\mathcal{N}(f)\setminus X_i} \mu_{X_j\to f}(X_j) \tag{3.8}$$

and exactly the same applies for the formulas of the max-sum algorithm. The reasoning above demonstrates that any other predictor can be used in place of (3.6) and the message-passing algorithm is just a special case with factors ψ as learnable parameters. Depending on the application, this framework gives us freedom in carefully selecting a predictor meeting any requirements in terms of capacity, speed or other properties. Using different class of predictors will lead to different classes of inference procedures. Following this reasoning, we may even replace a generative probabilistic model with a discriminative model, as only discriminating between classes is usually easier than building a full probabilistic model, and we are primarily interested in predicting the labels. Inference on graphs becomes a sequence of predictions of probability distributions on variables made by the predictor functions operating on inferred values from the last step. The decision to learn messages

instead of the potential functions also lowers the dimension of model's output from $|\mathcal{C}|^a$, where a is the arity of the potential function, to $|\mathcal{C}|$ and consequently reduces the number of model's parameters significantly. Also learning messages does not require performing inference procedure after every step of gradient descent as it is the case with potential functions.

This reasoning obviously opens a variety of directions to explore. It is possible to focus on optimizing predictors such that inference procedure works well over several steps by propagating important knowledge throughout the graph. Nonetheless, in this work, we are focused mainly on devising a high-quality predictor that can infer accurate values in a small number of steps. We performed only one step in our experiments and indeed, this may result into inference being inaccurate due to the lack of information in the close neighborhood. This issue is discussed further in the last section of this chapter. In the following section, we define the class of predictor functions used in the formalism and the details of their implementation.

3.2.3 Multiple instance learning predictor

One of the main challenges of designing such predictor lies in a variable number of neighbors of the graph variables. For instance, in the computer vision domain [Lin et al., 2015b, Ross et al., 2011] the representation of an image or a 3D point cloud as a graph is rather straightforward – single pixels (or superpixels) represent graph vertices, and their spatial relations are readily encoded into the graph by adding an edge between each pair of directly neighboring pixels or pixels lying close together. The number of neighbors of a vertex in the graph and consequently the input dimension of the predictor is therefore controllable and known beforehand. Unfortunately, this is not the case in many other domains. To give an example of such domain, we again resort to the CS domain in which computer network and communication over it is inherently represented by a graph, however, no assumption can be made about its shape or density. Furthermore, many machine learning algorithms demand a fixed size vector of values as their input. This would require every vertex in the graph to be of constant degree, which is hardly ever the case, and therefore a space of possible potential predictors is tiny. Another possible approach is to reduce or increase the input dimensionality as needed by some ad hoc techniques.

In the previous section, we identified message passing inference as the specific case of the sequential inference on graphs where probability distributions on the variables are obtained from the predictor function operating on distributions inferred in the previous step. This frees us from having to think about predictors in any concrete form like in message passing algorithms and reason about the inference on the higher level of abstraction. One of the key ideas of MINI formalism is to view the situation from the perspective of Multiple instance learning and attempt to find the desired predictor using the MIL framework. We identify each vertex (variable) in the graph as a bag and vertices in its neighborhood as instances in the bag. One can then attempt to define a model to learn the predictor function acting on the bag and optimize the model so that predictions are accurate. This approach copes with every possible number of neighbors in a natural way and is suitable for describing every relation in the graph with an arbitrary set of features. Note that this definition of a bag makes sure that all the variables from the Markov blanket of the current vertex are present in the bag and vice versa.

Formally, we define a *bag* b_u corresponding to a node $u \in \mathcal{V}$ in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as the set of all the neighbors of u, i.e., $b_u = \{v \in \mathcal{V} \,;\, \{u, v\} \in \mathcal{E}\}$. We then denote a space of all such bags as $\mathcal{B}$ and represent each relation between u and some v by a feature vector of fixed size $\boldsymbol{\alpha}_u(v) \in \mathbb{R}^n$. We will call the mapping $\boldsymbol{\alpha}_u$ the *feature mapping* in the context of the bag b_u. The highly variable size of the bag $|b_u|$ (number of neighbors of v) demands utilization of MIL techniques, namely the *embedded-space* paradigm [Chen et al., 2006]. Feature vectors of instances are firstly embedded into a Euclidean space of dimension m by applying a set of m nonlinear transformations $\phi_i \colon \mathbb{R}^n \to \mathbb{R}$:

$$\boldsymbol{\phi}(\boldsymbol{\alpha}_u(v)) = (\phi_1(\boldsymbol{\alpha}_u(v)), \phi_2(\boldsymbol{\alpha}_u(v)), \ldots, \phi_m(\boldsymbol{\alpha}_u(v))) \tag{3.9}$$

Subsequently, we aggregate all embedded instances $\boldsymbol{\phi}(\boldsymbol{\alpha}_u(v))$ into one vector with r functions $g_j \colon \mathbb{R}^{|b| \times m} \to \mathbb{R}^m$ ($r \geq 1$) and by concatenating every g_i we obtain:

$$\boldsymbol{g} = (g_1, g_2, \ldots, g_r) \colon \mathbb{R}^{|b| \times m} \to \mathbb{R}^{rm} \tag{3.10}$$

The composition of $\boldsymbol{\alpha}_u(v)$, $\boldsymbol{\phi}$ and $\boldsymbol{g}$ defines a mapping $\boldsymbol{\psi}$ from the space of all bags $\mathcal{B}$ to $\mathbb{R}^m$:

$$\boldsymbol{\psi}(b_u) = \boldsymbol{g}(\{\boldsymbol{\phi}(\boldsymbol{\alpha}_u(v))\}_{v \in b_u}) \tag{3.11}$$

The key observation is that the range of this composition is a Euclidean space $\mathbb{R}^m$, in other words it maps every bag $b \in \mathcal{B}$ to a vector of fixed size. Finally, we define a classificating function $\boldsymbol{h} \colon \mathbb{R}^{lm} \to \mathbb{R}^{|\mathcal{C}|}$ that assigns every aggregated bag $\boldsymbol{\psi}(b)$ a probability of b being assigned a class. The whole model $f \colon \mathcal{V} \to \mathbb{R}^{|\mathcal{C}|}$ can be summarized as follows:

$$f(u) = \boldsymbol{h}(\boldsymbol{\psi}(b_u)) = \boldsymbol{h}(\boldsymbol{g}(\{\boldsymbol{\phi}(\boldsymbol{\alpha}_u(v))\}_{v \in b_u})) \tag{3.12}$$

This general model definition allows for wide options for mappings $\boldsymbol{\phi}$, $\boldsymbol{g}$, and $\boldsymbol{h}$. For instance, the classificating function $\boldsymbol{h}$ can be chosen from the variety of off-the-shelf predictors taking a vector of fixed size as an input, since the composition of the other two functions $\boldsymbol{\psi}$ (3.11) does all the heavy lifting of embedding the bag into a space of given dimension. The other two functions $\boldsymbol{\phi}$ and $\boldsymbol{g}$ may be designed arbitrarily as well, the only requirement is to comply with the aforementioned embedded-space paradigm. The last part of the MINI formalism is to specify these functions and define their implementation.

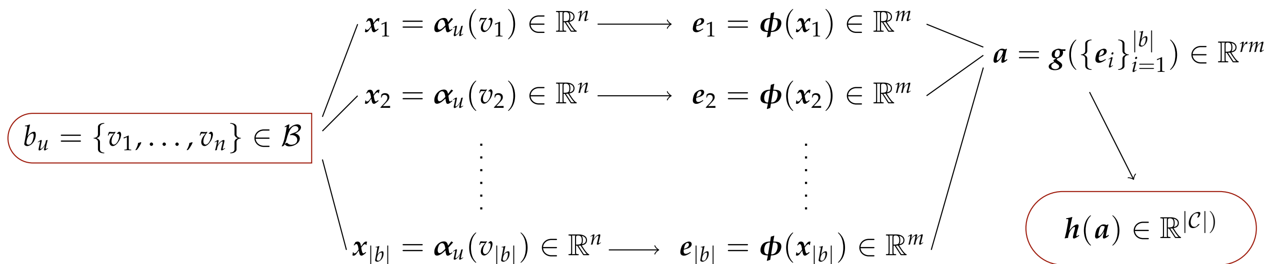


Figure 3.1: A picture of the whole model. Given a feature vector $\boldsymbol{x}$ of the length n of every instance v in the bag b_u corresponding to vertex u in the graph, we firstly embed each feature vector through $\boldsymbol{\phi}$ into space $\mathbb{R}^m$, aggregate all embeddings, and then feed the aggregation $\boldsymbol{a}$ through function $\boldsymbol{h}$ to obtain the prediction $f(b_u)$.

3.2.4 Neural networks function approximators

We have already mentioned the complexity of the task and that we presumably cannot hope that either potential functions factorizing the distribution or formulas for message passing inference can be expressed in the closed form. This motivates the use of neural networks, regarded as universal function approximators. As proposed in [Pevný and Somol, 2017,Stiborek et al., 2017], we implement both functions $\boldsymbol{\phi}$ (3.9) and $\boldsymbol{h}$ (3.12) as one or more (feedforward) neural network layers. Every instance from the bag is firstly fed through layer(s) embodying function $\boldsymbol{\phi}$ (note, that the layers use the same weights for every instance) and after applying the aggregation $\boldsymbol{g}$ (3.10), classification with the function $\boldsymbol{h}$ is performed by another set of feedforward layers. If $\boldsymbol{g}$ is at least peace-wise differentiable, full model becomes differentiable. This has the advantage that we train the embedding with respect to labels of bags and optimization by a plethora of gradient descent methods is possible.

Aggregating functions

In this subsection, we introduce few useful aggregating functions, suitable for use in place of the aggregating function g described in (3.11). Note that every function is (at least piecewise) differentiable, therefore the whole model remains differentiable as well.

Max

The max aggregating function is defined component-wise:

$$g^{(i)}_{max}(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m) = \max_{j \in \{1,\ldots,m\}} \boldsymbol{x}^{(i)}_j \tag{3.13}$$

As its derivation we obtain:

$$\frac{\partial g^{(i)}_{max}}{\partial \boldsymbol{x}^{(k)}_j} = \begin{cases} 1 & \text{if } i = k \text{ and } \forall j' : \boldsymbol{x}^{(i)}_j \geq \boldsymbol{x}^{(i)}_{j'} \text{ or } j \leq j' \\ 0 & \text{otherwise} \end{cases} \tag{3.14}$$

Maximum comes in handy when the label of the whole bag may be determined based on a single instance. For example, if many users who communicate with a server connected to a blacklisted server in the past, the server is very likely to be malicious as well. However, despite this strong evidence, this may be pure coincidence, but it is beneficial to extract this information from bag and let the classificating function decide.

Mean

The mean aggregating function is defined as:

$$g^{(i)}_{mean}(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m) = \frac{1}{m} \sum_{j=1}^{m} \boldsymbol{x}^{(i)}_j \tag{3.15}$$

this function simply averages over its inputs. Deriving the function we obtain:

$$\frac{\partial g^{(i)}_{mean}}{\partial \boldsymbol{x}^{(k)}_j} = \begin{cases} 1/m & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} \tag{3.16}$$

As opposed to g_{max}, this functions express a quantitative property of the whole bag.

Weighted mean

The weighted mean is a function similar to g_{mean} function, the only difference is that we must provide a weight for each of its inputs:

$$g_{wm}^{(i)}(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m, \boldsymbol{w}) = \sum_{j=1}^{m} \boldsymbol{w}^{(j)} \boldsymbol{x}_j^{(i)} \Big/ \sum_{j=1}^{m} \boldsymbol{w}^{(j)} \tag{3.17}$$

Its derivative is:

$$\frac{\partial g_{wm}^{(i)}}{\partial \boldsymbol{x}_j^{(k)}} = \begin{cases} \boldsymbol{w}^{(k)} \Big/ \sum_{j=1}^{m} \boldsymbol{w}^{(j)} & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} \tag{3.18}$$

This function can be used in the case when for some reason we want to assign a higher priority to some instances in mean computation.

Concatenation

Given two aggregating functions $\boldsymbol{g}_1$ and $\boldsymbol{g}_2$ each outputting a vector in $\mathbb{R}^m$ it is also possible to concatenate the result of both into a new function $\boldsymbol{g} = (\boldsymbol{g}_1, \boldsymbol{g}_2)$ and produce a vector from $\mathbb{R}^{2m}$. This results into the extraction of information of different types for the classification layer of the model.

Other aggregating functions

Due to the differentiability being the only requirement on the aggregating function, one can see that the space of all eligible aggregating functions is huge. In fact, this function can be designed from scratch with respect to the task being solved. Even the use of differentiable functions with learnable parameters is possible as long as the whole model remains fully differentiable. The idea to learn parameters of some aggregating function is not new and indeed, it has been studied for example in [Gulcehre et al., 2014], where authors discussed learning parameter p in L_p norm, or in [Pevný and Nikolaev, 2015], where authors learned parameters determining the shape of a bell curve.

Example

To conclude the description of the method, we demonstrate a simple example of how such model can be defined in concrete terms. We define $\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{W_\phi x}$ as a one-layer neural network with parameters $\boldsymbol{W_\phi} \in \mathbb{R}^{m \times n}$ and the identity activation function (applied coordinate-wise). For the aggregation, we take a concatenation of max and mean functions $\boldsymbol{g} = (\boldsymbol{g}_{max}, \boldsymbol{g}_{mean})$ and for the classification function we pick another one-layer neural network $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{\sigma}(\boldsymbol{W_h x})$ with parameters $\boldsymbol{W_h} \in \mathbb{R}^{2m \times |\mathcal{C}|}$ and the softmax activation function σ. Suppose a bag $b_u = \{v_i\}_{i=1}^{l}$ of size l. The whole model (function $\boldsymbol{f}$ (3.12)) consists of first taking the feature vector $\boldsymbol{\alpha}_u(v_j)$ of the j-th instance and computing its mapping through function $\boldsymbol{\phi}$ by multiplying it with matrix $\boldsymbol{W_\phi}$. Each instance mapping in $\mathbb{R}^m$ then acts as the input to the aggregating function, that compresses a set of l vectors from space $\mathbb{R}^m$ to a single vector from $\mathbb{R}^{2m}$. The last step is to multiply this vector by $\boldsymbol{W_h}$, and apply the softmax function on the resulting vector to obtain class distribution.

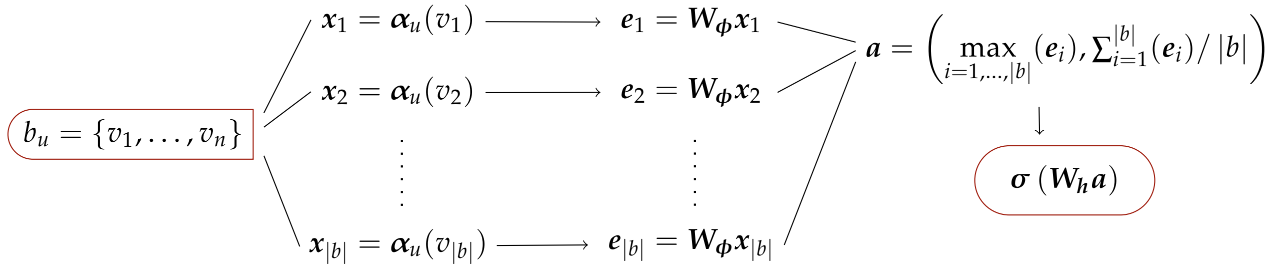


Figure 3.2: A picture of the specific model. This is the same model as the model depicted in Figure 3.1 with all specific mappings filled in except the feature mapping α_u (which is discussed in the next chapter). Functions max and σ are applied element-wise and $(\cdot, \cdot)$ means vector concatenation.

If we define the components of g to operate row-wise on a matrix, where input vectors are in columns, we can construct a matrix $X_{b_u} = [\alpha_u(v_1), \alpha_u(v_2), \cdots, \alpha_u(v_l)] \in \mathbb{R}^{n \times l}$ representing the bag b_u by putting feature vectors of bag instances into its columns. The function f defining our model is then defined as the composition of all the pieces:

$$f(b_u) = \sigma\left(W_h\left(g_{max}\left(W_\Phi X_{b_u}\right), g_{mean}(W_\Phi X_{b_u})\right)\right) \tag{3.19}$$

Instance representation

One of the big advantages of using the model described above is having a free hand in defining the mapping α_u associated with a bag b_u. The big benefit is that this mapping can take into account not only a neighboring vertex v but also the vertex u and even the relation between u and v, therefore available knowledge captured in the graph is effectively employed. Moreover adding external additional features results in only a little increase of computing cost, thus external knowledge about vertices is easily injected into the model.

