FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Temporal Aspect Aware Graph Neural Network in Cybersecurity
Student:	Anton Bushuiev
Supervisor:	Ing. Pavel Prochazka, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2021/2022

Instructions

The main goal of the thesis is to take advantage of natural data records origin inherently containing both sequences of events
and the corresponding timestamps. Current approaches typically ignore the temporal information and form a graph from the
records regardless of this information. The thesis should provide an overview of approaches regarding taking gain from the temporal
information in graph neural networks. Based on this overview a proper method should be selected and implemented on some
Cisco Cognitive maliciousness classification problem to demonstrate the temporal awareness advances over simple baseline methods.

Electronically approved by Ing. Karel Klouda, Ph.D. on 11 December 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Temporal Aspect Aware Graph Neural Network in Cybersecurity

Anton Bushuiev

Department of Applied Mathematics

Supervisor: Ing. Pavel Prochazka, Ph.D.

May 13, 2021

Acknowledgements

I thank Cisco, Cognitive Intelligence group and Ing. Pavel Prochazka, Ph.D. for the opportunity to enjoy this interesting research.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 13, 2021 . . . . . .

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

Bushuiev, Anton. *Temporal Aspect Aware Graph Neural Network in Cybersecurity.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Žít v dynamickém světě znamená řešit časově závislé úlohy. Avšak moderní nástroje pro strojové učení na grafech jsou především navržené pro statické sítě. Proto se v této závěrečné práci detailně zabývám problematikou strojového učení respektujícího časový aspekt pro grafové úlohy. Výsledkem tohoto teoretického výzkumu je návrh dynamické grafové neuronové sítě se spojitým časem. Zaměřuji se na problém Cisco Cognitive Intelligence maliciousness classification — úlohu odhalení internetových domén s bezpečnostním rizikem na základě interakcí mezi uživateli a doménami. Ukazuji, že tento problém lze efektivně vyřešit použitím různých přístupů strojového učení, včetně navrženého. Navíc demonstruji, že obecné zákonitostí bezpečnostního rizika domén nevykazují dynamické vlastnosti v uvažovaných datech z reálného světa.

Klíčová slova učení grafové reprezentace, grafová neuronová sít, dynamický graf, klasifikace úzlů, kybernetická bezpečnost

Abstract

Living in a dynamic world means dealing with time-dependent tasks. However, the modern toolbox for machine learning on graphs is mainly designed for static networks. Therefore, in this thesis, I deepen into the problematics of temporal-aware machine learning approaches for graph problems. The outcome of this study is a proposal for the new continuous-time dynamic graph neural network. I focus on the Cisco Cognitive Intelligence maliciousness classification problem — the task of malicious Internet domain exposure based on user-domain interactions. I demonstrate that this problem can be efficiently solved employing different approaches, including the proposed one. Moreover, I show that general maliciousness patterns do not exhibit dynamic properties in the considered real-world data.

Keywords graph representation learning, graph neural network, temporal network, dynamic graph, node classification, maliciousness classification, cybersecurity

Contents

List of Figures

List of Tables

Introduction

A human mind tends to analyze in terms of objects and relations between them. As a result, many real-world tasks are often considered as problems in the domain of graph theory. The rise of machine learning brought a number of effective techniques to solve such problems. Modern so-called representation learning approaches such as graph neural networks provide solutions for various tasks on graphs including, for example, *node classification* or *link prediction*.

Mentioned approaches are designed for static graphs which do not change over time, but obviously, in many real-world problems incorporating a temporal aspect of data is crucial. Consider for example a mobility network of the USA, i.e. a graph with nodes representing the US counties and labeled edges representing human mobility between pairs of counties. Given county-level dynamic COVID-19 statistics, i.e. node labels representing the numbers of newly infected, our goal is to predict the next-day numbers of new cases across the US counties based on the previous days. Obviously, described network changes over time and it is critical to leverage graph dynamics in this example. That is what Kapoor et al. [1] show in their work. They use a graph neural network on the constructed spatio-temporal graph to solve this problem leveraging the dynamics of data.

Living in a dynamic world, we are surrounded by similar problems in various domains from zoology [2] to e-commerce [3]. The goal of this thesis is to deepen into the problematics of temporal-aspect-aware approaches designed to tackle these problems. In this work, I focus on the *Cisco Cognitive Intelligence maliciousness classification problem* — the task of malicious Internet domain exposure based on user-domain interactions. Cisco, Cognitive Intelligence group tracks site visits by users to secure them from cyber threats utilizing the collected information. In this thesis, I study the influence of data temporal

aspect consideration on the possible approaches and their efficiency regarding the malicious domains detection.

This work involves several challenges. The most significant obstacle is the nature of the obtained data. Machine learning arose from the great amount of easy-of-access labeled data. However, tracking millions of domains we possess valuable information only about a tiny fraction of them, and the rest remain unknown. This fact makes challenging not only the leveraging of data dynamics but also the application of machine learning in general, challenging. To alleviate the lack of data there needed specialized techniques and approaches. The second significant challenge is the immaturity of machine learning on dynamic graphs. Non-trivial state-of-the-art approaches are often task-driven, while general techniques are typically incapable of capturing time at its finest granularity.

Thesis structure

The thesis starts with two preliminary chapters. In Chapter 1 I discuss the elements of machine learning on graphs and in Chapter 2 I aim to provide an overview of state-of-the-art machine learning approaches trying to leverage graph dynamics. Based on this theoretical study, I propose a dynamic graph neural network model in Chapter 3. Further, the *Cisco Cognitive Intelligence maliciousness classification problem* is formally defined and discussed in Chapter 4. Finally, in Chapter 5, I describe the experiments conducted to compare the performance of temporal-aspect-aware approaches and baseline methods regarding the considered problem.

Notation

Linear algebra

In this work, $\mathbb{R}^n$ ($n \in \mathbb{N}$) is used as a *standard vector space* over the field $\mathbb{R}$. The ith element ($i \in \{0, n-1\}$) of a vector $\mathbf{a} \in \mathbb{R}^n$ is denoted as $\mathbf{a}[i]$. Then, $\mathbb{R}^{m \times n}$ is a set of corresponding matrices of m rows and n columns. The element of a matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ placed at the ith row and the jth column ($i \in \{0, m-1\}, j \in \{0, n-1\}$) is denoted as $\mathbf{M}[i, j]$. Similarly, $\mathbf{M}[i, :], \mathbf{M}[:, j]$ represent the ith row and the jth column respectively.

A *standard inner product* $\langle \cdot, \cdot \rangle : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is defined as

$$\langle \mathbf{a}, \mathbf{b} \rangle := \sum_{i=0}^{n-1} \mathbf{a}[i]\mathbf{b}[i]. \tag{1}$$

With $\|\cdot\|_p : \mathbb{R}^n \to \mathbb{R}$ is denoted the *L_p-norm*, defined as

$$\|\mathbf{a}\|_p := \sqrt[p]{\sum_{i=0}^{n-1} |\mathbf{a}[i]|^p}. \tag{2}$$

A binary operator $\oplus : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^{n+m}$ represents a *vector concatenation*:

$$\mathbf{a} \oplus \mathbf{b} := (\mathbf{a}[0], \ldots, \mathbf{a}[n-1], \mathbf{b}[0], \ldots, \mathbf{b}[m-1]). \tag{3}$$

Miscellaneous

An *Iverson bracket* for a proposition P is defined as:

$$[P] := \begin{cases} 1 \in \mathbb{R} & \text{if P is true,} \\ 0 \in \mathbb{R} & \text{otherwise.} \end{cases} \tag{4}$$

Bold braces $\mathbf{\{}$, $\mathbf{\}}$ are used to denote *multisets* contrary to sets denoted with $\{$, $\}$.

Chapter 1

Machine learning on graphs

Many real-world problems can be formulated in terms of a graph theory. For example, a global computer network can be defined as a *graph* with *nodes* representing computers, routers, switches, and other entities; and links representing connections between them.

Definition 1.0.1. A (simple undirected static) *graph*[1] is an ordered pair $G = (V, E)$, where V is a non-empty finite set of *nodes* or *vertices* and $E \subseteq \{\, \{u, v\} \mid u, v \in V \wedge u \neq v \,\}$ is a set of *edges* or *links*.

Having such a network, we may be interested in finding a route between two computers. This problem, called routing, can be efficiently formulated and solved algorithmically with, for example, Dijkstra's algorithm [5]. As a different example, we may want to detect malicious entities in the described network. Such a task cannot be easily formulated precisely due to the ambiguity of the term "malicious", which cannot be efficiently defined mathematically in terms of a graph theory in real-world problems. In this case, maliciousness usually means the occurrence of certain patterns in graph topology or its features. Understanding such a problem requires expert knowledge about the network and the maliciousness. Extracting the knowledge from data and simulating a human expert is a subject of study for machine learning. I assume that the reader is familiar with this field of study, and, in this chapter, I directly proceed to machine learning on graphs — a technique to solve such problems.

The applications of machine learning on graphs can be found in various fields. Some interesting examples include antibiotics discovery [6] and question answering [7]. In the first example, the goal is to predict the properties of

[1] Sometimes I use term *network* instead of *graph*, especially when I refer to a specific, real-world instance of defined abstract structure, consistently with Hamilton [4].

molecules (extract the features of graphs), while in the second example, the proposed technique is aimed to find the edge between the vertices of a single graph which leads to a correct answer. We can see that in these applications, the underlying graph problems are totally different. Applications on graphs that require machine learning can be classified by the types of underlying problems. Let's take a look at two important examples of such problems: *node classification* and *link prediction*.

A *node classification* is a standard machine learning classification task. Consider a graph $G = (V, E)$. Having a mapping $\mathcal{T} \to L$ for a training set of nodes $\mathcal{T} \subset V$ and a set of labels L, our goal is to find the mapping $V \setminus \mathcal{T} \to L$. For example, malicious entity detection, described at the beginning of this chapter, may fall under this class of problems. Having $L = \{0, 1\}$, with $0 \in L$ and $1 \in L$ representing beingness and maliciousness respectively, the goal is to assign these labels to unknown nodes, having labels of the "training" nodes. I return to the similar problem in Chapter 4 and define it precisely.

A *link prediction* can be defined as an assignment of labels from $\{0, 1\}$ or probabilities to the edges of a graph *complement*. Having a graph $G = (V, E)$ and its complement $G^C = (V, E^C)$, the task is to find a mapping $E^C \to \{0, 1\}$ or $E^C \to [0, 1]$, indicating if the edges could/should be added to the graph G.

Definition 1.0.2. A graph $G^C = (V, E^C) = (V, \binom{V}{2} \setminus E)$ is called a *complement* of a graph $G = (V, E)$. With $\binom{V}{2}$ are all subsets of V of size 2 denoted, i.e all possible edges on vertices V.

For example, having a social network $G = (V, E)$ with V representing users and E representing a friendship relation, we may want to offer possible friends to users. Obviously, offering possible friends can be defined as link prediction in G.

Thus, this chapter aims to provide a brief overview of state-of-the-art approaches of machine learning on graphs to discuss the possible solutions of mentioned tasks.

1.1 Graph representation learning

Many modern machine learning techniques are strongly based on linear algebra. That is why dealing with non-numerical data often means representing them as numerical. For example, images in computer vision tasks are usually treated as matrices, or words in natural language processing problems are often converted to vectors. Similarly, having a numerical representation of a graph or its elements enables us to use general machine learning methods to solve graph problems. That is why in the rest of this chapter, I overview basic

ideas and chosen state-of-the-art models of *graph representation learning* — a dominant paradigm in machine learning on graphs. A comprehensive overview of this field of study can be found in the book by Hamilton [4] which is a basis for this chapter.

Representation learning on graphs means extracting knowledge of interest about a graph and representing it in so-called *latent space* — typically low dimensional[2] vector space $\mathbb{R}^d$. We are usually interested in the representation of graph nodes called *node embeddings*. Indeed, for example, embeddings of edges can be obtained from nodes representation using binary vector operators, or we can aggregate a set of nodes embeddings to get a representation of a whole graph. Then, having nodes represented as real vectors we are able to use all the power of machine learning to solve tasks such as node classification or link prediction — that is the main idea of graph representation learning.

However, what is the right way to place nodes in the latent space? Consider, for example, a cycle graph[3] C_{20}. How should we properly arrange its nodes on a plane $\mathbb{R}^2$? In the Figure 1.1 different ways of embedding are shown. In the first case (Figure 1.1a), we are placing *adjacent* nodes (see Definition 1.2) close to each other, since in the Figure 1.1b, we oppositely put them far from each other. Of course, the choice of embedding depends on the motivation to do it. The first method could be useful, for example, for visualization, as in the picture we can clearly recognize a circle. But if we want to find an approximation of a *graph coloring*[4] problem solution, we may proceed as follows: (i) embed the nodes, (ii) find the clusters with, for example, k-means[5] technique, (iii) associate clusters with distinct colors. Having the adjacent nodes placed far from each other in the latent space, such a method could provide decent results. That is why the second embedding (Figure 1.1b) might be useful as well.

Generally, it is difficult to preserve all the information about a graph in the latent space. That is why we usually want node embeddings only to preserve some graph property — typically, graph features essential for a particular problem, as in the examples above, or more generally, nodes similarity. In the following sections, focusing on node embeddings, I briefly overview two main categories of graph representation learning: *shallow embedding* and *graph neural networks*. But I would firstly like to start with an example of the node embedding technique that does not require machine learning — *spectral embedding*.

[2] $d \ll |V|$
[3] `https://en.wikipedia.org/wiki/Cycle_graph`
[4] `https://en.wikipedia.org/wiki/Graph_coloring`
[5] `https://en.wikipedia.org/wiki/K-means_clustering`

1.2 Laplacian eigenmaps

Laplacian eigenmaps (or spectral embedding) [8] is a vital predecessor of graph representation learning. It inspired some node embedding models and has a theoretical connection to some of the models discussed further. This technique stems from a *spectral graph theory* — a field of study concerned with connections between graph properties and the properties of matrices associated with graphs. The most important example of such a matrix is an *adjacency matrix*.

Definition 1.2.1. An *adjacency matrix* of a graph $G = (V, E)$ is a matrix $\mathbf{A} \in \mathbb{R}^{|V|\times|V|}$, defined as $\mathbf{A}[u, v] := [\{u, v\} \in E]$.[6]

Questioning on the connection between topological properties of a graph and algebraic properties of its adjacency matrix, we might get some interesting answers. For example, the powers of this matrix completely describe the numbers of all the possible paths between nodes, or the algebraic multiplicity of 0 as an eigenvalue of the adjacency matrix is a number of graph components [9]. Another central object of the spectral graph theory is a *Laplacian matrix*.