Returning to the example of the edge in the graph between two servers s_1 and s_2 present when existing user connected to both of them, we may utilize only known information extracted from the graph about s_1 or s_2. This may include the degree distributions or other graphical features. However, thanks to the flexibility of the formalism we can also include information about the relation between s_1 and s_2 itself and external knowledge about servers s_1 or s_2, which have nothing in common with the graphical interpretation, but would complement the set of features well. The mapping α_u evidently depends on our goal and the domain we are working in, and the specific mapping we use for the task of classifying servers is discussed in detail in the next chapter.

3.2.5 Recapitulation

To summarize, we began with stationary parameters θ describing a ‘situation’ and introduced a generative model producing graphs that we assume to have Markov property. This assumption allowed us to employ the message passing inference framework and by inspecting the message definitions we redefined the inference procedure as a sequential classification process performed by a predictor acting on the neigborhood of a vertex. This way we avoid having to define formulas for either the potential functions or the messages in MPI inference. After that, we formulated the task of finding the predictor as a MIL problem, which enabled us to accommodate to an arbitrary degree distribution without compromises. The clever use of instance embedding functions and aggregating functions bridged the gap between a bag defined by a set of vectors representing its instances to one fixed size vector which we classify using another function. Moreover, the MIL predictor takes

all the information about the adjacency relation in the graph (or equivalently being in the bag of another vertex) into account, this includes both participants in the relation and the properties of edges. The use of neural networks to approximate unknown and rather complex functions defined in the model provided us with a way to learn these unknown functions instead of defining them, which would introduce a bias to the model. The MINI formalism brings a novel perspective to the problem of inference in a graph and by combining several different techniques we obtain a general, easily adaptable and extensible method.

3.2.6 Multi-step algorithm

Lastly, we outline how the method introduced here can be extended to perform multiple steps. As it turns out, the MINI formalism accommodates to every approach to multi-step inference we sketch here. Let us denote the number of inference steps performed by a method as s. An the beginning of this chapter, we made an assumption that all important information needed to make the decision is contained in the Markov blanket of a vertex. This is the same assumption that MPI framework requires and if it holds, that means that methods with $s = 1$ are sufficient. Good experimental results presented at the end of this work show that one step is almost always enough in CS domain, however, this assumption may not be true in general and information about vertices and relations between vertices from a greater distance than one may be needed. In such case, MPI methods perform more steps and this results into all crucial information being propagated to vertices in the Markov blanket and used for the inference afterward.

One possibility is to train a different predictor for every step. The first predictor would be trained using the initial values $p(X_i)$, where we set $p(X_i)$ to one if X_i is contained on the ground-truth blacklist and zero otherwise. After that, we would train the second predictor operating on values $p(X_i)$ inferred in the previous step. Repeating this procedure produces a sequence of s different predictors. Another reasonable approach is to use the same predictor for every step and after one step add bags with values inferred in the previous step to the dynamic dataset. This approach is not dependent on the number of steps s, however, the size of the dataset grows exponentially. The theorems about various guarantees both approaches offer are rigorously proven in [Ross and Bagnell, 2010] and [Ross et al., 2010].

Lastly, we present one more possible approach, applicable to smaller values of s, that immediately suggests itself thanks to the properties of MIL formalism. Let us consider a vertex u, its corresponding bag b_u and a vertex v from the bag, that is $v \in b_u$. What we could do is alter the definition of feature mapping $\boldsymbol{\alpha}_u$ to include the neighborhood of v (Markov blanket) as well and define it as $\boldsymbol{\alpha}_u(v\,;\,b_v)$. There are more ways of accomplishing this and one of them is to return to Section 3.2.3 and employ the very same definition of a model f, operating on the space of all vertices $\mathcal{V}$ and returning a vector from $\mathbb{R}^{|\mathcal{C}|}$, and then define feature mapping as $\boldsymbol{\alpha}_u(v\,;\,b_v) = \boldsymbol{\alpha}_u(v, f(b_v))$. We can easily make f return even higher dimensional vector attempting to capture more information and then modify the definition of the feature mapping $\boldsymbol{\alpha}_u$ itself to take a higher dimensional vector together with v as the input. This yet again demonstrates the flexibility of the formalism. Moreover, we could apply the same idea for the neighborhood of v as well. This results into a powerful multi-step layered model which makes use of all knowledge in the graph within s steps from the vertex u we consider. The illustration of this approach is in Figure 3.3.

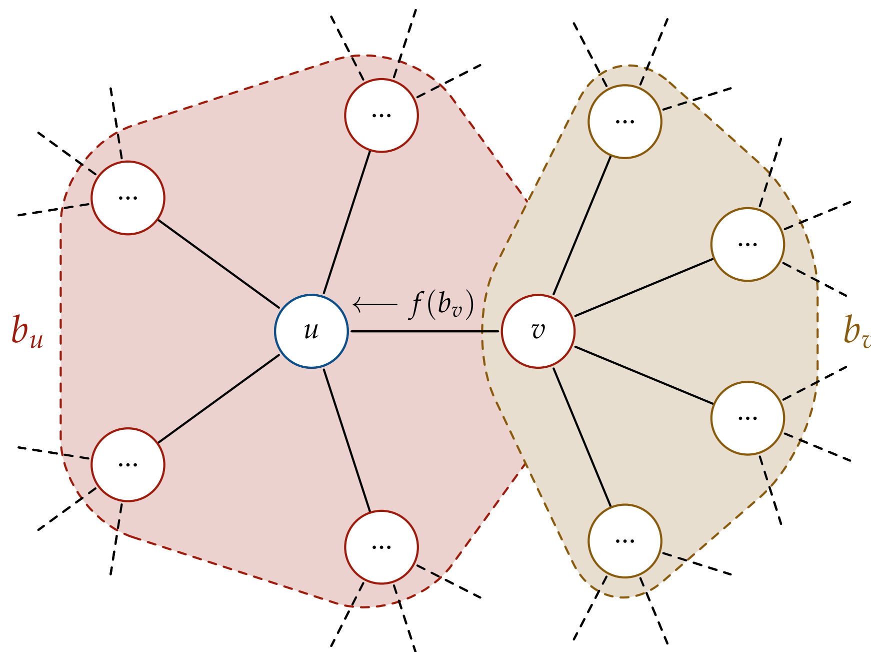


Figure 3.3: The depiction of how a multi-step layered model operates on a vertex u. Firstly, all vertices from the Markov blanket of u ($\{v_1, v_2, v_3, v_4, v_5\}$) are fed through a model similar to the one introduced for the one-step inference. The result $f(b_v)$ is included to the model operating on the bag b_u via a moddified feature mapping $\boldsymbol{\alpha}_u(v, f(b_v))$.

Chapter 4

Specific adjustments and techniques

The last chapter introduced our MINI formalism for performing graph inference in general terms. This chapter outlines several adjustments we made and techniques we devised to accommodate the universal approach to a more specific situation. We are mainly focused on the CS domain and huge, dense graphs.

4.1 Dealing with bipartite graphs

Until now we have considered our method to work for general graphs whose vertices represent our objects of interest. The formalism introduced in the last chapter enables us to construct an adjacency relation $\mathcal{E}$ on the set of the vertices (in other words where we put edges) in the graph arbitrarily. Thanks to the fact that our method works for any number of neighbors and instance mapping $\boldsymbol{\alpha}_u$ takes u, v and also relation $\{u, v\}$ into account, we are not constrained in any way in defining the the adjacency relation $\mathcal{E}$ to suit our needs. This is very convenient because as a result, we can only focus on defining this relation in a natural way and avoiding any loss of information about relations between objects of interest without a need to adapt to any rules. However, we are yet to specify how we obtained the graph of servers. In Section 3.1, we outlined how to construct the adjacency relation $\mathcal{E}$ on servers by putting an edge between two of them if any user communicated with both during a predefined time window. In this section, we describe this construction formally and in more detail as we decided to use it for further experiments. Nonetheless, we would like to emphasize that the formalism presented in the last chapter accommodates many more use cases and this is only one specific example. In the following text, we will sometimes use an alternative notation E instead of $\mathcal{E}$.

4.1.1 Construction of adjacency relation

Let us firstly define a *list of connections*. It is a set $L_{[t_1,t_2]} \subseteq \mathcal{U} \times \mathcal{D}$, where $\mathcal{U}$ and $\mathcal{D}$ are sets of all users and domains (servers) observed, and $[t_1, t_2]$ is a time interval during which the data was collected, usually termed *time window*. The presence of a tuple (u, d) in $L_{[t_1,t_2]}$ means that user u communicated with a domain d during the time window. Because the choice of the time window does not affect the next steps, we denote $L = L_{[t_1,t_2]}$.

To construct a graph of servers from a list of connections L, we begin with creating a bipartite graph, where one partite represents different users and the other different servers. We put an edge between a user and a server if the user connected to the server during the time window, or in other words, the connection between them is present in L. Next, we build a new graph of servers by taking servers present in L and putting an edge between two servers for each user who connected to both of the servers. This operation may add parallel edges into the resulting graph therefore to ensure that we obtain a simple graph, we merge all parallel edges between two servers into one carrying all information about every user connecting to both servers.

An example of this procedure is depicted in Algorithm 4.1. Note the striking similarity to Markov random field and its corresponding factor graph in Figure 2.4. In this way, we can very roughly view each user as a potential function operating on the servers that the user connected to.

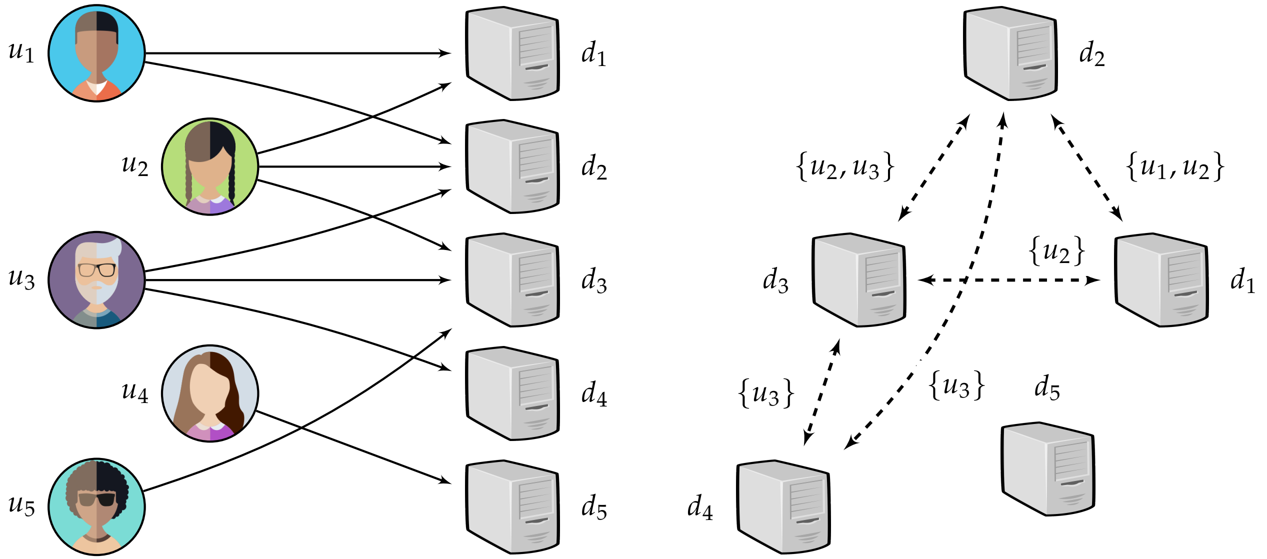


Figure 4.1: An example of a bipartite graph of users and servers and an adjacency relation in a graph of servers constructed from it. The connection list L used in this illustration is $\{(u_1, d_1), (u_1, d_2), (u_2, d_1), (u_2, d_2), (u_2, d_3), (u_3, d_2), (u_3, d_3), (u_3, d_4), (u_4, d_5), (u_5, d_3)\}$. We merged parallel edges into one to obtain a simple graph, but information about users is not lost. For example, there are two users u_2 and u_3 who connected to both domains d_2 and d_3, therefore we put two edges between d_2 and d_3 and merge them into one edge afterward. Also users u_4 and u_5 are not used in the graph of servers due to the fact that they connected to only one server and no relation may be therefore observed. The resulting relation is therefore $E = \{\{d_1, d_2\}, \{d_1, d_3\}, \{d_2, d_3\}, \{d_2, d_4\}, \{d_3, d_4\}\}$. Graphics by [Freepik, 2016].

Algorithm 1 Construction procedure of an adjacency relation E in a graph of servers. Servers D from $\mathcal{D}$ become vertices in the graph and edges E carry all the information.

1: **procedure** CONSTRUCT(L)
Input: $L \subseteq \mathcal{U} \times \mathcal{D} \leftarrow$ a list of connections
Output: $D \subseteq \mathcal{D}$, $E \subseteq \{\{u, v\}\,;\, u, v \in \mathcal{V}, u \neq v\} \leftarrow$ an adjacency relation, $I\colon E \to \mathcal{P}(\mathcal{U}) \leftarrow$ edges' information
2: $\quad U \leftarrow \{u \in \mathcal{U}\,;\, \exists d \in \mathcal{D}\colon (u, d) \in L\}$
3: $\quad D \leftarrow \{d \in \mathcal{D}\,;\, \exists u \in \mathcal{U}\colon (u, d) \in L\}$
4: $\quad E \leftarrow \{(d_1, d_2) \in D \times D\,;\, \exists u \in U\colon \{(u, d_1), (u, d_2)\} \in L\}$
5: $\quad$ **for** each edge $e = (d_1, d_2)$ in E **do**
6: $\qquad I(e) \leftarrow \{u \in U\,;\, \{(u, d_1), (u, d_2)\} \in L\}$
7: $\quad$ **end for**
8: $\quad$ **return** D, E, I
9: **end procedure**

From now on, when we mention *Bipartite graph (BG)*, we mean a bipartite graph of users and servers, and when we mention *Graph of servers* or a *Server graph (SG)*, we mean a graph derived from the bipartite graph of users and servers by the procedure defined above. Additionally, recall the mapping $\boldsymbol{\alpha}_u$ introduced in the previous chapter which returns a feature vector representing a relation between u and v, where u is a vertex in the graph and v is its neighbor. We can now rewrite this as $\boldsymbol{\alpha}_{s_1}(s_2) = f(s_1, s_2, e)$, where we renamed u and v to names s_1 and s_2, notation used with the graph of servers, e is an edge between s_1 and s_2 and f is any function. Going even further, we can replace the

edge e with information about users comprising it, we get $\boldsymbol{\alpha}_{s_1}(s_2) = f(s_1, s_2, \{u_1, u_2, \ldots, u_k\})$. This means that a mapping between instances in bags and feature vectors depends not only on both ends of the edge, but also on the properties of the edge, taking every user into account. In fact, this is just a concrete example of the remark we made earlier in Section 3.2.4 applied to the server graph.

To finish this section, there are some problems that must be addressed so that the procedure can be used in practice. The logical consequence of the definition of the adjacency relation in the previous graph is that some users with only one connection are omitted in the graph of servers and therefore the graph of servers may not be connected. On the contrary, users who are too active and create many connections introduce large cliques into the resulting graph. The next section describes how to deal with these side effects of adjacency relation construction.

4.2 Graph pruning

In a dense graph with n vertices, we are required to perform for each vertex u as many as $O(n) \cdot T(\boldsymbol{\alpha}_u(v))$ calculations to compute a feature vector $\boldsymbol{\alpha}_u(v)$ for every neighbor v (in other words computing representation of a bag b_u corresponding to the vertex u). The final cost of this operation depends on how we define the feature vector itself and its own computing cost $T(\boldsymbol{\alpha}_u(v))$, however many informative features derivable from graph may take linear time as well, accounting to the whole procedure of computing taking $O(n^2)$ time. This is clearly impossible to realize in reasonable time for graphs with a large number of vertices.

One possibility (besides of course increasing computational budget) is to use features available in constant time giving up information contained in features demanding more computational time, however, we used a different approach. The crucial observation is that in many domains vertices of highest degrees form a set of objects with different properties than the rest of the graph and it is harmless to erase them from the graph. To illustrate this claim with an example, in the CS domain, servers that are communicated with often, are usually benign. Because malicious servers form communities in network graphs we seek mainly the evidence of a relation to positive servers while making the decision. Besides that, servers with a high degree are in relation with the majority of servers in the graph. As a result, removing positive servers with a high degree from the graph significantly reduces computation demands while sacrificing little information. Instead of optimizing the latter factor in $O(n) \cdot T(\boldsymbol{\alpha}_u(v))$, we optimize the first one by re-

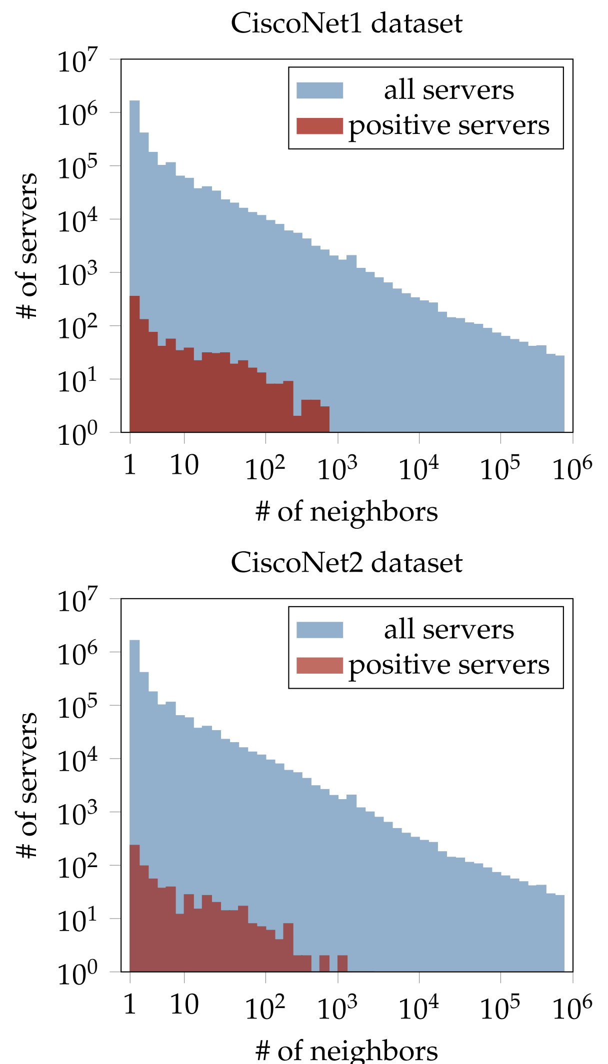


Figure 4.2: Degree distributions on a bipartite graph computed from datasets we worked with. Both X and Y axis are logarithmically scaled. Graphs support our claim that popular servers with high number of connections tend to be benign and can be removed from the graph.

moving some vertices and edges incident to them. The opposite extreme, servers with a low number of connections, were also removed from the graph. This is not inspired by any empirical knowledge of the domain, but rather by the fact that these servers offer only little irrelevant information to make use of and removing them from the dataset may be considered a means of cleaning noise in the data or preprocessing the data.

Another intuitive and empirically verified track to follow is removing users with few connections from the bipartite graph. If we consider a bipartite graph comprising domains and users in a predefined time window, some users may connect only to a server or two. These users are of little information value and it is safe to delete them. A printer that downloads software updates twice a year, showing no other network activity besides that, is an example of such uninteresting behavior. However, the same printer being much more active may signify a member of a DDoS botnet, which makes it interesting for us. To support our arguments above, we refer to Table B.1, which shows names of the most and least visited domains in one of our datasets. The most visited domains are evidently unlikely malicious. The same analysis can be done for the degrees of users, but we do not disclose user data here due to privacy concerns. Moreover, Figure 4.2, showing degree distributions in bipartite graphs, and Figure 4.3, showing how is the set of positive servers in the dataset influenced by pruning, provide empirical evidence for our claims. More insightful discussion on pruning network graphs is in [Jusko, 2017], which we consulted when setting the parameters for the pruning procedure. We present two pruning techniques that can be used to prune bipartite graphs of domains and users described in Section 4.1.

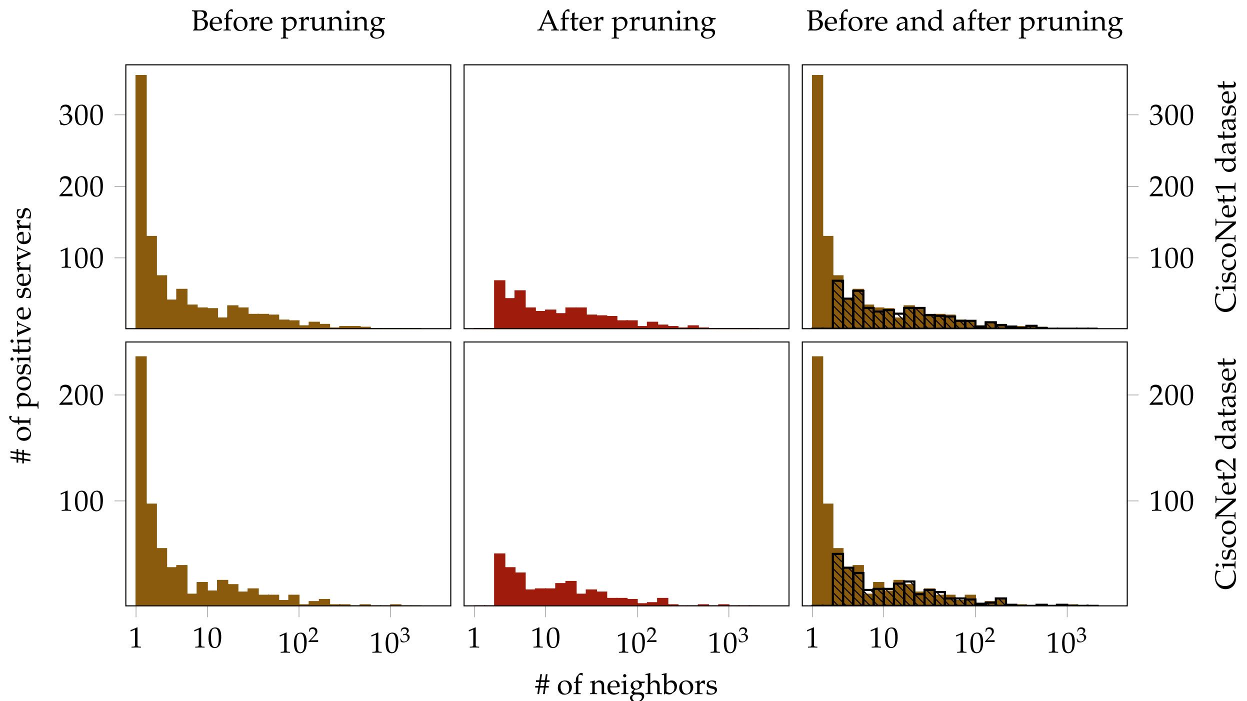


Figure 4.3: Influence of the graph pruning on the set of positive servers in the graph. Values used for lower and upper bounds are $\alpha_D = \alpha_U = 5$ and $\beta_D = \beta_U = 10^5$ and the procedure used was the *Complete pruning*. X axis is scaled logarithmically and Y axis linearly. In the first row there are graphs computed from the dataset CiscoNet1 and in the second row there are graphs computed from the dataset CiscoNet2. The first column shows the distribution of degrees of positive servers (in a bipartite graph of servers and users) before pruning procedure, the second column shows the distribution after the procedure is finished and the last column compares both in one figure. Graphs suggest that performing the pruning procedure properly removes noise in the form of servers with a low number of connections and we are left with clean data.

4.2.1 Complete pruning

The *Complete pruning* (Algorithm 2) procedure guarantees that every vertex in pruned graph satisfies the constraints on values of degrees. Note that under specific conditions this pruning procedure may prune the whole graph, but this does not happen in practice.

Algorithm 2 The Complete pruning technique. A graph is iteratively pruned by removing vertices that does not meet the constraints.

1: **procedure** COMPLETE PRUNING$(D, U, E, \alpha_D, \beta_D, \alpha_U, \beta_U)$
Input: $D \leftarrow$ a partite of domains, $U \leftarrow$ a partite of users, $E \leftarrow$ a set of edges, $\alpha_D, \beta_D \leftarrow$ a lower and an upper bound on domains' degrees, $\alpha_U, \beta_U \leftarrow$ a lower and an upper bound on users' degrees
Output: $D' \leftarrow$ a partite of domains remaining in the pruned graph, $U' \leftarrow$ a partite of users remaining in the pruned graph, $E' \leftarrow$ edges remaining in the pruned graph
2: $D_P \leftarrow \emptyset$, $U_P \leftarrow \emptyset$
3: **repeat**
4: $D' \leftarrow D \setminus D_P$
5: $U' \leftarrow U \setminus U_P$
6: $E' \leftarrow E \setminus (\{\{d,u\} \in E\,; d \in D_P\} \cup \{\{d,u\} \in E\,; u \in U_P\})$
7: $D_P \leftarrow D' \setminus \{d \in D\,; \alpha_D \leq \deg(d) \text{ and } \deg(d) \leq \beta_D\}$
8: $U_P \leftarrow U' \setminus \{u \in U\,; \alpha_U \leq \deg(u) \text{ and } \deg(u) \leq \beta_U\}$
9: $D \leftarrow D', U \leftarrow U', E \leftarrow E'$
10: **until** $D_P \cup U_P \neq \emptyset$
11: **return** D', U', E'
12: **end procedure**

4.2.2 Simple pruning

An alternative technique is the *Simple pruning* (Algorithm 3) that does not have this property and is on the one hand faster, but on the other hand offers no guarantee of lower bound on degree values in resulting graph. The two procedures presented here handle bipartite graphs, but can be easily modified to operate on general graphs or n-partite graphs.

Algorithm 3 The Simple pruning technique. Note that removing some vertices from a graph and edges incident to them may result into other vertices not meeting the requirements.

1: **procedure** SIMPLE PRUNING$(D, U, E, \alpha_D, \beta_D, \alpha_U, \beta_U)$
Input: $D \leftarrow$ a partite of domains, $U \leftarrow$ a partite of users, $E \leftarrow$ a set of edges, $\alpha_D, \beta_D \leftarrow$ a lower and an upper bound on domains' degrees, $\alpha_U, \beta_U \leftarrow$ a lower and an upper bound on users' degrees
Output: $D' \leftarrow$ a partite of domains remaining in the pruned graph, $U' \leftarrow$ a partite of users remaining in the pruned graph, $E' \leftarrow$ edges remaining in the pruned graph
2: $D_P \leftarrow D \setminus \{d \in D\,; \alpha_D \leq \deg(d) \text{ and } \deg(d) \leq \beta_D\}$
3: $U_P \leftarrow U \setminus \{u \in U\,; \alpha_U \leq \deg(u) \text{ and } \deg(u) \leq \beta_U\}$
4: $D' \leftarrow D \setminus D_P$
5: $U' \leftarrow U \setminus U_P$
6: $E' \leftarrow E \setminus (\{\{d,u\} \in E\,; d \in D_P\} \cup \{\{d,u\} \in E\,; u \in U_P\})$
7: **return** D', U', E'
8: **end procedure**

4.3 A mapping from instance relations to feature vectors

In the previous chapter, we defined our model as a composition of four mappings – the classificating function $\boldsymbol{h}$, the aggregating function $\boldsymbol{g}$, the instance-embedding function $\boldsymbol{\phi}$ and the feature mapping $\boldsymbol{\alpha}_u$. We have implemented $\boldsymbol{h}$ and $\boldsymbol{g}$ as neural networks and discussed prospective aggregating functions, however, the mapping from a relation between two instances to its corresponding feature vector is yet to be explained. We have already stressed the importance of extracting useful information from the graph while keeping the computing cost low, therefore every feature described here requires only linear time at worst. Consider a bag b_u corresponding to a vertex u in the graph of servers and a vertex v, one of its neighbors. Speaking in MIL terms, we treat v and its relation represented by an edge $\{u, v\}$ in the graph of servers as an instance in b_u, according to our definition from the last chapter. Recall that vertices u and v in a graph of servers represent domains and edge $\{u, v\}$ carries information about every user who connected to both u and v. Because some features may be more naturally interpreted in a bipartite graph (BG) than a server graph (SG) we include this interpretation to Table 4.2.

feature name	property of	SG interpretation	BG interpretation
# of neighboring servers	u	$\lvert\mathcal{N}_{SG}(u)\rvert$	#of vertices reachable from u by a trail of length 2
# of connected users (u)	u	# of users stored in edges incident to u	$\lvert\mathcal{N}_B(u)\rvert$
# of connected users (v)	v	# of users stored in edges incident to v	$\lvert\mathcal{N}_B(v)\rvert$
user intersection	$\{u, v\}$	amount of information about users saved in $\{u, v\}$	$\lvert\mathcal{N}_B(u) \cap \mathcal{N}_B(v)\rvert$
user union	$\{u, v\}$	amount of information about users saved in all edges incident to u or v	$\lvert\mathcal{N}_B(u) \cup \mathcal{N}_B(v)\rvert$
Jaccard index [1]	$\{u, v\}$	-	$\lvert\mathcal{N}_B(u) \cap \mathcal{N}_B(v)\rvert / \lvert\mathcal{N}_B(u) \cup \mathcal{N}_B(v)\rvert$
preferential attachment [2]	$\{u, v\}$	-	$\lvert\mathcal{N}_B(u)\rvert\,\lvert\mathcal{N}_B(v)\rvert$
one-hot class	v	-	-

Table 4.2: Table of features used in our experiment, where $\mathcal{N}_B(u)$ and $\mathcal{N}_{SG}(u)$ denote a set of neighbors of u in the bipartite and the server graphs respectively. Note that we employ knowledge about both endpoints of the edge between u and v as well as the relation $\{u, v\}$. We also include a one-hot encoded class of v in the features as it is important knowledge.

To make the input values more suitable for a neural network, we also applied the transformation $x \to \log(1 + x)$ on the features in every row except the Jaccard index and the one-hot class as they are already located in the interval $[0, 1]$. As the last step, we concatenate all values into a vector, obtaining a result of $\boldsymbol{\alpha}_u(v)$.

[1] [Jaccard, 1902]
[2] [Liben-Nowell and Kleinberg, 2003]

Note that in our task servers are both bags (when we infer the server's class) and instances (when the server appears in the neighborhood of server being considered). This allowed us to also include a class of the neighbor v into features and in the computer security domain, it is really important to know the class of the neighbor. For the first step of the inference procedure, we can consider tips from the blacklist as malicious and every other server as benign. In the following steps, we use predictions inferred in the previous steps (see Section 3.2.6). In this work, however, we only discuss one-step inference. Also, the way how features are defined hints, that working with both the bipartite graph of servers and users and the server graph enables us to use a set of features that would be really hard to obtain or even interpret in only one representation. This demonstrates the strength of this approach as we can keep both graphs in the memory, perform inference on the server graph, and at the same time compute some features from the bipartite graph. Another possibility is to keep only one of these graph in memory and computing the desired features on demand. This typically results into a lower memory requirements, but prolongs computing time.

It is relevant to emphasize that the features described above all represent information extracted only from the graph (or more precisely, the Markov blanket of the current node u) and no external knowledge about servers or relations is used. To give the example, we could describe every server u with its domain name, IP address, nameservers used or *Whois database* query result. Each relation $\{u, v\}$ can be associated with any information about users participating in the relation or frequency, packets' size and content of the communication. We omit this undoubtedly informative kind of features on purpose, in order to fully test the model's ability to infer labels only from the information contained in the graph and to investigate how far can we go in this direction. Therefore, we knowingly isolate the model from useful knowledge and can expect worse results, as we shall show in the experiments section.