Definition 1.2.2. A *Laplacian* of a graph $G = (V, E)$ is a matrix $\mathbf{L} \in \mathbb{R}^{|V|\times|V|}$, defined as:

$$\mathbf{L}[u, v] := \begin{cases} deg(u) & \text{if } u = v \\ -1 & \text{if } \{u, v\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

Definition 1.2.3. For a graph $G = (V, E)$ the *degrees* of nodes are defined with a mapping $deg : V \to \mathbb{N}$:

$$deg(u) := |\mathcal{N}(u)|,$$

where $\mathcal{N}(u)$ is the set of *direct neighbors* of a node u. Precisely, $\mathcal{N}$ is the mapping $V \to 2^V$, defined as:

$$\mathcal{N}(u) := \{v \in V \mid \{u, v\} \in E\}.$$

We say that the nodes $a, b \in V$ are *adjacent* if $a \in \mathcal{N}(b)$.

This sophisticatedly defined matrix, similarly as an adjacency matrix, has some interesting properties. Consider a graph $G = (V, E)$, its Laplacian $\mathbf{L}$

[6] To simplify the notation, graph nodes are used as matrix indices. It means that we assume that nodes are natural numbers $\{0, \ldots, |V| - 1\}$. This assumption is not restrictive, since V is, by definition, a finite set and one-to-one encoding of nodes with natural numbers always exists. This assumption holds for the rest of the text.

and the vector $\mathbf{v} \in \mathbb{R}^{|V|}$ representing an assignment of some real values to nodes. Then, multiplying $\mathbf{L}$ by a vector $\mathbf{v}$ we get the vector $\mathbf{w}$: $\mathbf{Lv} = \mathbf{w}$. The ith element of $\mathbf{w}$ equals to the sum of differences between the value assigned to $\mathbf{v}$ and the values assigned to its direct neighbors:

$$\mathbf{w}[i] = \sum_{j \in \mathcal{N}(i)} (\mathbf{v}[i] - \mathbf{v}[j]). \tag{1.1}$$

Then, the quadratic form $\mathbf{v}^T\mathbf{Lv}$ equals to the sum of squared differences between adjacent nodes values:

$$\mathbf{v}^T\mathbf{Lv} = \sum_{\{\{i,j\} \in E | i<j\}} (\mathbf{v}[i] - \mathbf{v}[j])^2. \tag{1.2}$$

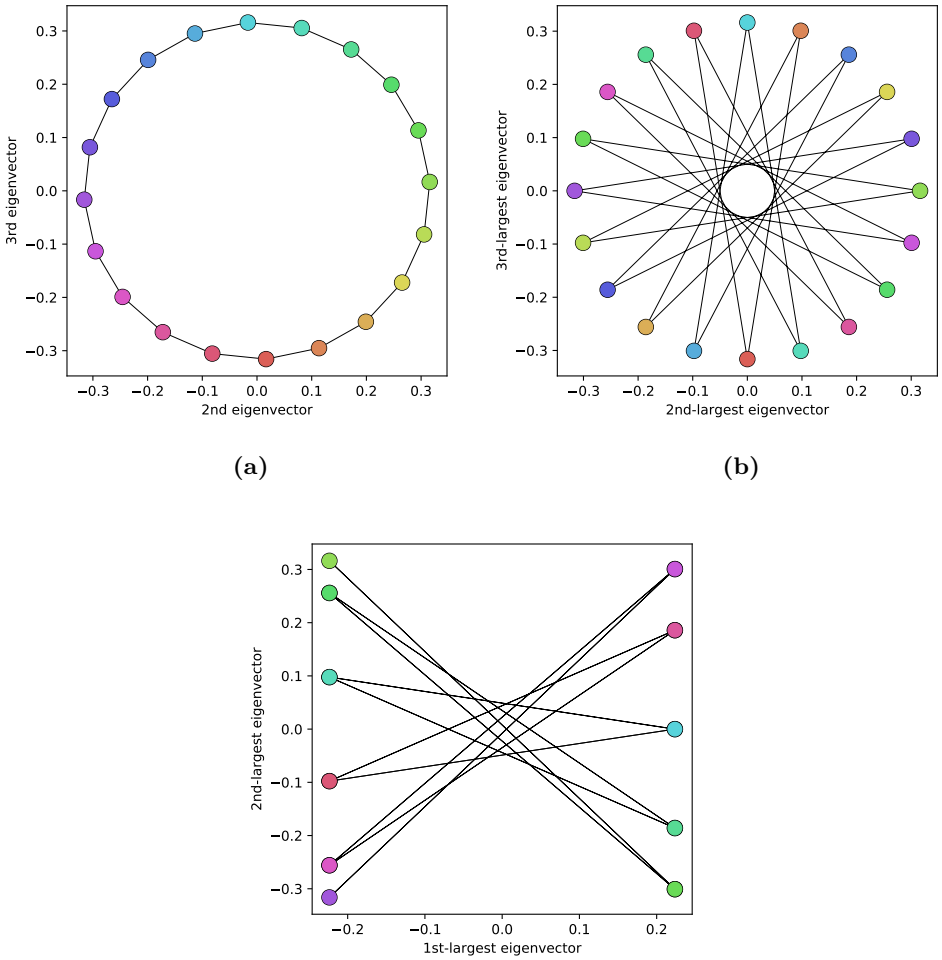
It means that the introduced quadratic form expresses how close the adjacent nodes are from the perspective of the assigned values $\mathbf{v}$. Hence, finding the

$$\arg\min_{\mathbf{v} \in \mathbb{R}^{|V|}} \mathbf{v}^T\mathbf{Lv} \tag{1.3}$$

means finding the embedding of nodes to a 1-dimensional latent space $\mathbb{R}$ that minimizes the distances between the nodes close in the graph. And vice versa, maximizing the given quadratic form we can achieve the embedding that maximizes these distances. It can be proved that this quadratic form is minimized if $\mathbf{v}$ is the eigenvector corresponding to the smallest eigenvalue of $\mathbf{L}$, and maximized if it corresponds to the largest eigenvalue [7]. Considering only unit vectors ($\|\mathbf{v}\| = 1$) and ignoring the trivial case — the smallest zero eigenvalue, which is always present and leads to the embedding of all the nodes at the same point, we can embed nodes into $\mathbb{R}^d$ taking d eigenvectors and using them as coordinates for nodes. For example, if we are interested in placing adjacent nodes close in the 3-dimensional latent space we choose the unit eigenvectors $\mathbf{v_2}, \mathbf{v_3}, \mathbf{v_4}$ corresponding to the 2-nd, 3-rd and 4-th smallest eigenvalues and embed a node i as a vector $(\mathbf{v}_2[i], \mathbf{v}_3[i], \mathbf{v}_4[i])^T$. The illustration of such an embedding can be seen in Figure 1.1. The details and skipped steps of this analysis can be found in [10].

Of course, the Laplacian eigenmaps technique has its limitations. Notably, the only criteria determining the positions of nodes in the latent space is their direct adjacency. It means that this method captures only the local information about nodes, and the deeper structural patterns cannot be utilized. Let us now proceed to the machine learning models that try to overcome this limitation.

[7] `https://sharmaeklavya2.github.io/theoremdep/nodes/linear-algebra/matrices/bounding-quadratic-form-using-eigenvalues.html`



(a) **(b)**

(c)

Figure 1.1: Spectral embedding of the cycle graph $C_{20} = (V, E)$. Colored points represent obtained vector representation of the nodes. The coordinates of a node $i \in V$ are set to **(a)** $(\mathbf{v}_2[i], \mathbf{v}_3[i])$, **(b)** $(\mathbf{v}_{|V|-3}[i], \mathbf{v}_{|V|-2}[i])$, **(c)** $(\mathbf{v}_{|V|-2}[i], \mathbf{v}_{|V|-1}[i])$, where $\mathbf{v}_k (k \in \mathbb{N})$ is a unit eigenvector corresponding to the kth smallest eigenvalue of the graph's Laplacian (or shortly kth eigenvector). The embeddings corresponing to the adjacent nodes are connected with lines.

1.3 Shallow embedding

In this section, I want to sum up the idea delivered by spectral embedding and extend it to machine learning. Having a graph and a defined machine learning task, the most straightforward way is to "properly" place graph nodes into the vector space and then use some general machine learning algorithm. For

example, having a node classification problem, we can use a classic k-NN[8] or Random forest[9] classifier on the obtained embeddings. By saying "properly", I mean preserving a general graph topology, regardless of the task. In this case, any task is reduced to obtaining node embeddings or *encoding* the nodes. Formally, it's defined as finding a mapping ENC: $V \to \mathbb{R}^d$. In the case of Laplacian eigenmaps, this mapping is given by associating nodes with the elements of eigenvectors.

Let us consider an example of the model that **learns** node embeddings of a graph $G = (V, E)$. We can start with initializing them with random values. Since we are interested in embedding of a finite number of vertices $|V|$, the ENC can be equivalently formulated as the matrix of embeddings $\mathbf{Z} \in \mathbb{R}^{|V| \times d}$, considering $\mathrm{ENC}(v) := \mathbf{Z}[v, :]$. Then, having a matrix of embeddings as a variable, we minimize the loss function

$$\mathcal{L}(\mathbf{Z}) := \sum_{u,v \in V} \|\mathrm{ENC}(u) - \mathrm{ENC}(v)\|_2^2 \cdot \mathbf{A}[u, v], \tag{1.4}$$

where $\mathbf{A}$ is an adjacency matrix of G.

Minimizing this loss function, we simply aim to place the embeddings of adjacent vertices close to each other in the sense of a squared Euclidean distance. This strongly resembles the Laplacian eigenmaps technique, but the difference is that now we are directly optimizing the embeddings, which makes this technique considered a machine learning model. I will return to this fact in Section 1.3.1 and let us now generalize such learning of embeddings.

The described model consists of three key parts: a chosen graph property that we aim to preserve in the latent space — nodes adjacency, a chosen way of decoding node embeddings — measuring the Euclidean distance, and the used loss function to find the ENC. By changing these parts, we can construct different models. An essential property of such models is that the ENC function is learned as a "lookup table" $\mathbf{Z}$ for given nodes. That is why such models, that learn embeddings only for specified nodes, are called *shallow embedding* techniques. Importantly, it means that obtained node embeddings can only be used in *transductive* applications — for solving tasks on the fixed nodes, with no generalization to unseen nodes or, for example, new graphs. In Section 1.4 I overview the popular class of models capable of generalization — *Graph neural networks*. Let us now proceed to the formal definition of the mentioned framework.

Shallow embedding can be better seen in terms of *encoder* and *decoder* functions. As mentioned before, having a graph $G = (V, E)$, encoder can be

[8] `https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm`
[9] `https://en.wikipedia.org/wiki/Random_forest`

defined as a function

$$\mathrm{ENC}\colon V \to \mathbb{R}^d \tag{1.5}$$

or equivalently as a matrix $\mathbf{Z} \in \mathbb{R}^{|V| \times d}$ ($\mathrm{ENC}(v) := \mathbf{Z}[v, :]$) and represents obtaining node embeddings. Further, the embedding of a node v is usually denoted as a vector $\mathbf{z}_v := \mathrm{ENC}(v)$. The decoder can be an arbitrary function associating embeddings with the output of interest. Usually, we want to encode some pairwise property of nodes — typically a chosen measure of node similarity $S\colon V \times V \to \mathbb{R}$. Then, decoder is a function decoding this property:

$$\mathrm{DEC}\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}. \tag{1.6}$$

The idea of such an approach is to train the ENC function having fixed DEC to roughly hold

$$(\forall u, v \in V)(\mathrm{DEC}(\mathrm{ENC}(u), \mathrm{ENC}(v)) \approx S(u, v)). \tag{1.7}$$

This equation can be often achieved minimizing the loss function $\mathcal{L}$ expressed as a value of discrepancy between a decoded measure of nodes similarity and an expected one — often as:

$$\mathcal{L}(\mathbf{Z}) := \sum_{u,v \in V} \ell(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v), S(u, v)), \tag{1.8}$$

having $\ell\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as a pairwise measure of values discrepancy. This goal function is typically minimized with Stochastic gradient descent (see for example [11]).

A number of various machine learning models can be defined under this formalism. Models defined in this way are distinguished by the chosen DEC, S, and $\mathcal{L}$(or ℓ) functions. The following two subsections contain a brief overview of two main categories of shallow embedding techniques from this encoder-decoder perspective.

1.3.1 Laplacian eigenmaps based approaches

In terms of the described framework, the model defined in the beginning of Section 1.3 can be defined as:

$$S(u, v) := \mathbf{A}[u, v], \tag{1.9}$$

$$\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) := \|\mathbf{z}_u - \mathbf{z}_v\|_2^2, \tag{1.10}$$

$$\ell(a, b) := a \cdot b. \tag{1.11}$$

We can see that this model aims to preserve node adjacency, which is measured as a squared Euclidean distance in the latent space. And the discrepancy between a true adjacency and a decoded one is measured simply by multiplying

these values. The resemblance of this model with the spectral embedding is not a coincidence. Replacing the adjacency matrix with a Laplacian leads to the same embedding, according to Hamilton [4]. So, inspired by this fact, we can define a number of models with different matrices, including the adjacency matrix. Such machine learning models are among the first emerged approaches. In this work, I do not go into details of these models and proceed to the state-of-the-art approaches.

1.3.2 Matrix factorization

Having data represented as a matrix $\mathbf{M} \in \mathbb{R}^{n \times m}$, in many applications it is convenient to embed them using a matrix factorization in the following form:

$$\mathbf{M} \approx \mathbf{U}\mathbf{V}, \tag{1.12}$$

where $\mathbf{U} \in \mathbb{R}^{n \times d}$, $\mathbf{V} \in \mathbb{R}^{d \times m}$ can be used as a low-dimensional representation of $\mathbf{M}$ (assuming low $d \in \mathbb{N}$).

For example, having the images of faces represented as the rows of $\mathbf{M}$, Nonnegative Matrix Factorization (NMF) [12] learns a parts-based representation of faces, so that the d columns of $\mathbf{U}$ represent the so-called basis images and $\mathbf{V}$ columns are the so-called encodings of original faces. The idea is that any face is treated as a nonnegative linear combination of the basis faces, which can be highly useful in, for example, face recognition tasks, and has psychological justifications. Similarly, for example, having n users and m movies, the matrix $\mathbf{M}$ can be constructed so that $\mathbf{M}[i, j] := 1$ if user i watched the movie j and $\mathbf{M}[i, j] := 0$ otherwise. Then, some recommender systems [13], learn embeddings of users and movies encoded in matrices $\mathbf{U}$ and $\mathbf{V}$ respectively.