4.4 Sampling

4.4.1 Bag subsampling

We have already explained how to transform a list of connections to a graph of servers and how to prune the graphs to select noiseless and more informative data. The resulting graph of servers however still remains too dense to extract useful features from every instance in a bag while training. If we somehow reduced the number of neighbors in the bag, we would be able to allocate more resources to the feature computation of the remaining portion of the bag more properly. *Bag subsampling* is a technique that attempts to represent the whole bag b with a set of *representative instances*, a subset of all instances in the bag. Let $n(c) = |\{v \in b\,;\, C(v) = c\}|$ denote a number of instances in the bag $b = \{v_1, v_2, \ldots, v_{|b|}\}$ belonging to class $c \in \mathcal{C}$ and $k(c)$ denote an intended number of instances used for the bag representation ($k(c) \leq n(c)$). First, we select $k(c)$ instances from the bag according to a *selection policy* and then assign each class c a *weight* proportional to its frequency in the bag:

$$w_{BS}(c) = \frac{n(c)}{k(c)} \tag{4.1}$$

As a consequence the following equality holds for every class c present in the bag b:

$$\sum_{v \in b,\ C(v)=c}^{n} w_{BS}(c) = n(c) \tag{4.2}$$

During the prediction phase, the vector of weights is fed together with mappings of selected instances into embedding space and consequently into a suitable aggregating function taking the weights on the instances as another input, for instance *Weighted mean* (Section 3.2.4). Weighing instances in this way enables us to put emphasis on instances that have larger weights while training, thus compensating for the rest of instances of the same class we opted to leave behind. The information about proportions of classes present in the bag is also preserved.

A strategy of which instances to select from the bag for its representation is an important decision to make and again can be carefully handcrafted to suit the particular domain's needs. Sometimes, the presence of one class in the bag may be more discriminative and sought after than the occurrence of another class, or, if we have an accurate and effective measure of *similarity* between instances, we are able to pick suitable and representative instances according to this measure. One possibility of measuring the similarity between instances in the bag and selecting the best subset is using *Greedy ϵ-Cover* [Kohout and Pevný, 2018], a technique applicable in any Hilbert space. In the CS domain, intuitively, not every instance in the bag is needed to make a decision, on the contrary, a sign of communication with only one well-known malicious server can be enough to uncover a threat. In our work with used the *random policy* – we selected representative instances randomly with uniform probability and without replacement. A pleasant side-effect of the bag subsampling with random policy is an introduction of stochasticity into learning, which is known to be beneficial for optimization with the *gradient descent* class of algorithms. Moreover, a stochastical sampling partially remedies the need to observe the network for a long time in order to capture all important behavior. For instance, instead of having 365 datasets for every day in the year, we can only have 12 from each month. Assuming that the pattern of the network traffic does not radically change between two following days, but rather during longer time spans, we reduce the loss of information by stochastic sampling.[3]

Since the class distribution in the CS domain is highly imbalanced, it is very often the case that a large number of negative (benign) instances is present in the bag and only a small number of positive (malicious) instances is present. Because the presence of a positive instance in the bag is more discriminating than the occurrence of a negative instance, we settled on using $k(c) = n(c)$ for a positive class and $k(c) = \min(k^{\ominus}, n(c))$ for a negative class for every bag, where $k^{\ominus}$ becomes a hyperparameter of our method.

4.4.2 High-level graph sampling

The last piece of the puzzle is how to sample bags themselves on the level of a server graph. To successfully train our model we need a way of selecting a representative sample of domains in each iteration of gradient descent and use bags associated with these domains for training. Due to the size of the graph, we cannot simply select a bag of every domain in a graph of servers, therefore we need to employ variants of the traditional *Batch gradient descent (BGD)* that do not require the whole dataset to be used in each iteration of learning, for example, *Mini-batch gradient descent (M-BGD)* [Ruder, 2016]. Ideally, we would like to keep all pieces of crucial information while selecting such subset of domains, similarly to preserving the information contained in the bag in the case of bag subsampling. This cannot be done by random sampling from domains in the graph as we need to consider classes of domains as well. An elegant solution is to first sample one class with uniform probability and after that sample a domain belonging to this class. This ensures that bags of every class are present in the resulting subset with approximately equal frequency, effectively dealing with the problem of imbalanced classes. On the other side, such sampling spoils the model's image about the real

[3]This assumption is not entirely correct since traffic changes a lot at weekends or on the public holiday in comparison with working days, however, the point has been made.

distribution of the data. We address this issue in the following section. Altogether, our sampling procedure is summarized in Algorithm 4. Finally, to construct a batch of bags used in M-BGD we repeat this procedure as many times as we wish making sure to sample without replacement so that a bag of one instance does not get duplicated in the batch.

Algorithm 4 Sampling procedure used to sample bags from a graph of servers. We firstly pick a class with uniform probability, then choose a domain belonging to this class with uniform probability again and finally perform bag subsampling on the bag.

1: **procedure** SAMPLING FROM A GRAPH OF SERVERS($D, E, k^{\ominus}$)
Input: $D \leftarrow$ vertices in a graph representing domains, $E \subseteq \{\{d_1, d_2\}; d_1, d_2 \in D\} \leftarrow$ edges in a graph of servers, $k^{\ominus} \leftarrow$ the sampling constant
Output: $b_s \leftarrow$ sampled bag, $w \leftarrow$ vector of weights of instances in sampled bag
2: $c_s \leftarrow$ a class sampled from a set of classes $\mathcal{C}$ with uniform probability
3: $d_s \leftarrow$ a domain sampled from $\{d \in D; C(d) = c_s\}$ with uniform probability
4: $b \leftarrow \{d \in D; \{d_s, d\} \in E\}$ ▷ neighbors of d_s
5: $b_+ \leftarrow \{d \in b; C(d) = +\}$ ▷ positive instances in the bag
6: $w_+ \leftarrow \underbrace{(1, 1, \ldots, 1)}_{|b_+|}$ ▷ weights on positive samples
7: $b_- \leftarrow \{d \in b; C(d) = -\}$ ▷ negative instances in the bag
8: $b_-^s \leftarrow \min(k^{\ominus}, |b_-|)$ sampled instances from b_- with uniform p. without replacement
9: $w_- \leftarrow \underbrace{(|b_-|/k^{\ominus}, |b_-|/k^{\ominus}, \ldots, |b_-|/k^{\ominus})}_{|b_-^s|}$ ▷ weights on negative samples
10: $b_s \leftarrow b_+ \cup b_-^s$ ▷ final bag
11: $w \leftarrow (w_+, w_-)$ ▷ final weights
12: **return** b_s, w
13: **end procedure**

4.5 Dealing with imbalanced classes

One of the typical characteristics of the CS domain is highly imbalanced classes as the overwhelming majority of all domains is benign. Similarly to spam detectors, our goal is to discover as many malicious servers as possible while keeping the false positive rate low. The desired outcome is, therefore, to train a model with the high precision which is preferred to having the high recall, ideally to have an option to increase one of these measures at the cost of lowering the other. An easy solution is to make use of a confidence value in the interval $(0, 1)$ output by both our method and the PTP algorithm. By increasing the decision threshold, we can increase the precision at the expense of the recall and vice versa. One useful way of comparing two methods outputting a confidence value is by plotting *Precision-recall curves (PR-curves)* and comparing them – this thesis is no exception. This offers a way to visualize the performance of a method taking all possible values of the decision threshold into account. We elaborate more on the measures of performance in the following chapter presenting experiments.

Besides a confidence value, a nice property of the method would be an option to include prior knowledge about the domain, namely about distributions of classes. For instance, in the CS domain there is a strong prior on a server being benign. It is estimated that the prior probability of an observed server being positive approaches the Bernoulli distribution with probability $p = 10^{-5}$, that it, only one server out of one hundred thousand is malicious. Training a model assuming uniform distribution of classes and isolating it from the priors about the domain leads to worse results.

Another useful property of the method is the possibility to define different costs c for false positives and false negatives. This is again the case with the CS domain, where tagging a negative server as positive is generally more unwanted mistake than tagging a positive server as negative. Finally, in the last section, we discussed sampling techniques which are beneficial in many aspects, however, they cause the difference between the distribution of the real data and the data the model encounters during training. We denote the true distribution of the data p and the sampling distribution q. In the following text, we introduce the loss function used for the training of models and show that it accommodates every issue described above. On the contrary, PTP offers no direct way to include priors on classes or costs of decisions and the only way to accomplish it is by various ad hoc solutions [Jusko, 2017].

Here, we abstract from our specific setting and denote objects of interest by x, their corresponding labels (classes) by y, and the output of our model by $f(x) = \hat{y}$. Let us narrow our focus on binary classification and set $y \in \{-1, 1\}$ to simplify equations. The following, however, applies to an arbitrary number of classes. We train our model to minimize the following *weighted cross entropy* objective:

$$\mathbb{E}_{(x,y)\sim q}\, w_y\, l(\hat{y}, y) \tag{4.3}$$

Note that we used the sampling distribution q instead of the true distribution p, because they are not the same. Remember that we first sample a class with probability $q(y)$ and then from all objects belonging to the sampled class we pick one and this is done evidently according to the true distribution p. The joint sampling probability can be therefore written as:

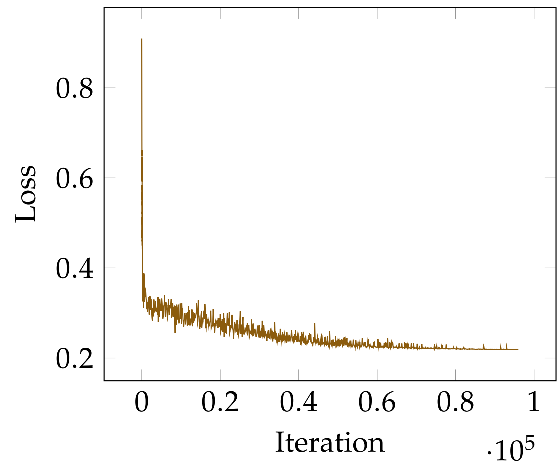


Figure 4.4: A characteristical loss function curve of learning with priors of classes. The loss initially rapidly decreases due to the bias we introduce to the model with weights in cross entropy loss function. Eventually, the steep decline stops and the value of the loss function stabilizes around a fixed point.

$$q(x, y) = p(x \mid y)q(y) \tag{4.4}$$

Returning to terms used in (4.3), w_y is a training weight of class y and l is the standard *multinomial cross entropy* loss:

$$l(\hat{y}, y_{real}) = -y_{real} \log(\hat{y}) - (1 - y_{real}) \log(1 - \hat{y}) \tag{4.5}$$

where y_{real}, $\hat{y}$ are real and predicted labels.

Now we rewrite the terms above using integrals and perform a specific substitution of w_y:

$$
\begin{aligned}
\mathbb{E}_{(x,y)\sim q}\, w_y\, l(\hat{y}, y) &= \int_{(x,y)} q(x, y)\, w_y\, l(\hat{y}, y)\, d(x, y) \\
&= \int_{(x,y)} p(x \mid y) q(y)\, w_y\, l(\hat{y}, y)\, d(x, y) = \left[w_y = \frac{p(y)}{q(y)} c_y \right] = \\
&= \int_{(x,y)} p(x \mid y) p(y)\, c_y\, l(\hat{y}, y)\, d(x, y) \\
&= \int_{(x,y)} p(x, y)\, c_y\, l(\hat{y}, y)\, d(x, y) = \mathbb{E}_{(x,y)\sim p}\, c_y\, l(\hat{y}, y)
\end{aligned} \tag{4.6}
$$

This result means that by optimizing the objective in (4.3) we simultaneously optimize (4.6), an expected value according to the true distribution p of the cross entropy loss with costs c_y included, provided that the substitution holds. Taking a closer look at the substitution of w_y, one may see, that the training weights are actually a product of training costs c_y and the *likelihood ratio* between p and q. This is an important observation as this enables us to include both priors $p(y = -1)$ and $p(y = 1)$ into the model. All we need to do is come up with suitable training weights w_y for classes. In this light, we can also consider a high-level graph sampling described in the previous section as a form of *importance sampling* on the distribution of server classes and therefore a model's variance reduction technique. The desired effect of including class priors to the model is reflected in the shape of the loss function curve, this is depicted in Figure 4.4.

To sum up the content of this chapter, we elaborated on a way to construct a graph of servers from a list of network connections and discussed the possible handling of large and dense graphs. We also introduced sampling techniques with low demands on both memory and time of computation that take properties of the dataset into account. Lastly, the weighted cross entropy loss function was introduced and it was explained how it helps us to reduce the variance of our model and to inject priors about classes and different costs for false positives and false negatives. We are now ready to employ the MINI formalism to perform the inference task.

Chapter 5

Experiments

This chapter discusses both graphs and server labels we used in experiments. Then a suitable evaluation pipeline is introduced, which takes into account several constraints that our task carries in comparison with classical statistical machine learning i.i.d setting. After that, we briefly elaborate on the remaining details of our model and the learning procedure. In the final part results from experiments are presented together with the comparison to the state-of-the-art PTP algorithm for inferring malicious servers.

5.1 Data

Here we specify which graphs and which ground-truth blacklists we worked with.

5.1.1 Graphs

To test our approach we used two datasets we call *CiscoNet1* and *CiscoNet2* collected from networks of companies serviced by Cisco Systems, Inc. during November 2017. A 24-hour-long time window was used (from 00:00:00 to 00:00:00 UTC) and two-week interval elapsed between the collection of *CiscoNet1* and *CiscoNet2*. The datasets were saved in the form of connection lists specifying bipartite graphs of servers and users. The properties of the graphs constructed from the datasets before and after pruning are shown in Tables B.2 and B.3. During our experiments, we trained our model using one dataset and tested it on the other. Pruning was performed every time with parameters $\alpha_D = \alpha_U = 5$ and $\beta_D = \beta_U = 10^5$. Note in particular the ratio of the positive and the negative class - each malicious server corresponds to roughly 10^5 benign servers.

5.1.2 Ground truth

Besides datasets in the form of lists of connections we also worked with ground truth provided through blacklists on various levels of abstraction:

- *Binary labels* only divide domains from the dataset into two disjoint sets, one set consisting of malicious (positive) domains and the other of benign (negative) domains.
- *High-level labels* split malicious servers even further and distinguish between types of malware according to its behavior. Examples include ransomware, spamware, adware, cryptocurrency-mining malware or malware designed to steal users' credentials.
- *Low-level labels* provide the most fine-grained labels, splitting high-level labels into several subcategories, based on the specific infrastructure of the threat contained on the domain. This could be for instance the protocol used for communication, the structure of a binary file or a presence of a *domain generation algorithm*. The set of domains sharing a low-level label is sometimes called a *cluster* because such domains tend to create dense subgraphs in the graph of servers.

In this work, we focused mainly on producing binary labels for the dataset, therefore, we did not need further separation. Nonetheless, the detailed classes of malware are important for the evaluation of the model, which is the topic of the next section. Table B.4 is included in the appendix to give the reader some examples of which low-level labels were defined on the blacklist.

5.2 Evaluation pipeline

In the introductory chapter we mentioned the difficulty of defining a suitable performance measure and as it turns out common measures like *accuracy* are not convenient for our task. This is due to the way we defined it – we want to infer labels to every vertex in a given graph and while inferring a label for one vertex we also make use of knowledge about other nodes in the graph. This strategy of considering a whole system of objects (graphs) at once instead of single objects is more general, because independently and identically distributed datapoints correspond to *discrete graph* in our setting, but causes another issue of how to measure the performance of a model on the whole system. The inference procedure is designed to extend our current knowledge about the system and not to derive it from scratch. Therefore we expose the blacklist of malicious domains to our model hoping that it would predict more malicious domains which are not currently on the blacklist. Many questions arise: How to simulate plausibly this inference environment when we test our model? The blacklist we have is very likely incomplete. How to deal with false positives? The model may discover a previously unknown domain which is not on the blacklist. Should the model be penalized for this even if it actually made the right decision? Definitely not, but how to tell the difference between this situation and genuine false positive? If we assume that the blacklist is perfect and we know about every malicious domain in the graph, how to test the model to find out that it will perform well given incomplete ground truth in real-world setting? The most straightforward solution is to use only a part of the blacklist and let the model infer the rest. However how to split appropriately the blacklist into two parts so that they accurately reflect the distribution of known malicious domains and malicious domains which we are yet to discover?

To gain a better understanding, imagine a scenario of using this method in practice. We have collected several datasets of lists of connections from the network we observe. Additionally, we know that some specific domains contain malicious content. We have a huge and frequently updated blacklist of them, but at the same time, we are aware that new malicious domains can appear every day and some even may not be detected by present methods. We have trained an inference model and decided to evaluate its performance. Which measure is accurate enough so that once our model attains higher scores we can be confident about its predictions? Designing such a measure requires complete information about the datasets, however, we lack in terms of both quantity (incomplete blacklists) and quality (weak labeling) of ground truth information. Unfortunately, the situation is not set to change in many domains. Precise labeling of a malicious domain or a malignant tumor is not going to be easier than detecting an object in the picture or pronouncing a review to be positive or negative. Thus, we have no choice but to devise measures that at least partially reduce the influence of the flaws described above on the results. For a more thorough discussion on performance evaluation we refer to [Jusko, 2017] or [Grill and Pevný, 2016].

In this work, we made an assumption that our ground truth blacklist is complete. The method produces a low number of false positives that can be examined by hand and potentially added to the blacklist, however, we did not head in this direction and considered the blacklist complete. Our main goal was to test properly the model's ability to discover new threats by giving it a suitable subset of domains from the blacklist and testing it on the remaining domains.

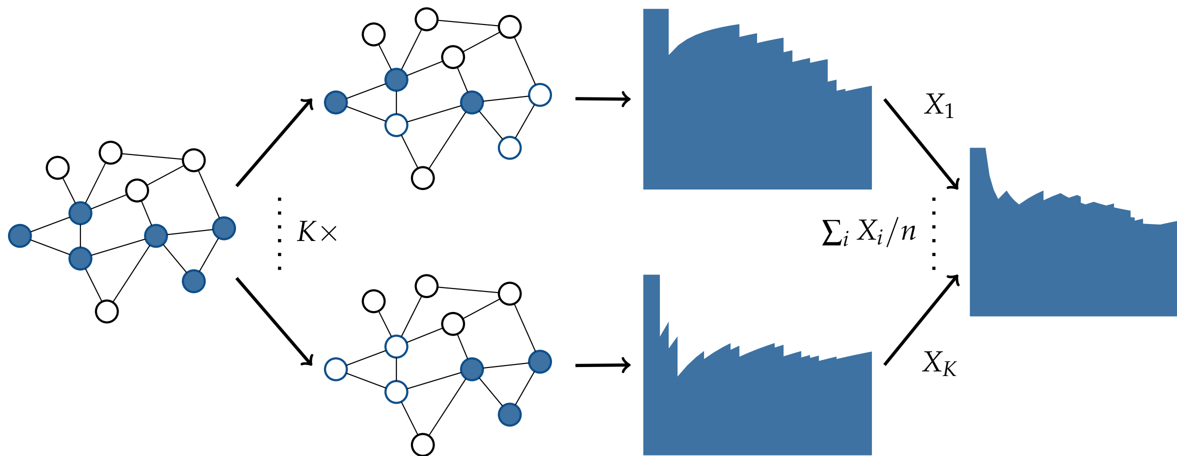


Figure 5.1: An illustration of how K-fold validation methods work. We begin by selecting a fold according to some strategy, which can be leaving one cluster out or stratified selection. We evaluate the performance of the model on the remaining domains, that are not in the fold. Having done that K times, we average the results.

5.2.1 Leave one domain out

This simplest *Leave one domain out (LODO)* evaluation method leaves the blacklist unchanged, takes one domain at a time and infers its label, hiding the true label of course, but the label of every other domain on the blacklist can be used for inference. Having done that on every domain in the server graph, we can plot PR-curves and judge the performance of the model, thus only one pass over the testing data is required, On the other hand the method makes an assumption that the blacklist used at the decision time is complete, which is not always the case. Therefore by evaluating with this method, we are given only a little evidence of how the model is capable of inference from an incomplete blacklist and can overall expect better curves (in fact this is the special case of methods explained next).

5.2.2 Fold validation

In the previous paragraphs, we explained the importance of selecting an appropriate subset of domains on the 'ground-truth' blacklist that would resemble the real world situation so that a model performing well in this simulated scenario would likely do well in practice too. This turns out as a non-trivial task and the main idea of *K-fold validation* methods is to remove the bias we introduce into evaluation by selecting a specific subset of domains by repeating the same procedure K times. We call such subsets *seeds*, *tips* or *folds* and define two possible approaches differing in the way we select them during each repetition. In comparison to LODO evaluation, K passes over the whole dataset are required. During each pass, we record the performance of the model on the domains that are not selected into the fold (both positive and negative). The total score (or performance curve) is then obtained by averaging over all K scores. Theoretically, if we let K approach infinity we would also approach the real world situation. The depiction of how *K-fold validation* methods work is in Figure 5.1.

Stratified fold validation

The method of *Stratified fold validation (SFV)* selects a proportion of every cluster to be included into the blacklist in each repetition. The proportion of each cluster is fixed and does not change during the evaluation. This approach ensures that on the blacklist there are at least some domains from every cluster and therefore the method tests if the model can discover additional malicious domains from the cluster given some (incomplete) knowledge about the cluster. If we lower the proportion of domains used, we make the task more challenging for the model. In our experiments, we used $K = 10$ due to the sheer size of the data we worked with.

Leave one cluster out

In the *Leave one cluster out (LOCO)* setting we remove a specific cluster of domains based on low-level labels from the blacklist, therefore K equals the number of such clusters. The model obtaining a high value of averaged results is able to discover new threats without any knowledge about them. This is a much harder task than the task of completing clusters simulated in *stratified validation* and the experimental results presented in this work do not evaluate the performance in LOCO setting.

5.3 Hyperparameters

In this section, we present our choice of hyperparameters. Two kinds of hyperparameters can be distinguished – the parameters of the model itself and the parameters of the training procedure. A very important parameter of the method is the choice of the feature mapping function $\boldsymbol{\alpha}_u$, which was described in full detail in the previous chapter (and for every experiment presented here the same function was used). One could consider the feature mapping as a parameter of data preprocessing together with pruning parameters, therefore it is not included in the following description in spite of being a part of method's pipeline.

5.3.1 Model hyperparameters

Instance embedding function $\boldsymbol{\phi}$ Instance embedding function was implemented as one or more feedforward neural network layers. Therefore a *number of layers* and *number of neurons* in each layer is adjustable. We experimented with more possibilities, however there were always 9 neurons in the first input layer (7 for each feature and 2 for one hot encoding of binary labels).

Aggregating function $\boldsymbol{g}$ Aggregating function was the same for all experiments, defined as a concatenation of element-wise maximum and weighted mean aggregating functions. As a consequence, output size of aggregation function $\boldsymbol{g}$ was doubled.

Classificating function $\boldsymbol{h}$ Classificating function was similarly to the instance embedding function $\boldsymbol{\phi}$ implemented as a neural networks with one or more layers. Because we used the same definition of $\boldsymbol{g}$ for all experiments, the size of input layer is required to be twice as big as an output layer of instance embedding function $\boldsymbol{\phi}$. Moreover, the number of neurons in the output layer must equal the number of classes. Number of layers and number of neurons in each other layer were hyperparameters we experimented with.

Layer types In all experiments, the layers composing $\boldsymbol{\phi}$ or $\boldsymbol{h}$ were standard fully-connected feedworward layers, differing only in their depth or width. Every feedforward layer was equipped with a type of normalization. Similarly to [Lei Ba et al., 2016], where authos scale all activations from every layer on the level of samples into a vector of zero mean and unit variance, we

normalized each vector $\boldsymbol{x}$ flowing through the network with following mapping:

$$\boldsymbol{x}_{norm} = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|_2}$$

This ensures that every sample is mapped to the unit sphere in the Euclidean space of activations, reducing the impact of potential concept drift.

Activation function To introduce nonlinearity into the model we employed either *Rectified linear unit (ReLU)* defined as:

$$f(x) = \max(0, x)$$

or *Leaky rectified linear unit (LReLU)* ([Xu et al., 2015]), known for mitigating a problem of *Dying ReLU*, defined as:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{otherwise} \end{cases}$$

where $a \in \mathbb{R}$ is a parameter. For all experiments with Leaky ReLU, we used the default value of $a = 0.01$. Last layer always consisted of linear transformation followed by the sigmoid function suitable for binary classification:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

5.3.2 Training hyperparameters

Loss function and training weights w_y In the last chapter we presented weighted cross entropy loss function taking class priors and costs of decision into account. This function was used for all experiments with training weights w_y being either fixed for the whole duration of training:

$$w_y(b) = \begin{cases} 0.01 & \text{if } c(x) = \textit{positive} \\ 0.99 & \text{if } c(x) = \textit{negative} \end{cases}$$

Gradient descent variant We used a variant called *Mini-batch gradient descent* to train neural nets constituting the model. Using the whole dataset as vanilla *Batch gradient descent* does is infeasible and we ruled out using only one or few datapoints in each step as *Stochastic gradient descent* does as the signal in them would be too weak.

Gradient descent optimizer *Adam* optimizer [Kingma and Ba, 2014] was applied to optimize the loss function in every experiment, with learning rate $\eta = 0.0001$ and other parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$.

Batch size A number of bags sampled from the graph, serving as an input to backpropagation algorithm in each iteration before weights in neural networks are updated, also varied in experiments. Typically, we would use 256 samples, however we experimented with bigger or lesser sizes. The batch-size of 256 samples seemed as a reasonable trade-off between the computational demands and the strength of the signal in the batch of bags.

Sampling strategy As it has already been discussed dataset size and imbalanced distribution of classes make a systematic iteration over dataset impossible, therefore on-demand uniform high-level sampling without replacement in combination with bag-subsampling was performed in each iteration.

Bag subsampling parameter $k^{\ominus}$ The parameter specifying how many negative instances from bag are sampled to represent it. Optimally, $k^{\ominus}$ should equal the number of all negative instances in the bag, leading to all negative instance being used, however this is too expensive to compute. We can also use different values of $k^{\ominus}$ for training and testing. During testing bag subsampling was also performed, but only several iterations over dataset were required, which enabled us to increase the value.

Number of iterations This hyperparameter specifies how many times weights were updated in the network with gradients obtained from backpropagation algorithm using a batch of bags as an input. We trained every model in experiments for the total number of 10^5 iterations.

Training and testing dataset In every experiment we trained a model using the dataset *CiscoNet1* and tested it on the dataset *CiscoNet2*.

Data normalization *Min-Max scaling*, which translates each value serving as an input to a network between zero and one, was performed in every experiment, using the formula:

$$X_{scaled} = \begin{cases} \frac{X - X_{min}}{X_{max} - X_{min}} & \text{if } X_{min} \neq X_{max} \\ 1 & \text{otherwise} \end{cases}$$

where X_{max} and X_{min} is the biggest and smallest possible value of the input.

Due to the fact that many hyperparameter were same for all experiments carried out, we will restrict our attention to the hyperparameters we actually changed during experimenting and every result will be accompanied by a table similar to this:

hyperparameter	value
$\boldsymbol{\phi}$	$(9, 30)$
$\boldsymbol{h}$	$(60, 2)$
activation	ReLU
batch size	256
$k^{\ominus}_{test}$	100

This means that the model consisted of two-layer instance embedding function $\boldsymbol{\phi}$ with 9 neurons in the input layer and 30 neurons in the output layer. Classification function $\boldsymbol{h}$ was also implemented with two layers, first one having 60 neurons and the second one 2. All layers in the model except the last layer in the classification function use ReLU as their activation and gradients in every iteration were computed from batch of 256 bags. The same constant $k^{\ominus}_{test}$ was set to 100 during the test time. All other hyperparameters do not change, that is, aggregating function $\boldsymbol{g}$ was the concatenation of the mean and the element-wise maximum, the same normalization was employed on the level of the data and the activations from the layers, model was trained for 10^5 iterations, with fixed training weights w_c (0.01 for the positive and 0.99 for the negative class), using Mini-batch gradient descent algorithm with Adam optimizer, and the sampling was performed with the same constant $k^{\ominus}_{train}$ equalling 100.

The model defined in the table above, in reality, turned out as a convenient baseline for further experiments. We began with training this model and then experimented with other configuration, comparing the results.

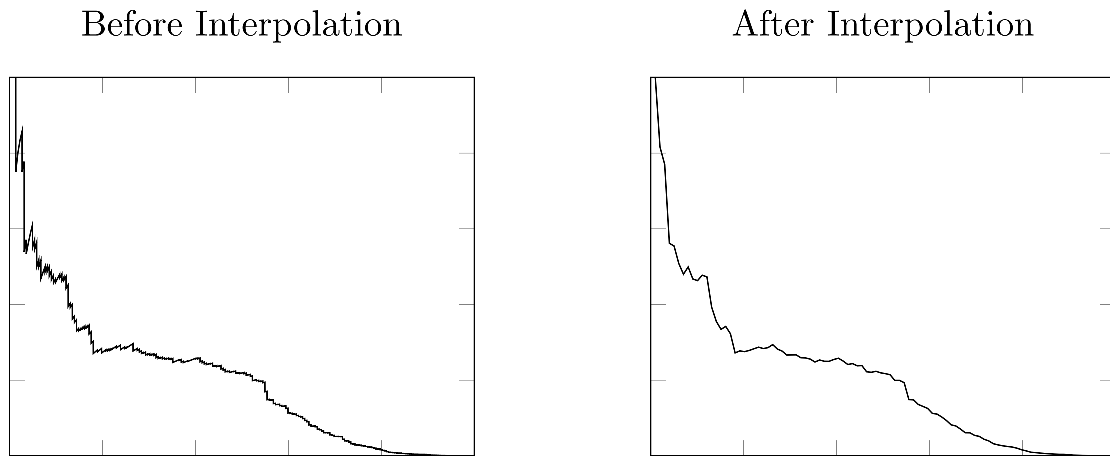


Figure 5.2: An illustration of linear interpolation on PR curves. The curve on the right is computed from every point in the dataset whereas 100 points for approximation was used on the left. We give up some details in order to better observe global characteristics of the curve.

5.4 Results

All experiments were carried out on a virtual server with 16 CPUs and 64 GB of RAM running on the Amazon Elastic compute cloud (EC2)[1]. No GPUs were used for training. The programming language chosen for experimenting was Julia[2], because of the speed it achieves comparable to the speed of compiled languages and its wide support for distributed and parallel computing. The training time ranged between 2 and 5 days, depending on the complexity of the model or the batch size we trained with. Inference for every domain in the datasets took approximately half a day.

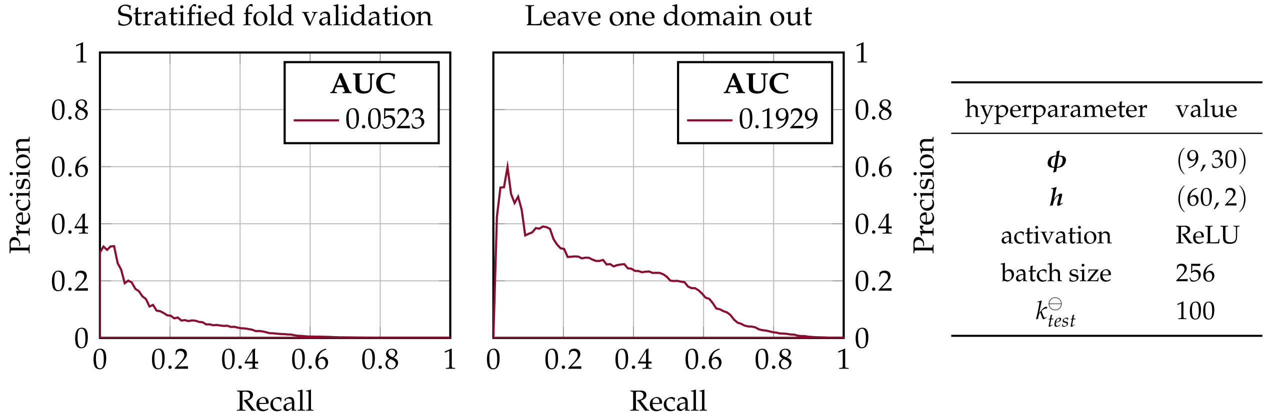
To compare the performance of the models, we examine PR-curves, which capture the performance very good in case of imbalanced classes. By gradually increasing the decision threshold we can calculate precision and recall for different points in the curve. For various values of the decision threshold we used values that were predicted for every domain in the dataset accurately compute *Area under (the) curve (AUC)* performance metric. However, the resulting curve has as many points as there are domains in the testing dataset that we evaluate on. This leads to generally chaotic and spiky curve, where global characteristics are difficult to observe. We, therefore, sampled one hundred linerly spaced points and used linear interpolation to compute their values as depicted in Figure 5.2.

We evaluated following models in *Stratified fold validation* and *Leave one domain out* settings. Due to the different amount of domains in each fold, the remaining portion of the domains we predict on differs as well. Therefore, interpolation of the curves also enables us to compute per-point average for *Stratified fold validation* setting. The shapes of the curves in following experiments may give the impression, that our model performs poorly. Remember that the seemingly poor performance is given by the unusual complexity of the task. Furthermore, keep in mind that the distribution of classes is highly imbalanced in both the real world and our datasets and also that we attempt to classify a server based on solely the simple features extracted from the server graph and nothing else (as explained in Section 4.3).