Inspired by results of such approaches, a number of matrix factorization models emerged in the domain of a graph representation learning. Probably, the first one was the model by Ahmed et al. [14]. The idea of this model, usually reffered to as Graph Factorization (GF), is to learn the matrix of node embeddings $\mathbf{Z} \in \mathbb{R}^{|V| \times d}$ of a graph $G = (V, E)$ to hold

$$\mathbf{A} \approx \mathbf{Z}\mathbf{Z}^T, \tag{1.13}$$

where $\mathbf{A}$ is an adjacency matrix of G. In terms of the encoder-decoder perspective it means using an adjacency matrix as a measure of similarity and expressing it geometrically (decoding) with a standard inner product:

$$S(u, v) := \mathbf{A}[u, v], \tag{1.14}$$

$$\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) := \langle \mathbf{z}_u, \mathbf{z}_v \rangle. \tag{1.15}$$

Then, the idea of the objective is:

$$\mathcal{L}(\mathbf{Z}) := \frac{1}{2} \sum_{u,v \in E} (\langle \mathbf{z}_u, \mathbf{z}_v \rangle - \mathbf{A}[u, v])^2. \tag{1.16}$$

Graph Factorization utilizes a linearly scalable algorithm for the loss function minimization constructed based on the one given by Equation 1.4.

The authors of the GraRep model [15] argue that the models like GF may be inefficient since they only capture *1-hop* neighborhoods of nodes, using $\mathbf{A}$ as a similarity measure. As mentioned at the beginning of Section 1.2, powers of $\mathbf{A}$ are related to the paths of different lengths in G. Inspired by this fact, GraRep uses the powers of $\mathbf{A}$ instead of $\mathbf{A}$ to capture k-hop neighborhoods of nodes ($k \in \mathbb{N}$). Moreover, it performs the factorization in a more sophisticated way basing on the Singular value decomposition (SVD)[10]. These facts make this model potentially more powerful but unscalable and time-consuming. From the perspective of an encoder-decoder framework, GraRep uses the different powers of $\mathbf{A}$ as a similarity measure S and again is based on the standard inner product decoder, implied by matrix factorization.

Definition 1.3.1. A *walk* of length $n \in \mathbb{N}_+$ on graph $G = (V, E)$ starting from node v_1 is a tuple $(v_1, \ldots, v_n) \in V^n$ if the following holds:

$$(n > 1) \implies (\forall i \in \{1, \ldots, n-1\})(\{v_i, v_{i+1}\} \in E).$$

Definition 1.3.2. A *path* on graph $G = (V, E)$ is a walk $(v_1, \ldots, v_n)$ if all nodes $v_1, \ldots, v_n$ are distinct.

Definition 1.3.3. We say that a node v is *reachable* from the node u if the path $(u, \ldots, v)$ exists. A *k-hop neighborhood* of a node u is a set of all the nodes reachable from u with the paths of length at most k.

Similar models include HOPE [16], which offers a more flexible choice of a similarity measure and is originally designed to extend the idea of matrix factorization to directed graphs.

1.3.3 Neighborhood sampling

Assigning vector representations to graph nodes is similar to the popular idea in NLP to assign vector representations to words. Considering an analogy

text corpus — graph, word — node, word context — node neighborhood

several successful machine learning techniques on graphs emerged. Models based on the famous word2vec skip-gram model by Mikolov et al. [17] achieved especial success. This NLP model is designed to produce word embeddings.

[10] `https://en.wikipedia.org/wiki/Singular_value_decomposition`

The key idea behind word2vec is that similar words share contexts. This assumption can be applied to graphs — similar nodes share contexts (or neighborhoods). An ambiguity of the notion "nodes neighborhood" has induced the emergence of various models. Formally, node neighborhoods can be defined by any function $N_s : V \to 2^V$ associating nodes with their neighborhoods according to *sampling strategy* s.

A model called node2vec [18] introduces a flexible way to sample node neighborhoods using random *walks*. In this case N_s associates nodes with specific random walks starting from them. Based on model hyperparameters, which affect random walks, neighborhoods are generated more in BFS or DFS manner. Generated walks are then used as training sentences for the skip-gram model. In terms of encoder-decoder perspective it means training ENC with S defined by empirical probabilities of nodes pairwise co-occurrence in generated walks and DEC given as a reconstruction of these probabilities:

$$\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) := \frac{e^{\langle \mathbf{z}_u, \mathbf{z}_v \rangle}}{\sum_{n \in V} e^{\langle \mathbf{z}_u, \mathbf{z}_n \rangle}}, \tag{1.17}$$

where inner product can be treated as a measure of similarity. Then $\mathcal{L}$ can be defined as

$$\mathcal{L}(\mathbf{Z}) := \sum_{u,v \in V} -\log\left(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v)\right)[v \in N_s(u)]. \tag{1.18}$$

Similar models include DeepWalk [19], which differs mainly in random walks sampling strategy, and LINE [20] which samples *1-hop* and *2-hop* neighborhoods of nodes instead of generating walks. Node2vec emerged as an improvement of these models. Interestingly, Qiu et al. [21] show that these models perform an implicit matrix factorization, as well as models described in Section 1.3.2.

1.3.4 Summary

The goal of this section was to briefly introduce the ideas of shallow embedding. Many of these models are considered state-of-the-art of graph representation learning. Let us now consider the main limitations of this approach, induced by their "shallowness": representing ENC as a lookup table $\mathbf{Z}$:

- Embeddings are learned only for specific nodes. It means that shallow embedding models can be applied only in transductive applications: solving tasks on "seen" nodes, with no generalization to "unseen" nodes or new graphs.

- Embeddings do not share any parameters. It means that the memory complexity of such approaches is at least $\mathcal{O}(|V|)$. Moreover, Hamilton [4] states that the parameters sharing can act as a powerful form of a model regularization.

Moreover, there are two more important facts:

- The majority of described models cannot utilize graph properties as node features or edge features, which obviously, may be highly preferable in many real-world tasks. Of course, there exist corresponding extensions, but they are not necessarily natural.
- Mentioned models learn embeddings in an unsupervised manner regardless of the task, which can be as good as bad but remains the typical property of shallow embedding.

1.4 Graph neural networks

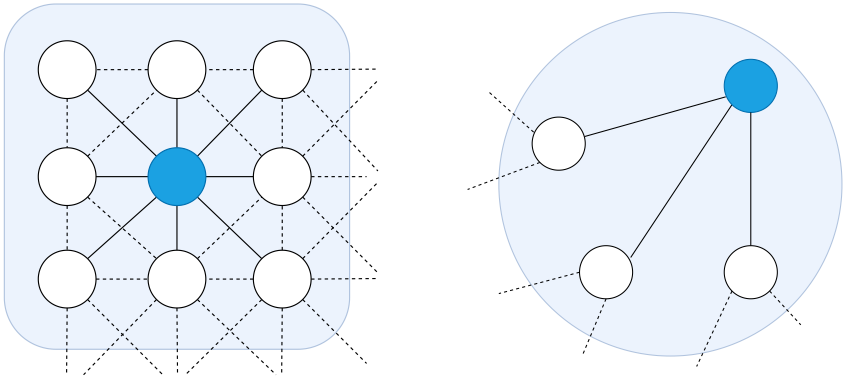
In the previous section, I shortly described some well-known machine learning models based mainly on either matrix factorization techniques or neighborhood sampling approaches. The goal of the section, as its name suggests, is to provide a brief introduction to the graph machine learning models based on deep learning. The recent and ubiquitous success of this branch of machine learning could not leave graph representation learning unaffected.

In computer vision, convolutional neural networks (CNNs) show impressive results in various tasks. The idea of these models is to sequentially apply 2D convolutional kernels (a filter) to transform the image by aggregating neighboring pixels (see for example [22, Chapter 9]). Learning kernels enables to efficiently solve computer vision tasks with $\mathcal{O}(1)$ parameters and is justified by the analogy to the mammalian vision system.

The first graph neural networks (GNNs) are strongly inspired by CNNs [23, 24]. An image can be treated as the special case of a graph: a fully-connected grid graph (see Figure 1.2b (a)). Then, considering an analogy

image — graph, pixel — node features, neighboring pixels — node neighborhood,

convolutional neural networks can be generalized to graphs. We do not need any efforts to associate an image with a graph (as shown in Figure 1.2b), or pixels with node features (since they both can be represented as real vectors),



(a) **(b)**

Figure 1.2: A comparison of 2D convolution and graph convolution. **(a)** 2D convolution: each pixel can be treated as a graph node. Then, in CNNs convolution is applied to the k-hop neighborhood of each node determined by the filter size; **(b)** graph convolution: different from the image data, node neighborhoods are not ordered and variable in size.

but an extension of the notion *convolution* to a graph domain is a cornerstone of GNNs problematics.

A discrete convolution kernel "distinguishes" the input vector elements by their positions. For example, in the case of CNNs, a 3×3 kernel of 2D convolution is always applied to 3×3 subimages so that only the corresponding elements are multiplied. It means that the input for convolution is always fixed in size and ordered. Obviously, this statement does not hold in the graph domain: nodes direct neighborhoods vary in size and cannot be naturally ordered. It leads to the conclusion that the analogue of convolution on graphs should be a multiset aggregator function. Namely, for a graph $G = (V, E)$ and some $n \in \mathbb{N}$:

$$\text{AGGREGATE} : 2^{\{\mathbf{x}_v | v \in V\}} \to \mathbb{R}^n, \tag{1.19}$$

where $\mathbf{x}_v$ denotes the feature of a node v (see Definition 1.4.1) .Then, the different GNN architectures are distinguished by the used AGGREGATE functions. Let us firstly sum up this framework and then proceed to some well-known architectures.

1.4.1 GNN framework

The goal of this section is to introduce a general framework of deep learning on graphs — a graph neural network — regardless of the specific architecture (the chosen AGGREGATE function). Consider a graph $G = (V, E)$ and its *node features* $\mathbf{X}$.

Definition 1.4.1. The *node features* of a graph $G = (V, E)$ are a matrix $\mathbf{X} \in \mathbb{R}^{|V| \times n}$, where $n \in \mathbb{N}$ is a number of *features*. The *features* or a *feature vector* of a node $v \in V$ is then a vector $\mathbf{x}_v := \mathbf{X}[v, :]$.

Then, to produce the embedding $\mathbf{z}_v \in \mathbb{R}^n$ of a node $v \in V$, a Graph neural network sequentially transforms its *hidden embedding* $\mathbf{h}_v$ as follows:

$$\text{(i)} \quad \mathbf{h}_v^{(0)} := \mathbf{x}_v \tag{1.20}$$

$$\text{(ii)} \quad \mathbf{h}_v^{(k+1)} := \text{AGGREGATE}^{(k)}(\{\mathbf{h}_u^{(k)} \mid u \in \mathcal{N}(v) \cup v\}) \tag{1.21}$$

$$\text{(iii)} \quad \mathbf{z}_v := \mathbf{h}_v^K \text{ for some } K \in \mathbb{N} \tag{1.22}$$

This iterative process is often referred to as a *message passing*, since at each iteration nodes collect messages (hidden embeddings) from their direct neighbors and pass their own message. See Figure 1.3 for the illustration.

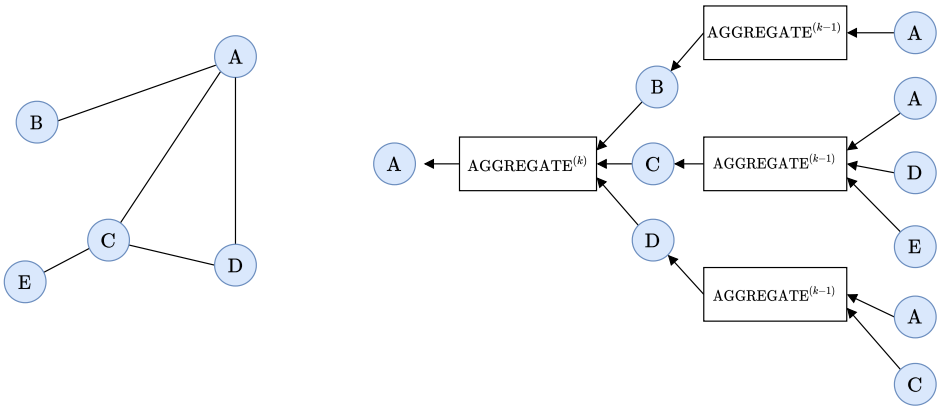


Figure 1.3: Illustration of how a single node A aggregates messages from its local neighborhood. The visualization shows a computation graph of a two-layer version of graph neural network model.

At the kth iteration of message passing, nodes collect information from their whole k-hop neighborhoods, as seen in Figure 1.3 for 2 iterations. That is why K represents a *depth* of the graph neural network and each iteration (Equation 1.21) represents a *layer*. In practical tasks, K is usually set to be very low (2-3), since the diameter[11] of real-world graphs is usually low [25]. Moreover, relevant to the nodes information is usually contained in their close neighborhoods.

[11] the length of the longest path

Assuming that AGGREGATE$^{(k)}$ functions are parametric and differentiable, we can construct different task-driven loss functions to learn their parameters. For example, having a binary node classification task, we can use a binary crossentropy:

$$\mathcal{L} := \sum_{v \in \mathcal{D}} y_v \log(\sigma(\mathbf{z}_v^T \boldsymbol{\theta})) + (1 - y_v) \log(1 - \sigma(\mathbf{z}_v^T \boldsymbol{\theta})), \tag{1.23}$$

where $\mathcal{D} \subset V$ is a training set of nodes, $y_v \in \{0, 1\}$ is a true label of a node v and $\boldsymbol{\theta}$ is a trainable vector of classification weights. Nevertheless, nodes can also be embedded in an unsupervized manner using the appropriate loss function (see for example [26]).

Note that by analogy to the shallow embedding models, Equation 1.23 defines a way of decoding node embeddings encoded with the Graph neural network. It means that GNNs are analogous to ENC functions defined in the Section 1.3. Under such a perspective, we can see a crucial difference between shallow embedding techniques and Graph neural networks. In the first case, the ENC is learned as a lookup table (Equation 1.5), but in the case of GNNs, it is learned as a function that maps any node along with its neighborhood to a real vector. I return to this fact in the Section 1.4.3

1.4.2 Popular architectures

In this subsection, I want to introduce two well-known architectures of graph neural networks.

GCN

Graph convolutional neural network (GCN) [24] layers are defined as follows:

$$\mathbf{h}_v^{(k+1)} := \sigma\left(\mathbf{W}^{(k)} \sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{\mathbf{h}_u^{(k)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right), \tag{1.24}$$

where σ is an activation function such as $\mathrm{ReLU}(\cdot) := \max(0, \cdot)$ and $\mathbf{W}^{(k)}$ is a trainable matrix. We can observe that in in this case the aggregation is based on the normalized sum of the node neighborhood. This sum is then transofrmed with a linear operator and activated with a non-linear function, typically for deep learning models. Authors state that the choice of this specifical normalization is based on the "localized first-order approximation of spectral graph convolutions", which is the concept related to spectral embedding (see Section 1.2) and can be considered a generalization of a discrete convolution.