[1] https://aws.amazon.com/ec2/
[2] https://julialang.org/

5.4.1 Baseline model

Firstly, we measured the performance of the baseline model defined in the previous section. The results are in Figure 5.3.



hyperparameter	value
ϕ	$(9, 30)$
h	$(60, 2)$
activation	ReLU
batch size	256
$k^{\ominus}_{test}$	100

Figure 5.3: PR curves of the baseline model.

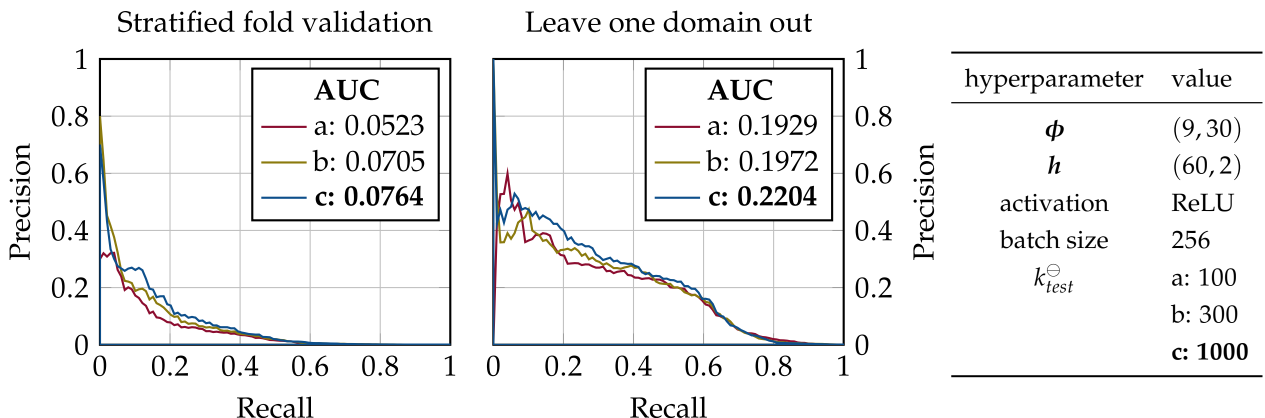
The curves indicate that the model is capable of detecting malicious servers even in Stratified fold validation setting, however falls behind in some aspects. For instance, the appearance of the curve on the left for small values of recall shows that only a few right decisions are made with high confidence and some benign servers incorrectly obtain high probability of being malicious.

5.4.2 Further experiments and improvements

In this section we present the results we obtained by altering either the defition of the model or the training procedure. After that we show the comparison of some selected models to the *Probabilistic threat propagation* algorithm.

Increasing the number of negative instances in the bag

The first thing we tried was increasing $k^{\ominus}$ at the test time. We experimented with values of $k^{\ominus}_{test}$ from $\{100, 300, 1000\}$ (Figure 5.4).



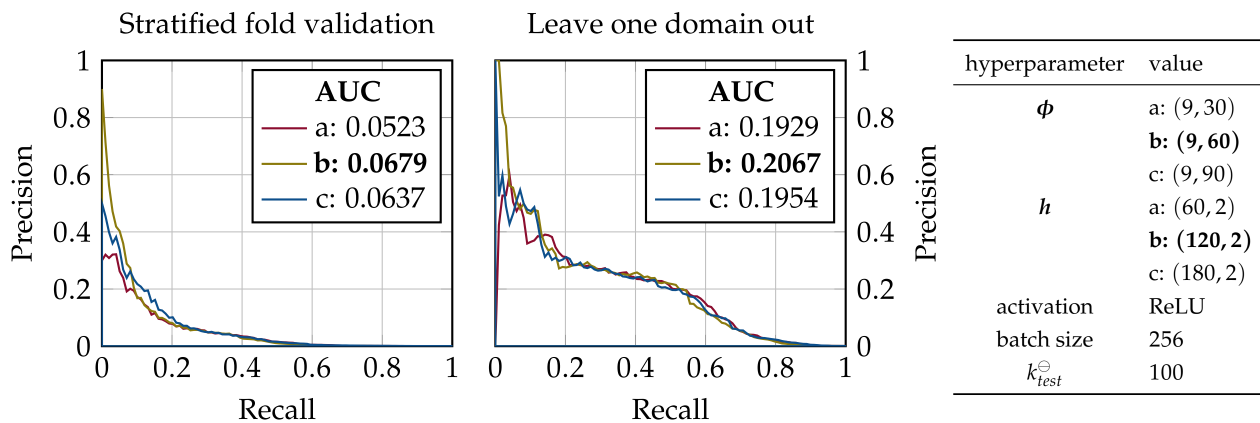
hyperparameter	value
ϕ	$(9, 30)$
h	$(60, 2)$
activation	ReLU
batch size	256
$k^{\ominus}_{test}$	a: 100 b: 300 **c: 1000**

Figure 5.4: PR curves of the baseline model with different numbers of negative instances in the bag

It can be clearly seen that enabling the model to work with more negative examples in the bag increases the performance. The result of the experiment where we set $k^{\ominus}_{test}$ to 1000 indicates that the area under the curve metric saturates and further change of $k^{\ominus}_{test}$ probably would not help. This is in agreement with the assumption that the whole set of negative instances in the bag is needed to make the decision. Also, the curves differ much more in the first picture describing stratified fold validation results. This is probably due to the fact, that increasing the number of sampled negative instances in the bag results into more malicious servers (that are not in the fold and thus considered positive) being in the bag.

Going wider

Another reasonable direction is changing the definition of the instance embedding function $\boldsymbol{\phi}$ and the classification function $\boldsymbol{h}$. The instance embedding function in the baseline model consisted of the input layer of 9 neurons followed by a layer of 30 neurons and the classification function was implemented by one layer of 60 neurons and the output layer of 2 neurons. We tried increasing the number of neurons in the second layer of instance embedding function to 60 and 90 (and also accordingly increased the number of neurons in the first layer of the classification function so that the dimensions agreed) (Figure 5.5).



hyperparameter	value
$\boldsymbol{\phi}$	a: $(9, 30)$
	b: $(9, 60)$
	c: $(9, 90)$
$\boldsymbol{h}$	a: $(60, 2)$
	b: $(120, 2)$
	c: $(180, 2)$
activation	ReLU
batch size	256
$k^{\ominus}_{test}$	100

Figure 5.5: PR curves of the models with layers of different widths in the bag.

Quick examination uncovers that the wider models classify serveral malicious instances with high confidence resulting into the 'peak' of curves near the origin.

Going deeper

We also attempted to change the number of layers in ϕ and h instead of changing the size of the layers, as it is usually the case that deeper models improve accuracy of neural networks. The following figure shows the results of the baseline model and models with 3 or 4 layers instead of 2 in both ϕ and h (Figure 5.6).

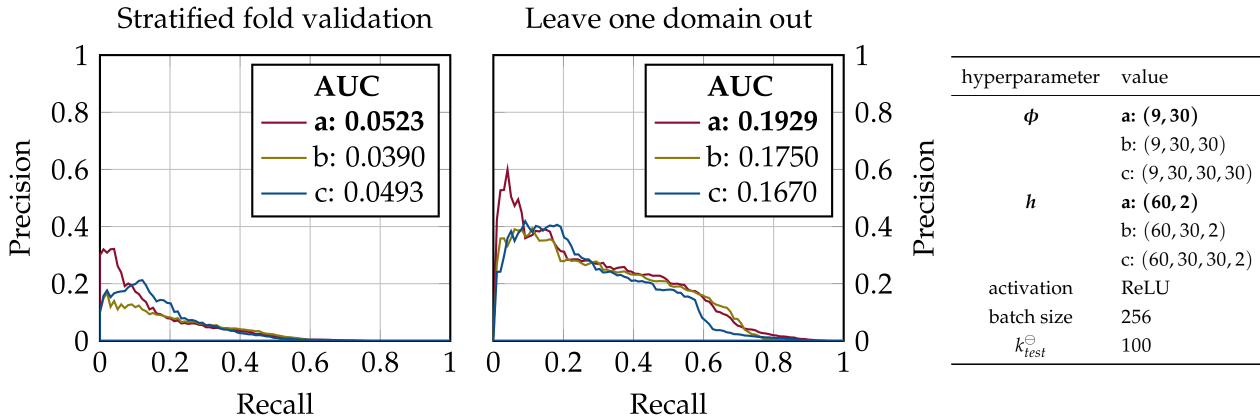


Figure 5.6: PR curves of the models of different depth.

Contrary to the expectations, adding more layers did not improve the results much and even worse performance was achieved.

Different batch size

It is a well-known empirical observation that increasing or decreasing of the batch size may increase (or decrease) the performance. The bigger batch carries stronger signal and the smaller batch makes the optimizing procedure more stochastic, resulting into bigger chance to escape local optimas. The baseline model was trained with 256 bags in each batch and we trained the same model using batches of size 64 or 512 (5.7).

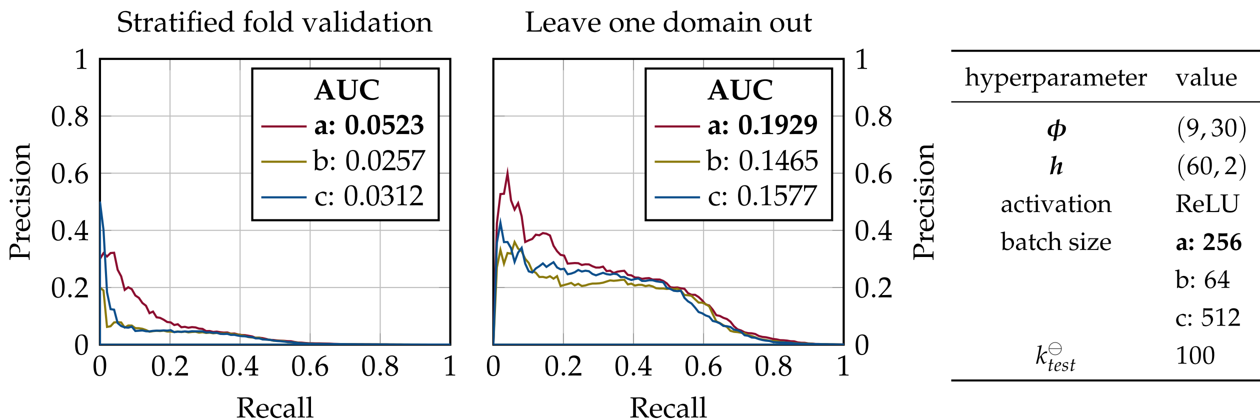
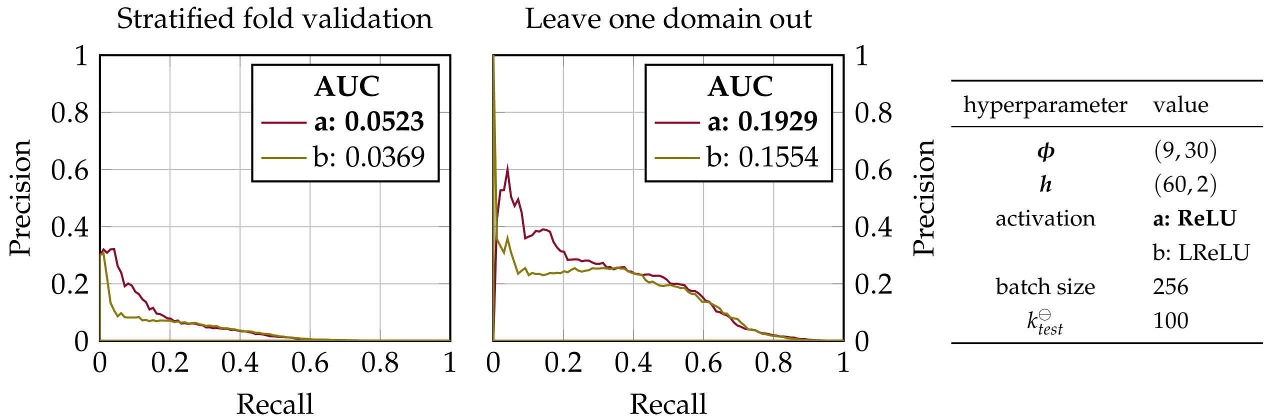


Figure 5.7: PR curves of the models trained with different batch sizes.

The results imply that the batch size of 256 was well-chosen.

Leaky ReLUs

Lastly, we tried different activation function due to our concerns of dying ReLU units (5.8).



hyperparameter	value
Φ	$(9, 30)$
h	$(60, 2)$
activation	**a: ReLU**
	b: LReLU
batch size	256
$k_{test}^{\ominus}$	100

Figure 5.8: PR curves of the models with ReLUs or Leaky ReLUs as the activation function.

Dying ReLUs are evidently not a problem of the baseline model.

5.4.3 Comparison to Probabilistic Threat Propagation

Lastly, we compared our results to the results of the Probabilistic threat propagation, the state of the art algorithm. The implementation was the same as in [Jusko et al., 2016], where no external knowledge used for the computation of edge weights as well, thus seemingly poor results may be expected. PTP ran for 40 iterations. In Figure 5.9 we compare the performance of the PTP method with the performance of several our models. Besides the baseline model and the model with increased $k_{test}^{\ominus}$, whose curves were already presented in the sections above, we also include models with combined hyperparameters. On the basis of previous results, we evaluated models with deeper or wider architectures with a higher number of sampled negative instances in the bag. Intuitively, performance should improve.

In each of our experiments, our method outperformed the PTP method by a large margin. Although all experiments were carried out using only one dataset for training and one for testing, thus to declare our method superior would require more experiments on different data, this shows bright prospects of the approach. The fact that even the simple baseline model, a launching pad for further experimenting, achieved better results, demonstrates that careful tuning of model properties and hyperparameters is likely to result into even better performance. We are fully aware that the method requires more thorough testing and consider it as future work. The sketch of our plans for the future is provided in the following, last chapter. Formalizing the task the way it was done is more general and less constraining. This opens the variety of directions for the future research. Some of these possible directions were already discussed and the rest is described in the chapter to follow. To finish, it is important to emphasize that better performance in experiments is not the only advantage of our method over the PTP algorithm. Benefits of drawbacks of both methods are summarized in Table 5.1.

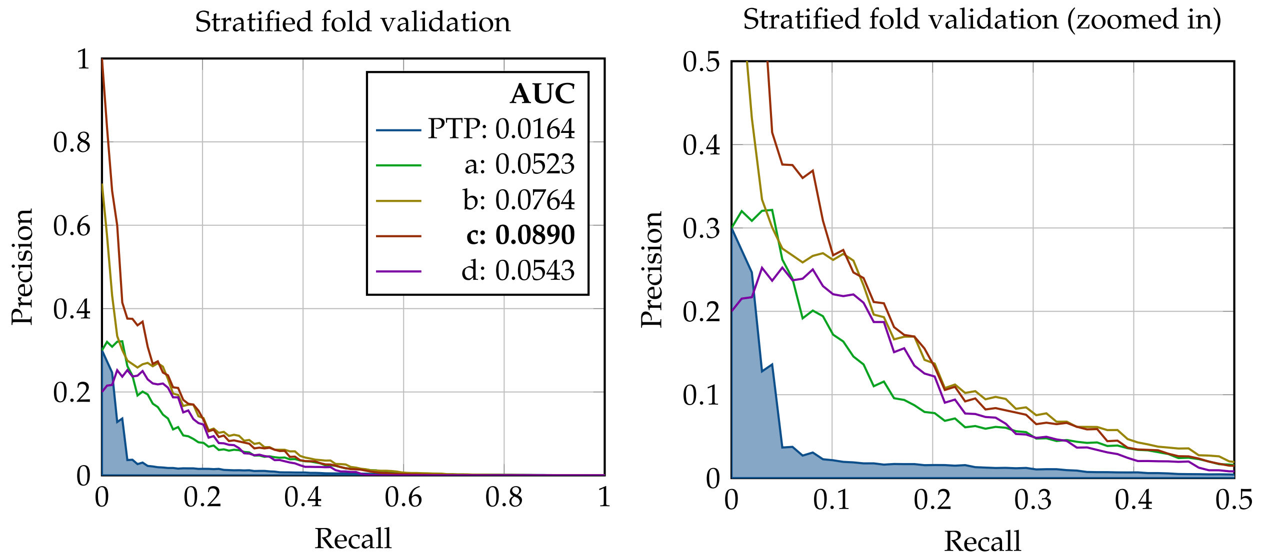


Figure 5.9: Comparison of selected models – baseline (a), $k^{\ominus}_{test} = 1000$ (b), **wider model with** $k^{\ominus}_{test} = 1000$ **(c)**, and deeper model with $k^{\ominus}_{test} = 1000$ (d) – with the state-of-the-art PTP method (40 iterations). Only the results from the stratified fold validation setting are shown, because multiple step nature of the PTP algorithm made it impossible to evaluate the leave one domain out setting for each positive domain in the dataset. The picture on the right is the same as the one on the left zoomed to interval $[0, 1/2] \times [0, 1/2]$.