GraphSAGE

The GraphSAGE[12] [26] model has emerged as a generalization and improvement of GCNs. It defines a more flexible layer:

$$\mathbf{h}_v^{(k+1)} := \sigma\left(\mathbf{W}^{(k)} \cdot \mathrm{AGG}(\{\mathbf{h}_u^{(k)} \mid u \in \mathcal{N}(v)\}) \oplus \mathbf{B}^{(k)} \cdot \mathbf{h}_v^{(k)}\right), \tag{1.25}$$

where σ is again an activation function and $\mathbf{W}^{(k)}$, $\mathbf{B}^{(k)}$ are the trainable matrices. AGG can be an arbitrary multiset aggregator function. Authors of the model propose multiple ones, including a simple mean aggregation and a more sophisticated one based on training the LSTM Recurrent neural network [27] with random permutations of multiset elements. Moreover, authors propose to sample neighbors $\mathcal{N}(v)$ instead of using a full set at a time to effectively trade off performance and runtime.

1.4.3 Summary

In this section, I introduced the elements of a powerful deep learning graph framework: a graph neural network. Now I want to sum up the key properties of GNNs comparing to other approaches (shallow embedding models). First of all, as mentioned in Section 1.4.1, using a GNN, the nodes are encoded with a learned parameterized function, not with a lookup table. There are two main effects[13] of this fact:

- Graph neural networks can be used in *inductive* applications. It means that using a GNN, we are not restricted to training nodes or a training graph. For example, after adding a new user to a social network, we do not have to recompute embeddings for the whole graph, but only one GNN inference is enough to obtain the embedding of a new user.
- Node embeddings are learned sharing the parameters. It means that the number of model parameters does not depend on the input graph size and can be possibly reduced to $\mathcal{O}(1)$.

Let us moreover consider two important facts:

- Graph neural networks inherently assume the presence of node features. It can be a shortcoming of such an approach, but node features can

[12] SAmple and aggreGatE

[13] Note that not all the graph neural networks should necessarily have the following properties, however, they are a key factor of the GNN framework success.

always be generated according to graph topological properties. For example, one could construct a feature vector for a node considering its degree, local clustering coefficient[14] or any other topological feature.

- GNNs can be trained with an arbitrary loss function, and therefore can be naturally used in a supervised manner, as shown in Section 1.4.1.

These facts make graph neural networks considered the main representative of graph representation learning. Note also that the paper introducing GCN [24], which is often considered a first GNN, was published quite recently, in year 2016. According to Leskovec [25] (Autumn 2019), the "Graph neural network" keyword was as frequent as an entire "NLP" one in the ICLR[15] 2019 keywords distribution, meaning the intense research in this area. GNN models convolve with other branches of deep learning and produce new models. See Zhou et al. [28] (2021) for the taxonomy of various GNN architectures.

1.5 Machine learning on bipartite graphs

Having a machine learning problem defined for *bipartite* graph, we can utilize any model discussed in this chapter. However, having such significant insight into the graph topology, it may be useful to utilize it. That is why there exist machine learning models designed specifically for bipartite networks.

Definition 1.5.1. A graph $G = (V, E)$ is *bipartite* if two disjoint sets U and I exist, such that $V = U \cup I$ and $E \subseteq \{\{u, i\} \mid u \in U \wedge i \in I\}$. U and I are said to be the *partitions*[16] of a graph G.

Definition 1.5.2. An *adjacency matrix* of a bipartite graph $G = (V, E)$ with partitions U and V is a matrix $\mathbf{A} \in \mathbb{R}^{|U|\times|I|}$ defined as $\mathbf{A}[u, v] := [\{u, v\} \in E]$.

The notion of the adjacency matrix of a bipartite graph is typically overridden with the Definition 1.5.2. Then, the direct application of some models discussed above in this chapter to a bipartite graph with its newly defined adjacency matrix leads to similar approaches as the ones invented for user-item systems, not necessarily considered as graphs. In this work, I do not dive into the problematics of such models but only overview two important architectures designed for specific bipartite networks.

[14] `https://en.wikipedia.org/wiki/Clustering_coefficient`

[15] `https://en.wikipedia.org/wiki/International_Conference_on_Learning_Representations`

[16] I use letters U and I to denote partitions since in this work dynamic bipartite networks typically represent interactions between users and items.

1.5.1 Implicit feedback

The term *implicit feedback* is used to describe specific relations in the user-item systems. Contrary to *explicit feedback* which describes explicit user actions, such as rating the movie or leaving a comment below the YouTube video, implicit feedback describes "passive" preferences. For example, the fact that the user watches the same movie every month can be treated as the user's positive feedback about the movie. Having a set of user U and a set of items I, this idea is typically formalized as a matrix $\mathbf{R} \in \mathbb{R}^{|U|\times|I|}$, where $\mathbf{R}[u,i]$ represents the number of times user u interacted with item i (e.g. a number of purchases, clicks or views). Obviously, $\mathbb{R}$ can be treated as the adjacency matrix of a graph representing user-item relations.

Alternating least squares

Alternating Least Squares (ALS) [29] is an algorithm used with implicit feedback data. Having implicit feedback $\mathbf{R}^{|U|\times|I|}$ describing relations between users U and items I, the idea of Alternating least squares application is to learn user embeddings $\mathbf{x}_u \in \mathbb{R}^d$ for each user $u \in U$ and item embeddings $\mathbf{y}_i \in \mathbb{R}^d$ for each item $i \in I$ minimizing the loss function

$$\mathcal{L}(\mathbf{X},\mathbf{Y}) := \sum_{u\in U}\sum_{i\in I} c_{ui}(p_{ui} - \langle \mathbf{x}_u, \mathbf{y}_i\rangle)^2 + \lambda_1\left(\sum_{u\in U}\|\mathbf{x}_u\|_2^2 + \sum_{i\in I}\|\mathbf{y}_i\|_2^2\right), \quad (1.26)$$

where d is a latent space dimensionality (or a number of factors), $\mathbf{X} \in \mathbb{R}^{|U|\times d}$ and $\mathbf{Y} \in \mathbb{R}^{d\times|V|}$ are the matrices of user and item embeddings: $\mathbf{x}_u = \mathbf{X}[u,:]$[17], $\mathbf{y}_i = \mathbf{Y}[:,i]$, and $\lambda_1 \in \mathbb{R}_0^+$ is a hyperparameter controlling the influence of L_2-regularization[18]. A constant $p_{ui} \in \{0,1\}$ represents the *preference* of a user u for a product i defined as follows:

$$p_{ui} := \begin{cases} 1 & \text{if } \mathbf{R}[u,i] > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (1.27)$$

Next, $c_{ui} \in \mathbb{R}$ represents the *confidence* in the preference observation p_{ui}. One of the possible ways to define it is as

$$c_{ui} := 1 + \lambda_2 \cdot \mathbf{R}[u,i], \quad (1.28)$$

where $\lambda_2 \in \mathbb{R}_0^+$ is another hyperparameter. Note how this algorithm resembles the models discussed in Section 1.3.2.

[17] Similarly as in Definition 1.2.1, U and I are assumed to be $\{0,1,\ldots,|U|-1\}$ and $\{0,1,\ldots,|I|-1\}$ for the notation simplicity.

[18] See Tikhonov regularization
`https://en.wikipedia.org/wiki/Regularization_(mathematics)`

Bayesian personalized ranking

Rendle et al. [30] argue that zero values of $\mathbf{R}$ should be treated as missing ones. That is why they propose to consider implicit feedback in form of triples:

$$R := \Big\{(u,i,j) \in U \times I \times I \;\Big|\; \mathbf{R}[u,i] > 0 \wedge \mathbf{R}[u,j] = 0\Big\}. \tag{1.29}$$

Entry $(u,i,j) \in R$ represents that user u prefers item i to the item j. It means that the only assumption made about data is that users prefer items that they interacted with to items they did not.

Then, based on the probabilistic point of view on this assumption they propose to maximize the following function:

$$\mathcal{L}(\mathbf{X},\mathbf{Y}) := \sum_{(u,i,j)\in R} \ln \sigma(\hat{x}_{uij}) - \lambda_1 \cdot \|\theta\|_2^2, \tag{1.30}$$

where σ is a sigmoid[19] function and

$$\hat{x}_{uij} := \hat{x}_{ui} - \hat{x}_{uj} := \langle \mathbf{x}_u, \mathbf{y}_i \rangle - \langle \mathbf{x}_u, \mathbf{y}_j \rangle. \tag{1.31}$$

Here, $\mathbf{x}_u \in \mathbb{R}^d$ for each $u \in U$ and $\mathbf{y}_i \in \mathbb{R}^d$ for each $i \in I$ are the user and item embeddings which can be the output of an arbitrary model. For example, one can factorize the implicit feedback matrix similarly as in Equation 1.26. Then, θ is a real vector representing the parameters of the chosen model, $\lambda_1 \in \mathbb{R}_0^+$ controls the L_2-regularization and $d \in \mathbb{N}_+$ is a latent space dimensionality.

[19] `https://en.wikipedia.org/wiki/Sigmoid_function`

Chapter 2

Machine learning on dynamic graphs

Approaches discussed in Chapter 1 are designed for static graphs with fixed nodes and edges (see Definition 1.0.1). However, a vast of real-world networks change in time. For example, consider a social network with nodes representing users and edges representing a friendship relation. We may want to classify users by some criteria [31], for example by their interests, or alternatively by their behavioral roles [32]. Having an e-shop, users and items can be treated as graph nodes, and purchases as edges. Then we may be interested in predicting the next user's purchase [33, 34, 3], for example, to provide an e-shop with personalized advertising. Having graph nodes representing researchers and edges representing co-authorship, an interesting task is to predict the collaborations [35]. Moreover, for instance, tracking the contacts between the attendees of a conference, our goal may be to analyze some spreading process, for instance, a virus, passing from person to person [36]. Another interesting example is a prediction of animal behavior: tracking interactions within a group of baboons, and having four types of this group's activities (sleeping, hanging out, etc.), one may be interested in predicting the next-day activities based on the past behavior [2]. Obviously, all these tasks require considering the dynamics of data to solve them as efficiently as possible, since the underlying graphs change over time and data at different timepoints are dependent on each other.

Note that in these examples not only the problems are totally different but the graph dynamics are expressed differently as well. For example, in animal behavior example, the interaction between two baboons represents an addition or removal of the edge, while in a co-authorship network, edges are only added as time passes which means that the graph only grows. Furthermore, the interests of people change, which means that in a social network, classifying

the users based on their interests, we should consider that labels change in time and thus classification should be performed for each timestamp. However, a user's behavioral role is a notion associated with the whole history of the user's activity and, in this case, the whole sequences are classified. The goal of this chapter is to overview modern approaches to solve such problems utilizing graph dynamics.

2.1 Dynamic graph

To discuss the machine learning approaches capable of capturing graph dynamics, we should firstly properly formalize the essence of a *dynamic graph*. As stated at the beginning of this chapter, graph dynamics may be expressed differently in different applications. Indeed, sometimes a constructed graph only changes its topology as time passes, but in other applications, for example, node or edge features may also evolve. That is why the notion of a *dynamic graph* is usually defined differently, driven by the task.

There exist various approaches to formalize the concept of a *dynamic graph* [37, 38, 39, 40]. Probably, the most general way to define it mathematically is given by Casteigts et al. [37]. They propose to define a *Time-Varying Graph* as follows. Consider a *time span* $\mathcal{T} \subseteq \mathbb{T}$, where $\mathbb{T}$ is assumed to be either $\mathbb{N}$ or $\mathbb{R}^+$. Then, a *Time-Varying Graph* can be defined as a tuple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}, \rho, \zeta, \psi, \varphi)$. Here, $\mathcal{V}$ is a non-empty finite set of *nodes* and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathcal{F}$ is a set of edges, similarly as in the static graph (see Definition 1.0.1). So far, the only difference is an implicit definition of edge feature $\mathcal{F}$ that can be of any nature. Then:

- $\rho : \mathcal{E} \times \mathcal{T} \to \{0, 1\}$ is an *edge presence* function indicating whether a given edge is present at a certain time point.
- $\zeta : \mathcal{E} \times \mathcal{T} \to \mathbb{T}$ is an *edge latency* function representing a time necessary to "cross" the given edge at a certain time.
- $\psi : \mathcal{V} \times \mathcal{T} \to \{0, 1\}$ is a *node presence* function indicating whether a given node is present at a certain time point.
- $\varphi : \mathcal{V} \times \mathcal{T} \to \mathbb{T}$ is a *node latency* function representing a time necessary to "travel" by the given node at a certain time.

Using such a definition we can formalize the majority of (if not all) real-world dynamic networks. But in many applications, we are not interested in all of the possible temporal graph changes and thus we do not require such a highly expressive model which may be overcomplicated. A big class of problems, including the ones introduced at the beginning of this chapter, can be defined using the Definition 2.1.1.

Definition 2.1.1. *Dynamic graph*[20] is a tuple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$, where $\mathcal{V}$ is a non-empty finite set of *nodes* or *vertices*, $\mathcal{E} \subseteq \{ (u, v, t) \mid u, v \in \mathcal{V} \wedge u \neq v \wedge t \in \mathcal{T} \}$ is a set of *edges*, *links*, *events* or *interactions*[21], and $\mathcal{T} \subseteq \mathbb{T}$ is a *time span*, where $\mathbb{T} := \mathbb{R}_0^+$ represents the *temporal domain*.

Note that with the dynamic graph defined in a such way we are restricted to certain systems: we always assume the fixed entities which do not implicitly change over time, and the instantaneous interactions between them. Mainly, in the literature related to machine learning on dynamic graphs, exactly this type of time-varying networks is considered, since they are highly important and ubiquitous in the real world.

2.2 Dynamic graph representation learning

In Chapter 1, I shortly described the main ideas of graph representation learning. This paradigm includes two main types of approaches: shallow embedding and Graph neural networks. Being the state of the art of machine learning on graphs, it serves as a basis for the majority of techniques for dealing with problems on dynamic graphs. In this section, I want firstly to introduce several important concepts related to *dynamic graph representation learning*: *temporal granularity*, *time decay* and *temporal smoothness*, and then proceed to the methods themselves.

2.2.1 Temporal granularity

Graph representation learning techniques that capture network dynamics are typically classified by how they "treat" time, i.e for which dynamic graph representation they are designed. There exist two main categories: *discrete-time approaches* and *continuous-time approaches*.

Having a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$, the most straightforward way to capture the network dynamics with the graph representation learning techniques is to split the graph into $T \in \mathbb{N}$ static graphs, called *snapshots*:

$$G^{(0)}, G^{(1)}, \ldots, G^{(T-1)}. \tag{2.1}$$

[20]The naming of this structure is inconsistent across the literature. It may be referred to with different word combinations, including the pairs from {dynamic, temporal, time-varying, continuous-time dynamic} × {graph, network} and others. Further in this work, whenever I use one of these combinations I mean the *dynamic graph*.

[21]Note that the edges are implicitly ordered. However, in this work this fact is typically not significant and I usually assume that the direction is not important. I use this definition of edges to be consistent with the literature.