Table 5.1: Benefits and drawbacks of both methods we compared.

MINI formalism method	**Probabilistic threat propagation**
Performs only one inference step. This increases the speed but at the same time, the assumption that all information needed to make the decision is present in the Markov's blanket of the server (see Section 3.2.6) is made. This is usually the case in the CS domain, but may not be true in general.	Its multi-step nature in theory allows the propagation of information throughout the whole graph and not only the close neighborhood.
The general definition of the predictors used in each step allows for proper selection of knowledge to propagate through the graph. This makes MINI formalism a theoretical concept applicable and adjustable to many problems.	On the other hand, PTP provides limited possibilities of selecting which information gets propagated further.
The general definition of feature mapping function opens up the possibility of describing both vertices taking part in the relation as well as the properties of the relation itself in a natural way.	The weights on edges and initial threats in the vertices are the only specifiable 'degrees of freedom' of the method. No features representing vertices can be directly used and the only way to include them is by some ad-hoc artificial methods.
Enables external knowledge to be effortlessly included into the model and complement features extracted from graph.	No external knowledge can be directly injected and can be only taken into account in the definition of edges' weights or the initialization of threat in individual vertices.
The functions comprising the model in the formalism implemented as neural networks accommodate many possible inference formulas and the correspondence between a bag describing a vertex and the Markov blanket of the vertex represent a natural and flexible view of the general inference problem.	Employs fixed formulas for threat propagation.
The training procedure can easily cope with new data and weighted cross entropy loss functions enables us to include prior knowledge about the distribution of classes or assigning different costs to mistakes made by the model.	Only possible learning of parameters would mean learning of edge weights. This requires a whole inference procedure to be run after every step of SGD. This is infeasible given the complexity of a single inference run. Therefore, adaptation to new data is inflexible.
The training took 2-5 days, depending on the model, and performing the inference took half a day, on an EC2 *m5.4xlarge* machine (16 CPUs and 64 Gb of RAM). We were able to achieve such small memory requirements because of the dataset pruning and the fact, that we do not store the whole server graph in the memory and computed everything from the bipartite graph or on demand. This results in much lower memory demands, but on the other hand, increases computing time.	No prior knowledge about classes can be leveraged and there is no direct way to express preferences by assigning different costs to false positives and false negatives.
	Performing the inference using carefully optimized source code demands several hours, on an EC2 *x1.32xlarge* machine (64 CPUs and 1,952 Gb of RAM). However, the high memory requirement is not a drawback of the method itself, but rather the approach to computation. The same tricks can be employed to sacrifice some computing time in order to save memory, but once we attempt to store the whole dense server graph in the memory, referring to B.3, there may be approximately 10^{11} edges in the server graph.

Chapter 6

Conclusion

This work proposed a novel formalism for thinking about the inference in graphical models and investigated its strengths and weaknesses. Our main interest lies in the development of an efficient and general algorithm that would be able to infer any unknown quantity in a graph. The universality of the formalism introduced in this thesis is one step towards this goal and many research ideas and directions are still open. In the end, we want to be able to define a high-level inference procedure and let the learning algorithm learn any inference rules by its own from a sufficient amount of data. It has been experimentally demonstrated that the method using our formalism overcomes older approaches due to its flexibility and the ability to learn any function used for inference. However, we still feel that the method can be further improved by defining better learning procedure or using a stronger model. The potential of the method is huge and many possible applications suggest themselves. Indeed, we have opted for the CS domain and the task of server classification, but this is only the tip of the iceberg. The world around us is full of dynamical systems evolving in time, consisting of objects that interact with others. Examples include:

- **Social graphs**. We may be interested for instance in detecting fake accounts, accounts exhibiting illegal activity or communities of users with similar interest that we can target with our product. We may experiment with multiple ways of defining adjacency relation in social graphs as there are more ways of how users present themselves on the network. Some of these ways, such as communication between users, require access to private data of users, others are publicly observable without any permissions.
- **Natural language processing**. Including a relations between texts being classified can improve the classification performance. For instance we might put edges between two documents one of which references the other.
- **Stock Marketing**. By putting an edge between companies offering similar products or companies with the same purpose, we can predict if their stock price is likely to grow or not. External knowledge in the form of previous stock prices and other companies features can be used.

6.0.1 Future work

We finish the thesis by listing several ideas for the future work. There is certainly much room for improvement of models and the training procedure. Trying to solve different problems with the formalism is another possible thing to investigate. Nonetheless, our future aim is to use the formalism and extend the inference method to perform more than one step (as it has been discussed in Section 3.2.6). Furthermore, we would like to improve our results in malware classification task. We intend to add external features to the model to complement purely graphical features and see how the results change. This way we may observe to what extent is the method able to efficiently use external knowledge together with the knowledge encoded in the graph. This may be done by comparing the performance of the model using only graphical features, the performace of the model using only external features and then the performance of the model using both. It is evident, that we may expect better performance, however, the question of how much the results would differ remains to be answered. Another possible way to go is to improve the adjacency relation construction described in Section 4.1 and extend it to work with multipartite graphs. Such graphs would contain not only

servers and users, but also programs that were used by users to access the servers, nameservers that store the location of a domain name's various services or Whois query results. The problem becomes more complex, however, one can verify that the general MINI formalism smoothly accommodates to the problem and apart from designing the adjacency relation construction process no additional effort is required.

Appendices

Appendix A

Probabilistic threat propagation

Probabilistic threat propagation [Carter et al., 2013] is an iterative method for detecting malicious objects in the network represented by an undirected graph. It assumes that the threat is highly concentrated in some parts of the network, forming *communities*. Utilizing tips from blacklists, probabilistic threat propagation firstly assigns high threat $P(X)$ to each blacklisted object X in the network representing the probability of X being dangerous. By iterative propagation of the threat throughout the network, other malicious objects are discovered. The probability of maliciousness is defined as the weighted sum of probabilities of neighbors in the graph conditioned on the current node being benign ((A.1)).

This condition prevents the method from the unwanted effect of 'direct feedback', when the probability of maliciousness of the current node is influenced by the threat the node had previously propagated to its neighbors. By setting zero threat to the node, this effect is nullified.

$$P(X_i) = \sum_{X_j \in \mathcal{N}(X_i)} w_{ij} P(X_j \mid X_i = 0) \quad \text{(A.1)}$$

A.0.1 Approximate Inference

Solving (A.1) exactly requires quadratic time for computing the conditional probability for all pairs of nodes in the graph. This makes exact inference infeasible for larger graphs, which is almost always the case for computer networks. Authors of the paper proposed an iterative approximation described in (A.2).

$$P^k(X_i) = \sum_{X_j \in \mathcal{N}(X_i)} w_{ij}(P^{k-1}(X_j) - C^{k-1}(X_i, X_j)) \quad \text{(A.2)}$$

Here $P^k(X)$ denotes the threat at iteration k and $C^{k-1}(X_i, X_j)$ is a part of $P^{k-1}(X_j)$ propagated from a node X_i (in the previous step). This successfully prevents the direct feedback from happening. In each iteration, all threats are simultaneously recalculated and iterations are repeated until terminating condition or convergence. To reinforce knowledge about tips from the blacklist, threats of the tip nodes are initialized to a high value at the beginning of every iteration.

A.0.2 Weights

The last remaining question to be answered is how to assign weights used in (A.2). In order to obviate the need of normalization weights should satisfy equality $\sum_j w_{ij} = 1$. While this enables one to alter properties and behavior of the algorithm by changing the way weights are assigned, it also introduces a necessity to decide even if it may not be clear how to do so. A thorough discussion on how to set weights w_{ij} is in [Jusko, 2017].

Apart from the need to pick suitable weights, the algorithm is not applicable for cases when external knowledge about objects (apart from their threat level) is important for decisions. Also, injecting any prior about the distribution of classes is impossible.

Appendix B

Dataset details

Table B.1: The most and the least frequently visited domains in the bipartite graph from the dataset *CiscoNet1*.

rank	domain	# of connections	domain	# of connections
1.	*google.com*	$2.356 \cdot 10^6$	*riverlifechurch.org.au*	1
2.	*microsoft.com*	$2.073 \cdot 10^6$	*marelec.gr*	1
3.	*googleapis.com*	$1.799 \cdot 10^6$	*exploreair.com*	1
4.	*gstatic.com*	$1.713 \cdot 10^6$	*sonharcomsonhos.co*	1
5.	*doubleclick.net*	$1.600 \cdot 10^6$	*tsunami-martial-arts.com*	1
6.	*windowsupdate.com*	$1.502 \cdot 10^6$	*167.114.174.141*	1
7.	*google-analytics.com*	$1.460 \cdot 10^6$	*yokumoku.co.jp*	1
8.	*symcd.com*	$1.279 \cdot 10^6$	*rugbytownfc.com*	1
9.	*facebook.com*	$1.190 \cdot 10^6$	*hertford.gov.uk*	1
10.	*live.com*	$1.179 \cdot 10^6$	*gorsko.eu*	1
11.	*digicert.com*	$1.175 \cdot 10^6$	*objessions.com*	1
12.	*bing.com*	$1.146 \cdot 10^6$	*browncountytx.org*	1
13.	*googlesyndication.com*	$1.068 \cdot 10^6$	*myfail-tube.com*	1
14.	*googleadservices.com*	$1.030 \cdot 10^6$	*tobegourmet.com*	1
15.	*adnxs.com*	$1.010 \cdot 10^6$	*153.3.233.106*	1
16.	*googleusercontent.com*	$9.871 \cdot 10^5$	*wszystkoconajwazniejsze.pl*	1
17.	*facebook.net*	$9.609 \cdot 10^5$	*hirsch-automobile.ch*	1
18.	*googletagmanager.com*	$9.584 \cdot 10^5$	*portosaude.com*	1
19.	*gvt1.com*	$9.412 \cdot 10^5$	*hmprg.org*	1
20.	*scorecardresearch.com*	$9.071 \cdot 10^5$	*risearmament.com*	1
21.	*googletagservices.com*	$8.944 \cdot 10^5$	*hegeoraekszer.hu*	1
22.	*yahoo.com*	$8.918 \cdot 10^5$	*pavelkogan.com*	1
23.	*msn.com*	$8.902 \cdot 10^5$	*patinageplateau.com*	1
24.	*cloudfront.net*	$8.718 \cdot 10^5$	*basketczechowice.pl*	1
25.	*demdex.net*	$8.371 \cdot 10^5$	*wearedepaul.com*	1
26.	*twitter.com*	$8.346 \cdot 10^5$	*cheztapioca.com.br*	1
27.	*youtube.com*	$8.330 \cdot 10^5$	*turtlemoon.com*	1
28.	*amazonaws.com*	$8.303 \cdot 10^5$	*praxisclarahof.ch*	1
29.	*adsrvr.org*	$8.174 \cdot 10^5$	*ifoodala.com*	1
30.	*adobe.com*	$8.161 \cdot 10^5$	*ufsinc.com*	1

Table B.2: Properties of the datasets used in experiments before pruning. Δ, δ, and d denote the maximal, the minimal and the average degree in a graph specified in a subscript (Bipartite graph or server graph).

name	*CiscoNet1*	*CiscoNet2*
# of servers	$3.97 \cdot 10^6$	$3.69 \cdot 10^6$
# of positive servers	955	655
# of users	$2.81 \cdot 10^6$	$2.33 \cdot 10^6$
Δ_{SG}	$2.73 \cdot 10^6$	$2.25 \cdot 10^6$
δ_{SG}	0	0
d_{SG}	$\approx 1.25 \cdot 10^4$	$\approx 7.91 \cdot 10^3$
Δ_{BG} (server)	$2.36 \cdot 10^6$	$1.95 \cdot 10^6$
Δ_{BG} (user)	$8.14 \cdot 10^4$	$4.1 \cdot 10^4$
d_{BG} (server)	≈ 77.23	≈ 77.605
d_{BG} (user)	≈ 54.57	≈ 49.56
δ_{BG} (server)	1	1
δ_{BG} (user)	1	1

Table B.3: Properties of the datasets used in experiments after the complete pruning using parameters $\alpha_D = \alpha_U = 5$ and $\beta_D = \beta_U = 10^5$. Δ, δ, and d denote the maximal, the minimal and the average degree in a graph specified in the subscript (bipartite graph or server graph).

name	*CiscoNet1*	*CiscoNet2*
# of servers	$4.74 \cdot 10^5$	$3.99 \cdot 10^5$
# of positive servers	340	222
# of users	$1.97 \cdot 10^6$	$1.68 \cdot 10^6$
Δ_{SG}	$4.29 \cdot 10^5$	$3.61 \cdot 10^5$
δ_{SG}	4	4
d_{SG}	$\approx 3.04 \cdot 10^4$	$\approx 2.19 \cdot 10^4$
Δ_{BG} (server)	$9.92 \cdot 10^4$	$9.94 \cdot 10^4$
Δ_{BG} (user)	$4.7 \cdot 10^4$	$3.33 \cdot 10^4$
d_{BG} (server)	≈ 166.9	≈ 172.54
d_{BG} (user)	≈ 40.08	≈ 40.98
δ_{BG} (server)	1	1
δ_{BG} (user)	1	1

Table B.4: Examples of low-level cluster labels on the blacklist. 'C&C' is an abbreviation for Command & control servers. The first column provides the name of the cluster (low-level label), the second column names of higher-level labels corresponding to the definition of the threat in the last column.

Cluster code	High-level categorization	Threat definition
CCPT04	• cryptowall • ransomware • C&C	Threat related to a variant of the last Cryptowall (3.0) botnet and ransomware which encrypts the disk of the infected device and extorts the affected user. Bot communicates using HTTP with the command-and-control server and sends information about the infected device before encrypting the disk. Botnet uses compromised, legitimate sites as temporary command-and-control servers.
CEXF01	• information stealer • exfiltration • malware distribution • C&C	Threat related to malware that exfiltrates large amounts of data to external servers using a custom protocol on TCP port 443, disguising it as HTTPS traffic. Additionally, threat can download malware and other malicious applications onto the affected device to perform click fraud.
CAST01	• asterope • click fraud • botnet • C&C	Threat related to Asterope click-fraud botnet. Asterope is a Trojan bot malware which performs click fraud by imitating the action of a user clicking on an advertisement. The bot communicates with the command-and-control server using HTTP from which it receives the next website to visit.
CLDM01	• loadmoney • ad injector • malware distribution • pua • adware	Threat identified as the potentially unwanted application (PUA) known as LoadMoney, which distributes additional adware and other malicious software onto the affected device. Threat arrives on the system while installing freeware and exfiltrates information from the infected system.
CADT03	• tracking • malicious advertising	Threat related to advertisements and web analytics tracking which leaks sensitive information from the user device such as unique user identifier, referrer, and system events. Threat is associated with known adware, ad injectors, and other malicious software.
CSPM03	• spam • phishing • tracking • malicious advertising	Threat related to spam campaigns which may include advertisements and phishing. May expose user email address to third parties. May be distributed via email or as advertisement banners on web mail sites such as Yahoo.

Appendix C

DVD Content

The attached DVD contains the Julia code that was used for the data preprocessing, the training and the evaluation. Due to the size of the dataset and the privacy regulations about user data we cannot disclose the dataset, however, a dummy dataset representing a simple task is prepared. To run the whole project, the following tools must be installed (numbers in parentheses signify particular version of the library used):

- Julia language (0.6.1)
- MLDataPattern.jl (0.4.0)
- MicroLogging.jl (0.2.0)
- GraphPlot.jl (0.2.0)
- Gadfly.jl (0.6.5)
- CSV.jl (0.1.5)
- Revise.jl (0.1.1)
- Hiccup.jl (0.1.1)
- Flux.jl (0.3.4)
- PyPlot.jl (2.5.0)
- Conda.jl (0.7.1)
- MLBase.jl (0.7.0)
- JLD.jl (0.8.3)
- PGFPlotsX.jl (0.2.1)
- AWSS3.jl (0.3.6)
- ForwardDiff.jl (0.7.5)
- DualNumbers.jl (0.3.0)
- FastClosures.jl (0.2.0)
- DataStructures.jl (0.8.2)
- Calculus.jl (0.3.1)
- DataFrames.jl (0.10.1)
- JSON.jl (0.17.2)
- StatsBase.jl (0.22.0)

The contents of the DVD are structured as follows:

/mandlsim_bachelor_thesis_*.pdf The electronic version of this work.