This can be achieved by selecting $T + 1$ timestamps $t_0, t_1, \ldots, t_T \in \mathbb{T}$ and defining the snapshots as follows:

$$G_k := (V, \{\{u, v\} \mid ((v, u, t) \in \mathcal{E} \vee (u, v, t) \in \mathcal{E}) \wedge (t_k \leqslant t < t_{k+1})\}). \quad (2.2)$$

A machine learning model which operates with graph snapshots is said to be a *discrete-time* [41, 42, 43]. The motivation of such an approach is to apply the standard graph representation learning techniques on snapshots and use the results as input for the models designed for sequences processing, or simply aggregate them if necessary. The number of snapshots is usually set to be much lower than the cardinality of a graph time span $\mathcal{T}$, due to the computational complexity. It means that such models lose some temporal information aggregating the events.

Contrary to discrete-time approaches, *continuous-time* approaches directly process a dynamic graph. Such models have the maximal potential expressivity since they do not lose temporal information. However, to build such models based on the classic graph representation learning approaches, it is necessary to modify them, which makes continuous-time approaches generally more complex. Consequently, continuous-time approaches are far not as popular as the ones working with graph snapshots [42, 43]. I will describe some of the models belonging to both classes further in this chapter.

2.2.2 Time decay

When dealing with temporal data it is often assumed that more recent records have higher significance. This idea is typically realised via a *time-decay* function $\Gamma : \mathbb{T} \times \mathbb{T} \to \mathbb{R}$ which associates pairs of timestamps with their weighted distance. Some typical examples of Γ [44, 33] include :

1. *Passage* time decay

$$\Gamma(t_1, t_2) := \begin{cases} 1 & \text{if } |t_1 - t_2| < \sigma \\ 0 & \text{otherwise} \end{cases} \quad (2.3)$$

2. *Triangle* time decay

$$\Gamma(t_1, t_2) := \begin{cases} 1 - \frac{|t_1 - t_2|}{\sigma} & \text{if } |t_1 - t_2| < \sigma \\ 0 & \text{otherwise} \end{cases} \quad (2.4)$$

3. *Gaussian* time decay

$$\Gamma(t_1, t_2) := e^{\frac{-(t_1 - t_2)^2}{2\sigma^2}} \quad (2.5)$$

Here, $\sigma \in \mathbb{R}_+$ is a parameter controlling time decay. For the visualization see Figure 2.1.

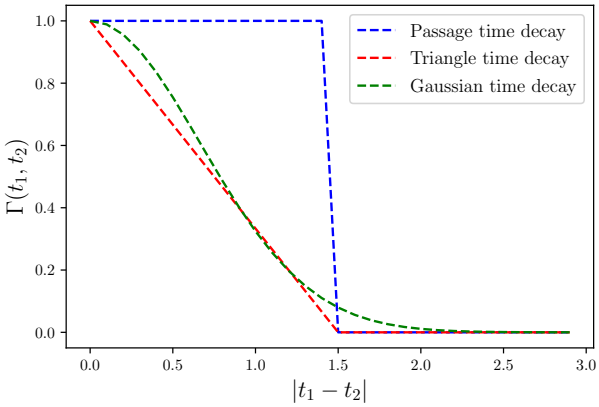


Figure 2.1: Illustration of the time-decay functions. Here, σ is set to 1.5.

2.2.3 Temporal smoothness

Many models try to capture network dynamics assuming the *temporal smoothness*. Consider for example a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ and a discrete-time model that produces node embeddings $\mathbf{z}_v^{(0)}, \mathbf{z}_v^{(1)}, \ldots, \mathbf{z}_v^{(T-1)}$ for a node $v \in \mathcal{V}$ at each of the $T \in \mathbb{N}$ snapshots. Then the *temporal smoothness* means that vectors $\mathbf{z}_v^{(0)}, \mathbf{z}_v^{(1)}, \ldots, \mathbf{z}_v^{(T-1)}$ form a trajectory in the latent space. Similarly, for example, getting a vector representation of dynamic graph edges, assuming the temporal smoothness, we may expect adjacent edges with similar timestamps to be close in the latent space. For the illustration see Figure 2.2.

2.3 Shallow embedding of dynamic graphs

This section aims to cover modern shallow embedding approaches designed for dynamic graphs. I rely on the surveys by Kazemi et al. [41] (2020) and Barros et al. [43] (2021), and the other, further mentioned, works I am aware of. All the described approaches are strongly inspired by techniques discussed in Section 1.3 and preserve the general framework. That is why at some points I only describe the key parts of time-aware models, relying on their obvious association with the ones described in Chapter 1.

2.3.1 Discrete-time approaches

As discussed in Section 2.2.1, dealing with discrete time means working with graph snapshots. That is why in this section, I consider shallow embedding models designed for the snapshots $G^{(0)}, G^{(1)}, \ldots, G^{(T-1)}$ of a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$.

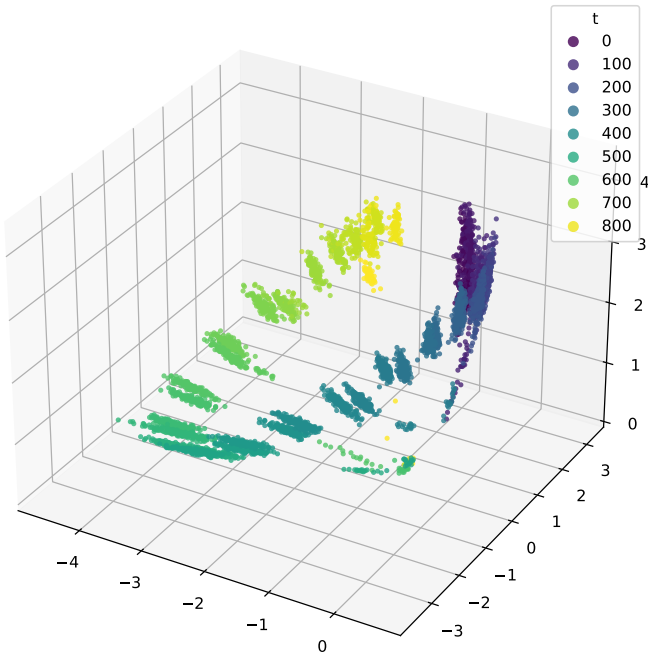


Figure 2.2: Example of temporal smoothness. Points in the plot represent 3-dimensional edge embeddings of the real-world co-presence network [45]. Nodes $\mathcal{V}$ of this network $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ represent the attendees of a scientific conference and an edge $(u, v, t) \in \mathcal{E}$ represents that the person u is in contact with the other attendee v at time t. The picture shows embeddings obtained with weg2vec [36]. This model aims to place adjacent edges close in the latent space and preserve temporal smoothness.

2.3.1.1 Snapshots aggregation

The easiest way to directly apply classic shallow embedding techniques for any downstream task on a dynamic network is to convert it to a single static snapshot G. Then, any technique described in Section 1.3 can be straightforwardly utilized. For example, Liben-Nowell and Kleinberg [46] follow this idea while studying the link-prediction problem for social networks.

Some succeeding approaches try to capture more temporal information by considering a matrix of weights $\mathbf{W} \in \mathbb{R}^{|V| \times |V|}$:

$$\mathbf{W}[i, j] := \sum_{t=0}^{T-1} \Gamma(T, t) \cdot \mathbf{A}^{(t)}[i, j], \tag{2.6}$$

where $\mathbf{A}^{(t)}$ is an adjacency matrix of the graph $G^{(t)}$, and Γ is an arbitrary time-decay function (see Section 2.2.2). For example, Ahmed et al. [47] set $\Gamma(T, t) := \gamma^{T-t}$ for some $\gamma \in [0, 1]$ and then generate random walks on G that are biased according to $\mathbf{W}$. Similarly, Ibrahim and Chen [48] construct

the same weight matrix, but add "self-loops" $\mathbf{W}[i,i] := 1$ for all $i \in \mathcal{V}$ and use different node similarity measures S based on this matrix. The idea of such approaches is that recent events carry more relevant information. The described models can be useful, however they are clearly doomed to the loss of temporal information.

2.3.1.2 Latent snapshots aggregation

Rather than aggregating snapshots to one graph in order to utilize static shallow embedding techniques, we can apply them to each snapshot and then aggregate the obtained embeddings. To get the embedding $\mathbf{z}_v$ of a node $v \in \mathcal{V}$, it means to obtain node embeddings $\mathbf{z}_v^{(0)}, \mathbf{z}_v^{(1)}, \ldots, \mathbf{z}_v^{(T-1)}$ for each of the T snapshots and then aggregate them:

$$\mathbf{z}_v := \sum_{t=0}^{T-1} \Gamma(T,t) \cdot \mathbf{z}_v^{(t)}, \tag{2.7}$$

where Γ is a time-decay function. For example, Yao et al. [49] design the model to learn static embeddings $\mathbf{z}_v^{(0)}, \mathbf{z}_v^{(1)}, \ldots, \mathbf{z}_v^{(T-1)}$ based on the node common neighbors and then apply aggregation described above with the time decay set to $\Gamma(T,t) := e^{-\sigma(T-t)}$.

Zhu et al. [50] use the similar idea and formulate the task of link prediction as finding the graph G_T based on aggregation of the d-dimensional latent node representations $\mathbf{Z}^{(0)}, \mathbf{Z}^{(1)}, \ldots, \mathbf{Z}^{(T-1)} \in \mathbb{R}^{|V| \times d}$ at each timestamp:

$$\mathbf{A}_T := \Phi(\mathbf{Z}^{(0)}, \mathbf{Z}^{(1)}, \ldots, \mathbf{Z}^{(T-1)}), \tag{2.8}$$

where $\Phi : \mathbb{R}^{(|V| \times d)^T} \to \mathbb{R}^{|V| \times |V|}$ is some predictor function and $\mathbf{A}_T$ is an adjacency matrix of G_T.

More specifically, node similarity measures $S^{(0)}, S^{(1)}, \ldots, S^{(T-1)}$ can be constructed for each snapshot and then processed with some model designed for time-series data. For example, to predict edges Güneş et al. [51] firstly construct simple similarity measures and then use them as an input for ARIMA — the model designed for time-series forecasting.

2.3.1.3 Explicit temporal smoothing

A number of shallow embedding techniques designed for dynamic networks aim to capture evolution of graph snapshots by preserving temporal smoothness (see Section 2.2.3). In this case, a model learns node embeddings for each snapshot minimizing the additional term

$$\sum_{v \in \mathcal{V}} \sum_{t=0}^{T-2} l(\mathbf{z}_v^{(t)}, \mathbf{z}_v^{(t+1)}), \tag{2.9}$$

where $l : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is any reasonable measure of discrepancy, typically based on the standard inner product.

For example, having snapshots $G^{(0)}, G^{(1)}, \ldots, G^{(T-1)}$ and corresponding adjacency matrices $\mathbf{A}^{(0)}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(T-1)}$ Zhu et al. [50] define the following loss function:

$$\mathcal{L}(\mathbf{Z}^{(0)}, \mathbf{Z}^{(1)}, \ldots, \mathbf{Z}^{(T-1)}) := \sum_{u,v \in \mathcal{V}} \sum_{t=0}^{T-1} \|\mathbf{A}^{(t)}[u,v] - \langle \mathbf{z}_u^{(t)}, \mathbf{z}_v^{(t)} \rangle\|_2^2 + \\ + \lambda \left(\sum_{v \in \mathcal{V}} \sum_{t=0}^{T-2} \left(1 - \langle \mathbf{z}_v^{(t)}, \mathbf{z}_v^{(t+1)} \rangle\right) \right), \tag{2.10}$$

where $\lambda \in \mathbb{R}_0^+$ is a hyperparameter. It can be observed that the proposed model simply optimizes the adjacency matrix factorization (see Section 1.3.2) for each snapshot and aims to preserve a temporal smoothness across snapshots.

TemporalNode2vec [52] follows the same idea but applies node2vec to each snapshot. As mentioned in the Section 1.3.3, random walks approaches can be equivalently defined as a matrix factorization. In this case, it can be achieved with Positive Pointwise Mutual Information matrix $\mathbf{PPMI} \in \mathbb{R}^{|V| \times |V|}$:

$$\mathbf{PPMI}[i,j] := max\left(0, log\left(\theta \cdot \frac{|i,j|_t^w \cdot |\mathcal{V}|}{|i|_t \cdot |j|_t}\right)\right) \tag{2.11}$$

for all $i \neq j \in \mathcal{V}$. Here $|i,j|_t^w$ is the number of times i and j co-appear in the set of node2vec walks of $G^{(t)}$ within a window of size w, $|v|_t$ is the number of occurrences of $v \in \mathcal{V}$ in the set of walks of $G^{(t)}$, and $\theta \in \mathbb{R}$ is a hyperparameter. Then, the total loss function is defined as follows:

$$\mathcal{L}(\mathbf{Z}^{(0)}, \mathbf{Z}^{(1)}, \ldots, \mathbf{Z}^{(T-1)}) := \sum_{u,v \in \mathcal{V}} \sum_{t=0}^{T-1} \|\mathbf{PPMI}^{(t)}[u,v] - \langle \mathbf{z}_u^{(t)}, \mathbf{z}_v^{(t)} \rangle\|_2^2 + \\ + \lambda_1 \left(\sum_{v \in \mathcal{V}} \sum_{t=0}^{T-2} \left(1 - \langle \mathbf{z}_v^{(t)} \mathbf{z}_v^{(t+1)} \rangle\right) \right) + \\ + \lambda_2 \left(\sum_{v \in \mathcal{V}} \sum_{t=0}^{T-1} \|\mathbf{z}_v^{(t)}\|_2^2 \right), \tag{2.12}$$

where $\lambda_1, \lambda_2 \in \mathbb{R}_0^+$ are trainable hyperparameters. Ferreira et al. [53] define the identical model inspired by DynamicWord2vec [54] — the model designed to analyze evolution of word semantics.

2.3.1.4 Tensor factorization

A sequence of adjacency matrices $\mathbf{A}^{(0)}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(T-1)}$ corresponding to snapshots $G^{(0)}, G^{(1)}, \ldots, G^{(T-1)}$, can be stacked into a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{|V| \times |V| \times T}$ of

order 3:

$$\boldsymbol{\mathcal{A}}[i, j, t] := \mathbf{A}^{(t)}[i, j]. \tag{2.13}$$

Then, inspired by adjacency matrix decomposion approaches (see Section 1.3.2), tensor decomposion appraches for dynamic graphs emerged. They are based on the idea to find the rank $d \in \mathbb{N}$ tensor decomposition of $\boldsymbol{\mathcal{A}}$ given as

$$\boldsymbol{\mathcal{A}} \approx \sum_{k=0}^{d-1} \lambda_k \mathbf{a}_k \otimes \mathbf{b}_k \otimes \mathbf{c}_k, \tag{2.14}$$

where $\lambda_k \in \mathbb{R}_+$ and $\mathbf{a}_k, \mathbf{b}_k \in \mathbb{R}^{|V|}$, $\mathbf{c}_k \in \mathbb{R}^T$ for all $k \in \{0, 1, \dots d-1\}$. Here, term $\lambda_k \mathbf{a}_k \otimes \mathbf{b}_k \otimes \mathbf{c}_k$ is defined[22] as a tensor from $\mathbb{R}^{|V| \times |V| \times T}$:

$$(\lambda_k \mathbf{a}_k \otimes \mathbf{b}_k \otimes \mathbf{c}_k)[i, j, t] := \lambda_k \mathbf{a}_k[i] \cdot \mathbf{b}_k[j] \cdot \mathbf{c}_k[t]. \tag{2.15}$$

Then, the obtained vectors can be used to construct d-dimensional node or edge embeddings. For example, Dunlavy et al. [55] utilize $\mathbf{c}_k$ vectors for link prediction. There exist other techniques to factorize the given tensor [56], however, Barros et al. [43] state that exactly the described type of decomposition is typical for dynamic graphs since it learns both topological and temporal information with computational efficiency.

2.3.2 Continuous-time approaches

Contrary to discrete-time approaches discussed in the previous section, continuous-time approaches are designed to directly process a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$. As mentioned in Section 2.2.1, such techniques are far not as developed as the discrete-time ones. In this section, I describe some of the existent continuous-time approaches relevant for the problem defined in Chapter 4.

2.3.2.1 Temporal neighborhood sampling

Nguyen et al. [57] propose a model named Continuous-Time Dynamic Network Embeddings (CTDNE). The idea behind this approach is to generate *temporal walks* on a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$.

Definition 2.3.1. A *temporal walk* of length $n \in \mathbb{N}_+$ on a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ starting from the node $v_1 \in \mathcal{V}$ is a tuple $(v_1, \dots, v_n) \in \mathcal{V}^n$ if there exist $t_1 \leqslant t_2 \leqslant \cdots \leqslant t_{n-1} \in \mathcal{T}$ such that

$$(n > 1) \implies (\forall i \in \{1, \dots, n-1\})((v_i, v_{i+1}, t_i) \in \mathcal{E}).$$

[22] Operator $\otimes$ denotes the *outer product*. In order not to deepen into linear algebra, here it is simply defined as a special syntax.

In the first step, CTDNE samples $w \in \mathbb{N}_+$ walk starting edges $e_1, e_2, \dots e_w \in \mathcal{E}$ according to the defined distribution. Nguyen et al. propose three different distributions to sample a random edge $(u, v, t) \in \mathcal{E}$:

1. Uniform. In this case each edge has the same probability of being selected:

$$p((u, v, t)) := \frac{1}{|\mathcal{E}|}. \tag{2.16}$$

2. Exponential. In this case random walks are biased to start from the more recent edges:

$$p((u, v, t)) := \frac{e^{t - t_{min}}}{\sum\limits_{(u_1, v_1, t_1) \in \mathcal{E}} e^{t_1 - t_{min}}}. \tag{2.17}$$

   Here $t_{min} := \min(\mathcal{T})$.

3. Linear. Consider a function $\eta : \mathcal{T} \to \mathbb{N}_+$ mapping each timestamp to its position in the linearly ordered set $\mathcal{T}$. For example, $\eta(t_{min}) = 1$ and $\eta(t_{max}) = |\mathcal{T}|$. Then, linear distribution means the following definition of sampling probability:

$$p((u, v, t)) := \frac{\eta(t)}{\sum\limits_{(u_1, v_1, t_1) \in \mathcal{E}} \eta(t_1)} \tag{2.18}$$

   Similarly, as in the case of exponential distribution, more recent edges are considered more relevant but at different time granularity.

Of course, this list is not exhaustive and one can define any distribution based on an arbitrary time-decay function or any other strategy. After obtaining the edges $e_1, e_2, \dots e_w$, CTDNE proceeds to the generation of w random walks of the fixed length $l \in \mathbb{N}_+$. A first node v_1 of each walk is determined by the corresponding starting edge. Then, the kth node v_k of each walk is sampled from the *temporal neighborhood* $\mathcal{N}_t(v_{k-1})$ of the previous node for every $k \in \{2, \dots, l\}$, where $t \in \mathcal{T}$ is fixed. For the purpose of sampling neighbors, Nguyen et al. propose distributions similar to the ones designed for starting edge selection. The obtained walks are then processed with skip-gram architecture [17], following the approaches discussed in the Section 1.3.3.

Definition 2.3.2. Consider a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$. Then, the set $\mathcal{N}_t(u) := \{(v, t_n) \mid (u, v, t_n) \in \mathcal{E} \wedge\ t_n > t\}$ defines a *direct temporal neighborhood* of a node $u \in V$ at time $t \in \mathcal{T}$.

A temporal random walk may be interpreted as a feasible route for a piece of information through the dynamic network. For example, considering a network representing interactions between people, random walks may represent a virus propagation. Node embeddings that capture this information may accurately describe dynamic graphs in many real-world applications.

Torricelli et al. [36] propose a different continuous-time approach. To embed the edges of a dynamic graph they firstly convert $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ to a *directed* (see Definition 2.3.3) graph $G' = (\mathcal{E}, E')$, where $E' \subset \mathcal{E} \times \mathcal{E}$ is defined as:

$$E' := \Big\{(u_1, v_1, t_1), (u_2, v_2, t_2) \in \mathcal{E} \times \mathcal{E} \Big| \{u_1, v_1\} \cap \{u_2, v_2\} \neq \emptyset \wedge |t_2 - t_1| < \Delta t\Big\} \tag{2.19}$$

with $\Delta t \in \mathbb{T}$. Moreover, edges E' are associated with two types of weights $w_{path} : E' \to [0, 1]$ and $w_{co} : E' \to \mathbb{N}$:

$$w_{path}((u_1, v_1, t_1), (u_2, v_2, t_2)) := \frac{1}{1 + |t_2 - t_1|}, \tag{2.20}$$

$$w_{co}((u_1, v_1, t_1), (u_2, v_2, t_2)) := \Big|\Big\{(u_a, v_a, t_a), (u_b, v_b, t_b) \in E' \big| u_a = u_1 \wedge v_a = v_1 \wedge u_b = u_2 \wedge v_b = v_2\Big\}\Big|. \tag{2.21}$$

Here, w_{path} associates pairs of events with their time-decay-weighted temporal distances and w_{co} represents the number of co-occurrences of event pairs on the same nodes. The obtained directed acyclic[23] graph is called a *weighted event graph* and represents the relation of causality on events. To obtain the final graph authors also remove some specific edges: for a given event if it has multiple future adjacent events with the same pair of nodes, only the earliest one is considered.

Definition 2.3.3. A directed *graph* is an ordered pair $G = (V, E)$, where V is a non-empty finite set of *nodes* or *vertices* and $E \subseteq \{\, (u, v) \in V \times V \mid u \neq v \,\}$ is a set of *edges* or *links*.

Definition 2.3.4. Consider a directed graph $G = (V, E)$. Then, the set $\mathcal{N}(v) := \{u \in V \mid (u, v) \in E \vee (v, u) \in E\}$ defines a *direct neighborhood* of a node $v \in V$.

Then, Torricelli et al. to obtain the embeddings of events $\mathcal{E}$, sample their *direct neighbors* in G' with the probability $p(e_l|e_k)$ of choosing the event $e_l \in \mathcal{N}(e_k)$ as a neighbor of $e_k \in E'$ defined as:

$$p(e_l|e_k) := \lambda F(w_{path}(e_k, e_l)) + (1 - \lambda)F(w_{co}(e_k, e_l)), \tag{2.22}$$

[23] `https://en.wikipedia.org/wiki/Directed_acyclic_graph`

where $\lambda \in [0, 1]$ is a hyperparameter and

$$F(w(e_k, e_l)) := \frac{w(e_k, e_l)}{\sum_{n \in \mathcal{N}(e_k)} w(e_k, e_n)} \tag{2.23}$$

normalizes the weights. The generated neighborhoods are then used as an input for the skip-gram architecture [17]. The obtained embeddings implicitly preserve temporal smoothness and aim to capture the relation of causality between events. For the illustration of the possible embeddings see Figure 2.2. The described model is referred to as weg2vec [36].

2.4 Dynamic graph neural networks

Deep learning has proved to be a powerful tool for problems in different domains. In the Section 1.4 I described its embodiment on graphs — Graph neural networks. In turn, Recurrent neural networks [22, Chapter 10] are designed to solve tasks on sequences, specifically time series [58]. Modern deep learning approaches designed for dynamic graphs try to combine these paradigms. In this section, I briefly discuss such approaches — Dynamic graph neural networks (DGNNs). I mainly rely on the survey by Skarding et al. [42] (2020) covering the corresponding problematics.

2.4.1 Discrete-time approaches

Skarding et al. [42] distinguish two types of discrete-time Dynamic graph neural networks: *stacked* DGNNs and *integrated* DGNNs. In the following subsections I introduce the representatives of both classes considering a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ and its snapshots $G^{(0)}, G^{(1)}, \ldots, G^{(T-1)}$.

2.4.1.1 Stacked dynamic graph neural networks

A straightforward way to utilize the power of rraph neural networks on dynamic graphs is to apply GNN to each snapshot and then process the obtained embeddings with some RNN architecture. It can be formalized as follows:

$$\mathbf{Z}^{(t)} := \mathrm{GNN}(G^{(t)}) \tag{2.24}$$

$$\mathbf{S}^{(t)} := \mathrm{RNN}(\mathbf{S}^{(t-1)}, \mathbf{Z}^{(t)}). \tag{2.25}$$

Here, GNN represents an arbitrary Graph neural network architecture that produces node embeddings $\mathbf{z}_0^{(t)}, \mathbf{z}_1^{(t)}, \ldots, \mathbf{z}_{|\mathcal{V}|-1}^{(t)} \in \mathbb{R}^{d_1}$ of graph snapshot at time t given as a matrix $\mathbf{Z}^{(t)} \in \mathbb{R}^{|V| \times d_1}$. Then, RNN is any Recurrent neural network with $\mathbf{S}^{(t)} \in \mathbb{R}^{|V| \times d_2}$ representing a hidden state of RNN at time t, $\mathbf{S}^{(-1)}$ may be set to zero matirx. Numbers $d_1, d_2 \in \mathbb{N}_+$ are the dimensionalities of the RNN and GNN latent spaces. The final hidden states of

the model given by matrix $\mathbf{S}^{(T-1)}$ can then be used as global node embeddings $\mathbf{z}_0, \mathbf{z}_1, \ldots, \mathbf{z}_{|\mathcal{V}|-1}$. Note that parameters of GNN are shared across the snapshots. Similarly, a single RNN is used to handle each node.

Seo et al. [59] propose a Graph Convolutional Recurrent Network (GCRN), which falls under the described framework. They use a GCN-like (see Section 1.4.2) Graph neural network and LSTM-based [27] Recurrent neural network. Similar approaches include RgCNN [7] and DyGGNN [2] which can be distinguished by the used graph and recurrent components. Manessi et al. [60] propose slightly different architectures Waterfall Dynamic-GCN (WD-GCN) and Concatenated Dynamic-GCN (CD-GCN) which are based on GCN and LSTM, however, a different LSTM for each node is used.

2.4.1.2 Integrated dynamic graph neural networks

Another way to combine GNNs with RNNs is to directly integrate recurrence into graph aggregation layers by introducing a recurrent dependence of GNN parameters while processing consecutive snapshots. For example, considering an L-layer GNN, with its layers denoted as GNN_l, and any architecture of RNN, this idea can be realized as follows:

$$\mathbf{W}^{(l)(t)} := \text{RNN}(\mathbf{H}^{(l)(t)}, \mathbf{W}^{(l)(t-1)}). \tag{2.26}$$

$$\mathbf{H}^{(l+1)(t)} := \text{GNN}_l(G^{(t)}, \mathbf{H}^{(l)(t)}, \mathbf{W}^{(l)(t)}). \tag{2.27}$$

Here, $\mathbf{W}^{(t)}$ denotes GNN parameters (weights) at time t and $\mathbf{W}^{(t-1)}$ should be initialized explicitly. Next, $\mathbf{H}^{(l)(t)} \in \mathbb{R}^{|V| \times d}$ represents hidden node embeddings at the kth layer of GNN at time t. Note that the parameters of GNN layers are shared. The defined layer can then be used as the ones designed for static graphs (see Section 1.4.1).

Pareja et al. [61] propose such an architecture referred to as EvolveGCN-H. They use GCN and GRU [62] as a graph and recurrent modules. They also propose a slightly different version of this architecture named EvolveGCN-O. In this case, the LSTM is used and it does only process the GNN weights regardless of the hidden embeddings. Chen et al. [63] propose GC-LSTM which is also defined as an integrated layer.

2.4.2 Continuous-time approaches

Skarding et al. [42] (2020) state that there exist only two approaches to continuous-time dynamic graph representation learning: RNN-bashed methods and the techniques based on temporal point processes. The former methods are task-driven. They include *streaming* graph neural networks [64] designed for strictly evolving graphs (i. e. edges are only added) and JODIE [3] designed for bipartite graphs. I describe the JODIE architecture in Section 2.5.

The approaches based on temporal point processes include DyREP[65] which I do not discuss in this work.

2.5 Machine learning on dynamic bipartite graphs

In this section, I want to discuss some approaches designed specifically for *dynamic bipartite graphs*.

Definition 2.5.1. A dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ is *bipartite* if two disjoint sets U and I exist such that $\mathcal{V} = U \cup I$ and $\mathcal{E} \subseteq U \times I \times \mathcal{T}$. U and I are said to be the *partitions* of a graph $\mathcal{G}$.

Snapshots of a dynamic network can be conveniently defined as bipartite graphs. Having a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ with partitions U and V and its snapshots $G^{(0)}, G^{(1)}, \ldots, G^{(T-1)}$ for some $T \in \mathbb{N}$, many models discussed in this chapter can be used to utilize the topological-awareness by slight modifications. For example, Rafailidis and Nanopoulos [66] and Fang et al. [67] find user and item embeddings factorizing the tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{|U| \times |V| \times T}$ constructed in the same way as discussed in Section 2.3.1.4, in this case utilizing bipartite adjacency matrices.

Nevertheless, there also exists a variety of machine learning models designed specifically for dynamic bipartite networks. These models are typically ad hoc for specific tasks. For example, Li et al. [33] for the time-aware product recommendation firstly construct a product-product graph based on the number of times products are purchased by the same users in terms of separate snapshots. After that, they apply LINE algorithm (see Section 1.3.3) on the constructed graph in order to obtain product embeddings. Users are then placed in the same latent space with the simple calculations based on the time-decayed embeddings of products they have purchased. Guo et al. [34] for the session-based recommendations propose the TA-GNN architecture. The idea of this approach is to construct two separate graphs for users and items and then apply GNN and RNN technologies to produce separate embeddings and then combine them for the final downstream task. In more detail, I want to discuss the JODIE [3] model, highlighted by Skarding et al. [42] as one of the few continuous-time dynamic graph neural networks.