/src/ A directory containing all code and provided data.

/src/data/ Dummy dataset representing simple task and dummy blacklist of domains that can be used for experimenting (generated by *random_graph.jl*)

/src/Datasets/ Algorithms and data structures for working with various types of datasets, such as dataset of graphs or dataset of bags.

/src/FluxModels/ Definitions of various neural network layers and models used in experiments.

/src/_init.jl Initial script needed to be run after starting Julia.

/src/train.jl Training script containing definitions of hyperparameters and main train loop.

/src/evaluate.jl Evaluation script, usually executed right after training. Provides all validation methods discussed.

Both **/src/Datasets/** and **/src/FluxModels/** are parts of a bigger library, which is a property of Cisco Systems, Inc., therefore the code cannot be published. The attached DVD does not contain these parts, however, they can be disclosed to those interested.

Acronyms

A

AUC	Area under (the) curve	[43]

B

BG	Bipartite graph	[27]
BGD	Batch gradient descent	[33]
BN	Bayesian network	[7]

C

C&C	Command and control	[3, 56]
CRF	Conditional random field	[9, 10]
CS	Computer security	[2, 3, 6, 15, 16, 19, 24, 26, 28, 33–35, 49, 50]

D

DDoS	Distributed denial of service	[3, 29]

E

EC2	Elastic compute cloud	[43, 49]

L

LOCO	Leave one cluster out	[40]
LODO	Leave one domain out	[39]
LReLU	Leaky rectified linear unit	[41, 47]

M

M-BGD	Mini-batch gradient descent	[33, 34]
MB	Markov blanket	[7]
MIL	Multiple instance learning	[1, 2, 4, 16, 19, 20, 23, 24]
MINI	Multiple instance neural inference	[2, 5, 15–17, 19, 20, 24, 26, 36, 51, 59]
MPI	Message passing inference	[10, 17, 18, 23, 24]
MRF	Markov random field	[8–11, 14, 17, 59]

P

PR-curves	Precision-recall curves	[34, 39, 43]
PTP	Probabilistic threat propagation	[2, 5, 34, 35, 37, 47, 48]

R

ReLU	Rectified linear unit	[41, 42, 44–47]

S

SFV	Stratified fold validation	[40]
SG	Server graph	[27]
SGD	Stochastic gradient descent	[18, 49]

U

UTC	Coordinated universal time	[37]

Nomenclature

Graph interpretation

$\mathcal{G}$	Space of all observable graphs (Section 3.1, page 15)
$\mathcal{V}$	Set of all vertices in a graph (Section 2.1, page 6)
$u, v, w, \ldots$	Vertices in a graph (Section 3.1, page 15)
$\mathcal{E}, E$	Set of all edges in a graph, adjacency rleation on the set of vertices (Section 2.1, page 6)
$\mathcal{N}(v)$	All neighbors of the vertex v (Section 2.2, page 11)
d_G	Average degree in the graph G (Section B.0, page 55)
Δ_G	Maximal degree in the graph G (Section B.0, page 55)
δ_G	Minimal degree in the graph G (Section B.0, page 55)

Probabilistic interpretation

$\mathcal{X}$	Finite set of all considered variables (Section 3.2, page 16)
$\mathbf{X}_O$	All observed random variables (Section 3.1, page 15)
$X, Y, Z, \ldots$	Random variables (Section 2.1, page 6)
$X \perp\!\!\!\perp Y \mid \boldsymbol{Z}$	Conditional independence of X and Y given Z (Section 2.1, page 6)
$\mathcal{N}(X)$	All neighbors of the random variable X in MRF (Section 2.1, page 8)
$\mathbf{MB}(X)$	Markov blanket of the variable X (Section 2.1, page 7)
$\mathbf{X}_K$	Set of vertices (or random variables) in the clique K (Section 2.1, page 8)
$\psi(\boldsymbol{X})$	Potential function operating on the set of random variables $\boldsymbol{X}$ (Section 2.1, page 8)
$f, f', \ldots$	Factors in a factor graph (Section 2.2, page 10)
$\boldsymbol{X}(f)$	Random variables in the subtree with root f (Section 2.2, page 11)
$F_f(X_i, \boldsymbol{X}(f))$	Product of all factors in the subtree formed by the factor f in the tree rooted in X_i (Section 2.2, page 11)
$\mu_{f\to X}$	Message sent from the factor f to the variable X (Section 2.2, page 11)
$\mu_{X\to f}$	Message sent from the variable X to the factor f (Section 2.2, page 11)
$E(\boldsymbol{X})$	Energy of the set of random variables (Section 2.1, page 8)
$d(\boldsymbol{X})$	Integration over the values of all random variables in vector $\boldsymbol{X}$ (in the right order) (Section 2.1, page 8)
$\boldsymbol{X} \setminus X_i$	Vector of random variables with the variable X_i left out (Section 2.2, page 11)

MINI formalism

θ	Parameters describing the situation of the system at the observation time (Section 3.2, page 16)
$(\mathcal{G}, \mathcal{A}, P_\theta)$	Probability space of all graphs (Section 3.2, page 16)
$\mathcal{F}$	Mapping between observable graphs and the parameter space (Section 3.2, page 16)
$\mathcal{C}$	Set of all classes (Section 3.1, page 15)

$C(v)$	Class of the vertex v (the actual value of random variable associated with it) (Section 3.1, page 15)
$\mathcal{B}$	Space of all bags (Section 3.2, page 19)
b_u	Bag associated with the vertex u (Section 3.2, page 19)
$\boldsymbol{\alpha}_u(v)$	Feature mapping of the relation $\{u, v\}$ (in the context of the bag b_u) (Section 3.2, page 19)
$\boldsymbol{\phi}$	Instance embedding function (Section 3.2, page 19)
$\boldsymbol{g}$	Aggregating function (Section 3.2, page 19)
$g_{max}, g_{mean}, g_{wm}$	Maximum, mean and weighted mean activation functions (Section 3.2, page 21)
$\boldsymbol{h}$	Classification function (Section 3.2, page 19)
s	Number of steps performed by an inference method (Section 3.2, page 24)
$L_{[t_1,t_2]}$	List of connections observed in the interval between t_1 and t_2 (Section 4.1, page 26)
$\mathcal{U}$	Set of all users connected to the network (Section 4.1, page 26)
$\mathcal{D}$	Set of all domains accessible from the network (Section 4.1, page 26)
$I(e)$	Information carried by the edge e (Section 4.1, page 26)
$\alpha_D, \beta_D, \alpha_U, \beta_U$	Lower and upper bounds for pruning (Section 4.2, page 28)
$\mathcal{N}_{SG}(u), \mathcal{N}_B(u)$	Set of all neighbors of v in server graph and bipartite graphs of servers and users (Section 4.3, page 31)
$w_{BS}(c)$	Bag subsampling weight of the class c (Section 4.4, page 32)
$k^{\ominus}_{train}, k^{\ominus}_{test}$	Maximal number of negative instances left in the bag after performing bag subsampling, during training or testing phase (Section 4.4, page 32)
p, q	True and Sampling distributions of the data (Section 4.5, page 34)
$y_{real}, \hat{y}$	True class and Estimated class (Section 4.5, page 34)
w_y	Training weight of the class y (Section 4.5, page 34)
c_y	Training cost of the class y (Section 4.5, page 34)

Probabilistic threat propagation

$P^k(X_i)$	Threat computed on the vertex X_i at step k (Section A.0, page 53)
$C^k(X_i, X_j)$	Part of $P^k(X_i)$ that was propagated through node X_j (Section A.0, page 53)
w_{ij}	Weight on the edge between vertices X_i and X_j (Section A.0, page 53)

Other Symbols

$\mathbb{R}^n$	Euclidean space of dimension n (Section 3.2, page 19)
$\mathcal{P}(M)$	The powerset of the set M (Section 3.2, page 16)
$\boldsymbol{x}^{(i)}$	i-th component of the vector $\boldsymbol{x}$ (Section 3.2, page 21)

Bibliography

[Babenko, 2008] Babenko, B. (2008). Multiple instance learning: Algorithms and applications. Page(s): [1, 4].

[Baum and Petrie, 1966] Baum, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite state markov chains. *Ann. Math. Statist.*, 37(6):1554–1563. Page(s): [4, 10].

[Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. Page(s): [4, 14].

[Carter et al., 2013] Carter, K. M., Idika, N., and Streilein, W. W. (2013). Probabilistic threat propagation for malicious activity detection. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 2940–2944. IEEE. Page(s): [4, 53].

[Chapelle Olivier, 2006] Chapelle Olivier, Schölkopf Bernhard, Z. A. (2006). *Semi-supervised learning*. Cambridge, Mass.: MIT Press. Page(s): [2].

[Chen et al., 2006] Chen, Y., Bi, J., and Wang, J. Z. (2006). Miles: Multiple-instance learning via embedded instance selection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):1931–1947. Page(s): [20].

[Freepik, 2016] Freepik (2016). Assorted user avatars collection. Page(s): [27].

[Frey and MacKay, 1998] Frey, B. J. and MacKay, D. J. C. (1998). A revolution: Belief propagation in graphs with cycles. In Jordan, M. I., Kearns, M. J., and Solla, S. A., editors, *Advances in Neural Information Processing Systems 10*, pages 479–485. MIT Press. Page(s): [4, 10, 12].

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. Page(s): [1].

[Grill and Pevný, 2016] Grill, M. and Pevný, T. (2016). Learning combination of anomaly detectors for security domain. *Computer Networks*, 107:55–63. Page(s): [38].

[Gulcehre et al., 2014] Gulcehre, C., Cho, K., Pascanu, R., and Bengio, Y. (2014). Learned-norm pooling for deep feedforward and recurrent neural networks. In Calders, T., Esposito, F., Hüllermeier, E., and Meo, R., editors, *Machine Learning and Knowledge Discovery in Databases*, pages 530–546, Berlin, Heidelberg. Springer Berlin Heidelberg. Page(s): [22].

[Hammersley, 1971] Hammersley, J. M.; Clifford, P. (1971). Markov fields on finite graphs and lattices. Page(s): [4, 8, 17].

[Horng Polo Chau et al., 2011] Horng Polo Chau, D., Nachenberg, C., Wilhelm, J., Wright, A., and Faloutsos, C. (2011). Polonium: Tera-scale graph mining and inference for malware detection. pages 131–142. Page(s): [4].

[Hu et al., 2017] Hu, J., Shen, L., and Sun, G. (2017). Squeeze-and-Excitation Networks. *ArXiv e-prints*. Page(s): [1].

[Jaccard, 1902] Jaccard, P. (1902). Lois de distribution florale dans la zone alpine. *Bull Soc Vaudoise Sci Nat*, 38:69–130. Page(s): [31].

[Jusko, 2017] Jusko, J. (2017). Graph-based detection of malicious network communities. Page(s): [29, 35, 38, 53].

[Jusko et al., 2016] Jusko, J., Rehak, M., Stiborek, J., Kohout, J., and Pevny, T. (2016). Using behavioral similarity for botnet command-and-control discovery. *IEEE Intelligent Systems*, 31(5):16–22. Page(s): [4, 47].

[Kazato et al., 2016] Kazato, Y., Sugawara, T., and Fukuda, K. (2016). Detecting malicious domains with probabilistic threat propagation on dns graph. *Computer Software*, 33(3):16–28. Page(s): [4].

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *ArXiv e-prints*. Page(s): [41].

[Kohli and Torr, 2005] Kohli, P. and Torr, P. H. S. (2005). Efficiently solving dynamic markov random fields using graph cuts. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pages 922–929 Vol. 2. Page(s): [4, 10].

[Kohout and Pevný, 2018] Kohout, J. and Pevný, T. (2018). Network traffic fingerprinting based on approximated kernel two-sample test. *IEEE Trans. Information Forensics and Security*, 13(3):788–801. Page(s): [33].

[Kschischang et al., 2001] Kschischang, F. R., Frey, B. J., and Loeliger, H. A. (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519. Page(s): [4, 10, 11].

[Lafferty et al., 2001] Lafferty, J. D., McCallum, A., and Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 282–289, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. Page(s): [4].

[Lei Ba et al., 2016] Lei Ba, J., Kiros, J. R., and Hinton, G. E. (2016). Layer Normalization. *ArXiv e-prints*. Page(s): [40].

[Liben-Nowell and Kleinberg, 2003] Liben-Nowell, D. and Kleinberg, J. (2003). The link prediction problem for social networks. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 556–559, New York, NY, USA. ACM. Page(s): [31].

[Lin et al., 2015a] Lin, G., Shen, C., Hengel, A. v. d., and Reid, I. (2015a). Efficient piecewise training of deep structured models for semantic segmentation. *ArXiv e-prints*. Page(s): [18].

[Lin et al., 2015b] Lin, G., Shen, C., Reid, I., and van den Hengel, A. (2015b). Deeply Learning the Messages in Message Passing Inference. *ArXiv e-prints*. Page(s): [4, 19].

[Pearl, 1988] Pearl, J. (1988). *Probablistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann. Page(s): [4, 10, 11, 12].

[Pevný and Nikolaev, 2015] Pevný, T. and Nikolaev, I. (2015). Optimizing pooling function for pooled steganalysis. In *2015 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6. Page(s): [22].

[Pevný and Somol, 2017] Pevný, T. and Somol, P. (2017). Using neural network formalism to solve multiple-instance problems. In *International Symposium on Neural Networks*, pages 135–142. Springer. Page(s): [4, 21].

[R. Kindermann, 1980] R. Kindermann, J. S. (1980). *Markov Random Fields and Their Applications*. American Mathematical Society. Page(s): [4, 8].

[Ross and Bagnell, 2010] Ross, S. and Bagnell, D. (2010). Efficient reductions for imitation learning. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 661–668, Chia Laguna Resort, Sardinia, Italy. PMLR. Page(s): [24].

[Ross et al., 2010] Ross, S., Gordon, G. J., and Bagnell, J. A. (2010). A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. *ArXiv e-prints*. Page(s): [24].

[Ross et al., 2011] Ross, S., Munoz, D., Hebert, M., and Andrew Bagnell, J. (2011). Learning message-passing inference machines for structured prediction. Page(s): [4, 18, 19].

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*. Page(s): [33].

[Schwing and Urtasun, 2015] Schwing, A. G. and Urtasun, R. (2015). Fully Connected Deep Structured Networks. *ArXiv e-prints*. Page(s): [18].

[Silver et al., 2017] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G. v. d., Graepel, T., and Hassabis, D. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354. Page(s): [1].

[Stiborek et al., 2017] Stiborek, J., Pevný, T., and Rehák, M. (2017). Multiple instance learning for malware classification. *arXiv preprint arXiv:1705.02268*. Page(s): [4, 21].

[van den Oord et al., 2016] van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). WaveNet: A Generative Model for Raw Audio. *ArXiv e-prints*. Page(s): [1].

[Wang et al., 2000] Wang, L., Liu, J., and Li, S. Z. (2000). Mrf parameter estimation by mcmc method. *Pattern Recognition*, 33(11):1919 – 1925. Page(s): [4, 10].

[Wu et al., 2016] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, Ł., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *ArXiv e-prints*. Page(s): [1].

[Xu et al., 2015] Xu, B., Wang, N., Chen, T., and Li, M. (2015). Empirical Evaluation of Rectified Activations in Convolutional Network. *ArXiv e-prints*. Page(s): [41].

[Yu et al., 2010] Yu, T., Lippmann, R., Riordan, J., and Boyer, S. W. (2010). Ember: a global perspective on extreme malicious behavior. In *VizSEC*. Page(s): [3].

[Yu et al., 2017] Yu, Z., Chen, F., and Dong, J. (2017). Neural network implementation of inference on binary markov random fields with probability coding. *Applied Mathematics and Computation*, 301:193 – 200. Page(s): [4].

[Zheng et al., 2015] Zheng, S., Jayasumana, S., Romera-Paredes, B., Vineet, V., Su, Z., Du, D., Huang, C., and Torr, P. H. S. (2015). Conditional Random Fields as Recurrent Neural Networks. *ArXiv e-prints*. Page(s): [18].