JODIE

The idea of JODIE[24] is to use two mutually-recursive recurrent neural networks in order to capture the user-item interactions. Contrary to the other discussed models, JODIE tries to learns user and item embeddings for each

[24] JOint Dynamic user-Item Embeddings

user $u \in U$ and item $i \in I$ as continuous functions of time $u : \mathbb{T} \to \mathbb{R}^d$, $u : \mathbb{T} \to \mathbb{R}^d$, where $d \in \mathbb{R}$. I remind that $\mathbb{T}$ is a temporal domain of $\mathcal{G}$. The idea of this model is to capture the temporal smoothness of embeddings by mutually training the following recurrent neural networks for each consecutive interaction between u and i:

$$u(t) = \sigma\left(\mathbf{W}_1^u \cdot u(t_u^-) + \mathbf{W}_2^u \cdot i(t_u^-) + \mathbf{W}_3^u \cdot f + \mathbf{w}_4^u \cdot \Delta_u\right), \tag{2.28}$$

$$i(t) = \sigma\left(\mathbf{W}_1^i \cdot i(t_i^-) + \mathbf{W}_2^i \cdot u(t_i^-) + \mathbf{W}_3^i \cdot f + \mathbf{w}_4^i \cdot \Delta_i\right). \tag{2.29}$$

Here, $\mathbf{W}_1^u, \mathbf{W}_2^u, \mathbf{W}_1^i, \mathbf{W}_2^i \in \mathbb{R}^{d \times d}, \mathbf{W}_3^u, \mathbf{W}_3^i \in \mathbb{R}^{d \times d_f}, \mathbf{w}_4^u, \mathbf{w}_4^i \in \mathbb{R}^d$ are trainable parameters. t_u^- denotes the previous time of u's/i's interaction with any item/user. Next, $f \in \mathbb{R}^{d_f}$ is an optional interaction feature vector of the currently processed interaction and $\Delta_u := t - t_u^-$, $\Delta_i := t - t_i^-$. Then, he given recurrent neural networks are trained to preserve temporal smoothness by trying to predict embeddings of a next processed interaction.

Chapter 3

Time-series GNN

Contrary to the nature of time, the prevailing majority of state-of-the-art machine learning techniques for dynamic graphs consider time as a discrete variable [42]. However, based on the conducted study, represented by Chapters 1 and 2, I find it natural to define a continuous-time dynamic graph neural network in the way I propose in this chapter. First of all, I introduce a general framework which I refer to as *Time-series GNN*, and then I describe a simple example of its possible instance. For the explanation I use the notation built in Chapter 1, particularly in Section 1.4.1.

3.1 Time-series GNN framework

Consideration of a general graph in Section 1.4.1 — GNN framework, led to the conclusion that the graph message passing should be defined with a multiset aggregator function. It was motivated by two reasons: (*i*) node neighborhoods are not naturally ordered, (*ii*) neighborhoods are variable in size. However, considering a dynamic graph, edge timestamps introduce an order on the neighborhoods: a node receives messages from the neighbors in the certain order. This fact allows treating node neighborhoods as sequences. Then, instead of a parameterized multiset aggregator function any sequence-based deep-learning architecture such as a simple recurrent neural network [22, Chapter 10], Gated recurrent unit, [62], Long short-term memory [27], 1D convolutional neural network [68] or Transformer [69] can be used. More specifically, node neighborhoods define time series. Thus, any deep learning technique for time series processing [58] may be utilized. Conclusively, it means that for a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$, a GNN layer (see Equation (1.21)) can be generically defined as:

$$\mathbf{h}_v^{(k+1)} := \text{UPDATE}\Big(\mathbf{h}_v^{(k)}, \text{AGGTS}\Big(\big\{(t, \mathbf{h}_u^{(k)}) \mid (t,u) \in \mathcal{N}(v)\big\}\Big)\Big), \tag{3.1}$$

where UPDATE : $\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ and AGGTS[25] : $2^{\mathcal{T}\times\{\mathbf{h}_u^{(k)}|u\in\mathcal{V}\}} \to \mathbb{R}^n$ are any (parameterized) differentiable functions. Next, $\mathcal{N}(v)$ is a *neighborhood* (or alternatively a set of *successors* or *predecessors*) of v. Note that here AGGTS is defined to be the most general possible aggregator and the notion of time series is slightly more general than the usual one. For the illustration of the described framework see Figure 3.1.

Definition 3.1.1. Consider a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$. Then, the set $\mathcal{N}_S(u) := \{(t_n, v) \mid (u, v, t_n) \in \mathcal{E}\}$ is a set of *successors* of a node $u \in \mathcal{V}$. The set $\mathcal{N}_P(u) := \{(t_n, v) \mid (v, u, t_n) \in \mathcal{E}\}$ is similarly a set of its *predecessors*. Finally, the set $\mathcal{N}(u) := \mathcal{N}_P(u) \cup \mathcal{N}_S(u)$ is a *direct neighborhood* of u.

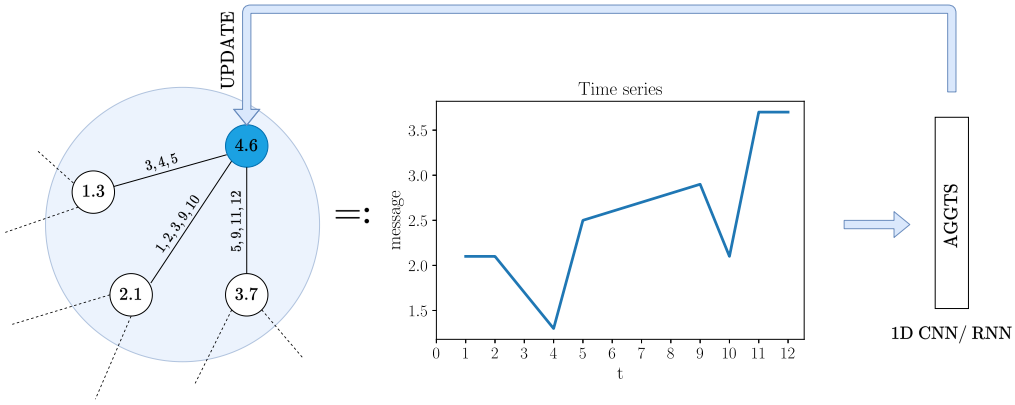


Figure 3.1: Illustration of the Time-series GNN. The figure depicts an iteration of the proposed message passing on a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ to update the hidden state $\mathbf{h}_u^{(0)} = 4.6$ of a node $u \in \mathcal{V}$. Next, figure illustrates the *direct neighborhood* of u: $\mathcal{N}(u) = \{(3, u_1), (4, u_1), (5, u_1), (1, u_2), (2, u_2), (3, u_2), (9, u_2), (10, u_2), (5, u_3), (9, u_3), (11, u_3), (12, u_3)\}$, where $u_1, u_2, u_3 \in \mathcal{V}$ and $\mathbf{h}_{u_1}^{(0)} = 1.3, \mathbf{h}_{u_2}^{(0)} = 2.1, \mathbf{h}_{u_3}^{(0)} = 3.7$. The idea of Time-series GNN is to treat the considered neighborhood as time series $\{(3, 1.3), (4, 1.3), (5, 1.3), (1, 2.1), (2, 2.1), (3, 2.1), (9, 2.1), (10, 2.1), (5, 3.7), (9, 3.7), (11, 3.7), (12, 3.7)\}$. The curve in the middle of the figure demonstrates the given time series, preprocessed such that the messages at the same timepoints are aggregated with their mean value. Note that this aggregation is not necessary and is done only for the illustration. The obtained time series is used as an input for the AGGTS function. Finally, the output of AGGTS is used to update the $\mathbf{h}_u^{(0)}$ with the UPDATE function.

[25] AGGregate Time Series

3.2 Simple RNN-based Time-series GNN

In this subsection I want to give a simple straightforward example of a possible instance of the discussed framework. Consider a node $v \in \mathcal{V}$ of a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$. Then its *neighborhood* (or alternatively a set of *successors* or *predecessors*) $\mathcal{N}(v)$ defines a sequence of nodes $v_0, v_1, \ldots, v_{|\mathcal{N}(v)|-1}$ such that for the corresponding timestamps holds $t_0 \leqslant t_1, \leqslant, \ldots, \leqslant t_{|\mathcal{N}(v)|-1}$. Then, the sequence of corresponding node hidden states $\mathbf{h}_0^{(k)}, \mathbf{h}_1^{(k)}, \ldots, \mathbf{h}_{|\mathcal{N}(v)|-1}^{(k)}$ can be used as an input for the RNN architecture to update the hidden state $\mathbf{h}_v^{(k)}$ of v:

$$\mathbf{s}_i := \tanh\left(\mathbf{W}_1 \mathbf{h}_i^{(k)} + \mathbf{W}_2 \mathbf{s}_{i-1} + \mathbf{b}\right), \tag{3.2}$$

Here, $\mathbf{h}_i^{(k)} \in \mathbb{R}^{d_1}$ is the ith node hidden state (or the message) and $\mathbf{s}_i \in \mathbb{R}^{d_2}$ is a hidden state of the recurrent neural network while processing the ith message; $\mathbf{s}_{-1}$ should be initialized. Next, tanh is a hyperbolic tangent[26] non-linearity function, $\mathbf{W}_1 \in \mathbb{R}^{d_2 \times d_1}$, $\mathbf{W}_1 \in \mathbb{R}^{d_2 \times d_2}$ and $\mathbf{b} \in \mathbb{R}^{d_2}$ are trainable parameters. Here, $d_1, d_2 \in \mathbb{N}_+$ are the dimensionalities of the GNN and RNN latent spaces respectively. Then, the vector $\mathbf{s}_{|\mathcal{N}(v)|-1}$ can be used as an updated hidden state of v in GNN:

$$\mathbf{h}_v^{(k+1)} := \mathbf{s}_{|\mathcal{N}(v)|-1}. \tag{3.3}$$

[26] `https://en.wikipedia.org/wiki/Hyperbolic_functions#Hyperbolic_tangent`

Chapter 4

Problem definition

Nowadays, the Internet becomes more and more powerful and uncontrolled tool. Every day fraudsters create websites to spread malware or steal private information. For example, today, visiting a seemingly innocent domain like `covid19doctors.com` may be highly dangerous. Impressively, from January 1st 2020 through March 23rd 2020 at least 62000 of such COVID-19-related malicious domains were spotted[27]. Fraudsters utilize such names to attract traffic and then involve unaware users in their malicious activities. Typical examples of such lawbreaking activities include *phishing*[28] and turning the attracted users' machines into bots, which then silently listen to attacker's commands [70].

An in-time detection of such domains may prevent serious consequences. Researchers have been developing different methods to reveal suspicious domains among the millions of benign ones. Classic so-called *knowledge-base* approaches [71] are based on human-expert insights. For example, Sato et al. (2012) [72] notice that some of the malicious domains tend to co-occur in DNS traffic data. With this assumption, they propose a method to extend the known blacklist of domains. However, the majority of modern approaches, developed to detect malicious domains, are data-driven with machine-learning algorithms at their core [71]. These approaches include techniques based on the feature extraction from domain names and their behavior. For example, Chiba et al. [73] exploit the so-called temporal variational patterns of domains in DNS logs, which include information about how and when a domain name has been listed in legitimate/popular and/or malicious domain name lists. Other popular techniques for detecting "black" domains are based on graph machine learning technologies [74].

[27] `https://www.domaintools.com/resources/blog/free-covid-19-threat-list-domain-risk-assessments-for-coronavirus-threats`

[28] `https://en.wikipedia.org/wiki/Phishing`

Arguably, the key challenge of using machine learning in the domain classification task is the nature of labeled data. Having a list of domains, we would possibly like to associate each of them with a binary label indicating that the domain is malicious or benign respectively. Then, we can apply some supervised machine-learning models to learn the patterns necessary for distinguishing between the two types of domains. Nevertheless, such an approach is infeasible in real-world applications.

First of all, having a domain, what is the proper way to decide on its maliciousness? A common approach is to use reputable blacklists as the source of ground truth. Despite the fact that blacklists such as PhishTank[29] or Web of Trust[30] are generally considered as a reliable source, the obtained labels cannot be completely truthful. Some works demonstrate a relatively high rate of false-positive data in some recognized blacklists [71].

Using techniques such as voting based on several blacklists [75], one is still able to assign relatively reliable positive labels to malicious domains. However, assigning negative labels to benign ones is a far more challenging problem. One cannot easily conclude that a given domain is malicious. In 2016 over 296 million second-level domains[31] existed [73]. Obviously, this number keeps growing rapidly and regular scanning of such an amount of domains is computationally infeasible. Some approaches use top-k domains from the services like Alexa Top Sites[32] as ground truth for domain benignness [71]. However, this service scores domains mainly based on their popularity and statistics, but not on cybersecurity aspects, and cannot be generally considered a reliable source for the described purposes. Stevanovic et al. [76] show a relatively high false-positive rate of this system. In general, it is not possible to obtain the labels for benign domains. It is only possible to use some heuristics based on combinations of domain filtration and ranking services usage [71], but in the best case, it leads to obtaining a small number of ground truth data.

In such circumstances, the usage of machine learning techniques becomes challenging. The facts described above mean dealing with noisy data and a severe class imbalance between labeled and unlabeled data. Such conditions require special approaches for domain classification. Some of the typical approaches include one-class classification [77] or a modified binary classification with certain assumptions and corresponding evaluation metrics.

Described problematics and challenges outline the background for a formal

[29] `https://www.phishtank.com/`
[30] `https://www.mywot.com/`
[31] `https://en.wikipedia.org/wiki/Second-level_domain`
[32] `https://aws.amazon.com/alexa-top-sites/`

definition of the task studied in this thesis.

4.1 Cisco Cognitive maliciousness classification problem

In this section, I introduce the *Cisco Cognitive Intelligence maliciousness classification problem* — the task of malicious domains detection in anonymized real-world data provided by Cisco, Cognitive Intelligence group.

The given dataset is composed as an anonymized sample of the Cisco Cognitive customers traffic. The dataset contains all traffic of users that at least once visited a known high risk site. It contains 1 924 728 entries, each represented with 6 features:

user_id	Anonymized user IP address (`123.45.678.91`)
url_id	Anonymized URL name (`https://en.wikipedia.org/wiki/URL`)
hostname_id	Anonymized hostname (`en.wikipedia.org`)
sld_id	Anonymized second-level domain name (`wikipedia.org`)
timestamp	Timestamp represented as a non-negative integer (`23:59`)
label	Label. 1(labeled) if **url_id** is malicious and 0(unlabeled) otherwise (`0`)

Each entry represents a visit to a website by some user at a certain timepoint. Examples in the brackets give an intuition of the possible original record. There are in total of 179 malicious URLs and, consequently, 54 corresponding malicious second-level domains, which is in high contrast with the total number of URLs and domains, which are equal to 325 915 and 28 685 respectively. The more detailed statistics are given in Table 4.1. In terms of this thesis, I focus on relations between users and second-level domains. Moreover, I remove 80% of unlabeled second-level domains due to the limited computational capacity. The final statistics of the preprocessed dataset can be found in Table 4.2

Let us now proceed to the formal definition of the machine learning task studied in this thesis. Many machine learning problems can be conveniently defined by three components: the *experience* that machine learning models are allowed to have during the learning process, the *task* they aim to solve based on the received experience, and the *performance measure* designed to

	unique values
user_id	83
url_id	325915
hostname_id	28685
sld_id	16931
timestamp	1433012

(a) All interactions (1 924 728)

	unique values
user_id	79
url_id	179
hostname_id	56
sld_id	54
timestamp	6923

(b) Malicious interactions (7 051)

Table 4.1: Quantitative characteristics of the Cisco Cognitive dataset. The table shows the numbers of unique values in **(a)** the whole dataset **(b)** in interactions with malicious URLs (malicious interactions).

	unique values
user_id	83
sld_id	3440
timestamp	464088

(a) All interactions (533 474)

	unique values
user_id	79
sld_id	54
timestamp	6923

(b) Malicious interactions (7 051)

Table 4.2: Quantitative characteristics of the preprocessed Cisco Cognitive dataset. The table shows numbers of unique values in **(a)** the whole dataset **(b)** in interactions with malicious URLs (malicious interactions).

evaluate the models' efficiency [22, Chapter 5]. In the following sections, I describe each of these parts which holistically define the problem.

4.1.1 Experience

The described above dataset defines a graph structure. Considering unique **user_id** and **sld_id** values, we can construct a set of users $U = \{u_0, u_1, \ldots, u_{m-1}\}$ and a set of second-level domains $D = \{d_0, d_1, \ldots, d_{n-1}\}$ for some $m, n \in \mathbb{N}_+$. Then, relations between users and domains can be modelled as a dynamic bipartite graph (see Definition 2.5.1) $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ with partitions U and D, where $(u, d, t) \in \mathcal{E}$ represents an interaction between user u and domain d at time t, given by the corresponding dataset entry. Additionally, unique pairs of **sld_id** and **label** values define a mapping

$$\varphi : D \to \{0, 1\} \tag{4.1}$$

that associates domains with their labels. For a domain $d \in D$, $\varphi(v) = 1$ means that v is malicious, while $\varphi(v) = 0$ means that we do not posses any information about the maliciousness or benignness of v. Figure 4.1a illustrates the described dynamic graph.

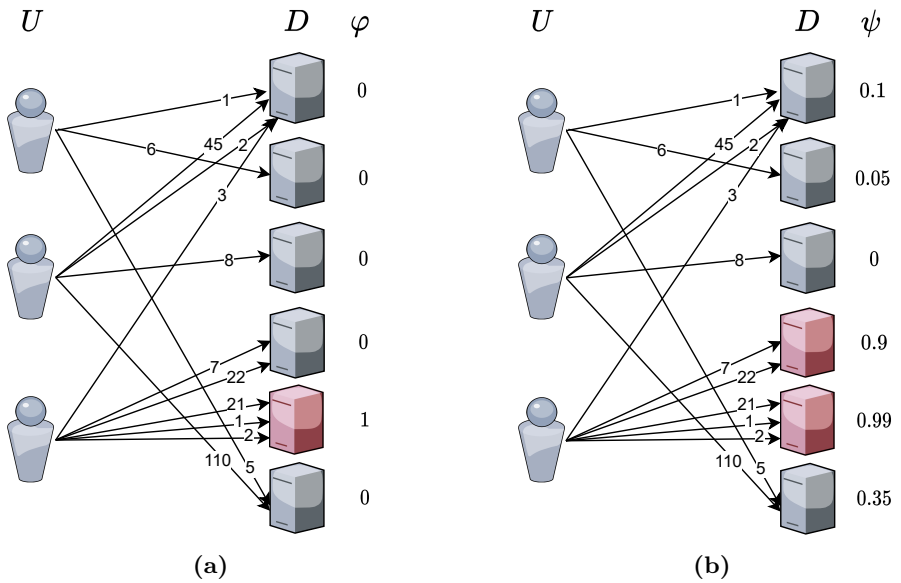


Figure 4.1: An illustration of the Cisco Cognitive maliciousness classification problem: **(a)** initial dynamic graph, **(b)** inference.

During the learning process, a machine learning model is allowed to consider the graph $\mathcal{G}$ and the φ values of the *training* set of labeled domains $\mathcal{D} \subset \{d \in D \mid \varphi(d) = 1\}$ and all the *unlabeled* ones $\mathcal{U} := \{d \in D \mid \varphi(d) = 0\}$. Such a learning style is usually called *learning with positive labels*. From the pure graph perspective, this learning may be referred to as *semi-supervised*, since a model captures information about the whole graph, including all the nodes, but considers labels only for some subset of the nodes [4].

4.1.2 Task

For a model, the goal of the learning process is to find a mapping:

$$\psi : D \to [0, 1] \tag{4.2}$$

which associates domains with their *scores*. For a node $d \in D$, its score $\psi(d)$ represents a probability that d is malicious.

Then, in practice, some fixed number of domains with the highest scores can be claimed to be malicious. This task can be seen as an information retrieval problem: having a collection of resources (domains), we aim to obtain the relevant (malicious) ones. I refer to the process of associating domains with their scores as *inference*. For the illustration of the described approach see Figure 4.1b.

4.1.3 Performance measure

In information retrieval tasks, typically used evaluation metrics include *precision at* k, defined as:

$$\mathrm{P@}k := \frac{|\{\text{relevant}\} \cap \{\text{retrieved}\}|}{\{\text{retrieved}\}}. \tag{4.3}$$

Another possible metric is *recall at* k:

$$\mathrm{R@}k := \frac{|\{\text{relevant}\} \cap \{\text{retrieved}\}|}{\{\text{relevant}\}}, \tag{4.4}$$

where {relevant} is a set of relevant resources and {retrieved} is a set of k resources with the highest scores.

In terms of the considered task $\{\text{relevant}\} := \{d \in D \mid \varphi(d) = 1\}$ is a set of malicious domains, and $\{\text{retrieved}\} \subseteq D$ is a set of k domains with the highest scores given by ψ. For the maliciousness classification task studied in this thesis, P@k is used as a performance criterion: having top-k domains with the highest maliciousness probabilities, we are interested in how many of them are indeed labeled as malicious.

In this work, I evaluate the models with P@k for different k values chosen based on the total number of malicious domains. More importantly, with the same metrics, I evaluate the models' generalization capacities. In this case their precision is measured on the *test* set of domains $\mathcal{T} := D \setminus \mathcal{D}$. It means: $\{\text{relevant}\} := \{d \in \mathcal{T} \mid \varphi(d) = 1\}$ and $\{\text{retrieved}\} \subseteq D \setminus \mathcal{D}$ is the set of top k domains excluding the training ones.

Moreover, I additionally measure R@$|\mathcal{T}|$ on the test set to have a piece of dual information about a model's performance. This criterion represents how many of the hidden, during the training, malicious domains are revealed by the inference. Indeed, if all of the test domains are exposed then R@$|\mathcal{T}| = 1$. Regarding the considered task, this evaluation metric is auxiliary and less significant than P@k, since we are mainly interested in the correct top-k prediction and not in the exposure of as many domains as possible.

4.2 Related problems

The defined maliciousness classification problem is related to different domains. First of all, the learning experience (Section 4.1.1) is defined as a dynamic graph. It means that this task is primarily related to the problematics described in Chapters 1 and 2. More specifically, the given graph is bipartite. There exist a range of works on machine learning with dynamic bipartite networks [3, 33, 34, 78, 79, 80], some of them as discussed in Section 2.5. The

prevailing majority of such works consider a graph representing interactions between users and items, typically products, as mentioned in Section 1.5 and followed in Section 2.5.

More specifically, we may treat the given interactions between users and domains as implicit feedback (see Section 1.5.1). Having a bipartite dynamic network $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$, considered in this chapter, with partitions U and I, we can represent it as implicit feedback $\mathbf{R} \in \mathbb{R}^{|U|\times|I|}$ in the following way:

$$\mathbf{R}[u, i] := \big|\{(v_1, v_2, t) \in \mathcal{E} | v_1 = u \wedge v_2 = i\}\big|. \tag{4.5}$$

It makes the maliciousness classification task related to the corresponding problematics [29, 30, 81] as well. Implicit feedback representation may be considered equivalent to the bipartite network representation. However, considering the nature of relations may be favorable. For example, models discussed in Section 1.5.1 rely on some special assumption about user-item connections.

Moreover, the maliciousness classification problem is related to the problematics of imbalanced learning [82, 77, 83, 84], due to the discussed origin of data, and to the information retrieval domain [85], from the perspective of defined task (Section 4.1.2) and performance measure (Section 4.1.3).

Despite the connections of the considered problem with the various fields, I am not aware of other problems that can be reasonably defined in exactly the same way. Moreover, despite the existence of extensive graph dataset collections [86, 87], it is challenging to find bipartite dynamic graph data with corresponding binary labels.

4.2.1 Foreign customer classification problem

	unique values
StockCode	200
CustomerID	4141
InvoiceDate	17021

(a) All purchases (128 334)

	unique values
StockCode	200
CustomerID	415
InvoiceDate	2049

(b) Purchases by foreigners (14 396)

Table 4.3: Quantitative characteristics of the preprocessed e-commerce dataset. The table shows numbers of unique values in **(a)** the whole dataset **(b)** in interactions involving foreign customers. **StockCode** and **CustomerID** represent the user and product identifiers respectively and **InvoiceDate** defines timestamps.

Based on the overview of related tasks I construct a similar e-commerce-related problem based on the open data[33]. The obtained dataset contains

[33] `https://www.kaggle.com/carrie1/ecommerce-data`

all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail. It contains records about the purchases from 37 countries, mainly from the United Kingdom. I preprocess this dataset to define the analogous task of a foreign customer detection. See Table 4.3 for the statistics. This problem serves as an auxiliary task for more objective experiments, however, is still distant from the maliciousness classification problem.

Chapter 5

Experiments

In this chapter, I describe the experiments I conduct to demonstrate the possible solutions of the Cisco Cognitive maliciousness classification problem defined in Chapter 4 and show the influence of leveraging the temporal aspect of data.

5.1 Workflow

In this work, I use Python 3 to implement the experimental framework and machine learning models. Mainly, I rely on the following libraries: NumPy [88] for numerical arrays handling, SciPy [89] for operations with linear algebra objects, pandas [90] for datasets processing, scikit-learn [91] for general machine learning tools, PyTorch [92] and PyTorch Geometric [93] for deep learning models. Also I use Matplotlib [94] for visualizations.

5.2 Experimental framework

In this section, I describe how I set up the experimental framework to evaluate the performance of machine learning models on the considered task . First of all, I construct the training set $\mathcal{D}$ (see Section 4.1.1) with 60 % of randomly chosen labeled domains. The rest of them define the test set $\mathcal{T}$. The following sections cover the training, test and hyperparameter tuning methods. Note that, as described in Chapter 4, the set of unlabeled domains $\mathcal{U}$ is considered by a model during both training and test.

5.2.1 Training

As described in Chapters 1 and 2, there are two leading approaches in machine learning on (dynamic) graphs: shallow embedding and graph neural networks.

These types of techniques require different approaches to use them regarding the considered problem: to obtain the mapping ψ (see Section 5.2.1).

To make an inference with a shallow embedding model, it is firstly trained in an unsupervised manner to produce node embeddings $\mathbf{z}_0, \mathbf{z}_2, \dots \mathbf{z}_{|\mathcal{V}|-1}$. Then the training pairs $\{(\mathbf{z}_d, \varphi(d)) \mid d \in \mathcal{D} \cup \mathcal{U}\}$ are fed to the Random forest classifier. A mapping ψ is then given by the probability predictions of Random forest for each $d \in D$. The choice of Random forest is motivated by its robustness against imbalanced data.

In case of graph neural networks, ψ can be directly defined as

$$\psi(d) := \sigma(f(\mathbf{z}_d)), \tag{5.1}$$

where σ is a sigmoid function and $f : \mathbb{R}^n \to \mathbb{R}$ is a 1-layer feedforward neural network. Here $\mathbf{z}_d \in \mathbb{R}^n$ denotes the embedding of a node d and n is the dimension of the latent space (see Section 1.4.1). Then, such a model can be trained end-to-end in a supervised manner with mean squared error loss $\mathrm{MSE} : 2^{\mathcal{D}\cup\mathcal{U}} \to \mathbb{R}$ with additional weights, defined as:

$$\mathrm{MSE}(\mathcal{B}) := \frac{1}{|\mathcal{B}|} \sum_{d\in\mathcal{B}} w(d)(\varphi(d) - \psi(d))^2, \tag{5.2}$$

where $\mathcal{B} \subseteq \mathcal{D} \cup \mathcal{U}$ is some batch of domains used for training. Here I add the weights $w : D \to \{0, 1\}$ defined as

$$w(d) := \begin{cases} \frac{|\mathcal{D}|}{|\mathcal{U}|} & \text{if } d \in \mathcal{U}, \\ 1 & \text{otherwise} \end{cases} \tag{5.3}$$

to alleviate the class imbalance. After the training, the values of ψ for each domain $d \in D$ are given by a forward pass of GNN.

5.2.2 Test

To test a shallow model, I repeat the training (see Section 5.2.1) with the obtained embeddings 5 times. It means the repetition of the following procedure: (*i*) choose the Random forest's parameters using 2-fold cross-validation (see Section 5.2.3), (*ii*) train the Random forest, (*iii*) make an inference and obtain the metric scores. After that, I calculate the mean value for each metric. In case of graph neural networks, I directly calculate metric scores on the values given by a forward pass.

5.2.3 Hyperparameter tuning

To set a model's hyperparameters for training I use 2-fold cross-validation[34]. It means that firstly the predefined half of training domains $\mathcal{D}$ is used to

[34] `https://en.wikipedia.org/wiki/Cross-validation_(statistics)`

train a model and then it is tested with the rest of $\mathcal{D}$. I find the described setup a trade-off between time complexity and efficiency, having a limited computational capacity.

5.3 Applied models

In this section, I overview the machine learning models applied to the considered problems. The description of the underlying datasets and problem details are discussed in Chapter 4. All the applied models can be classified by their *topological* and *temporal awareness*. See Figure 5.1 for the illustration of the proposed taxonomy.

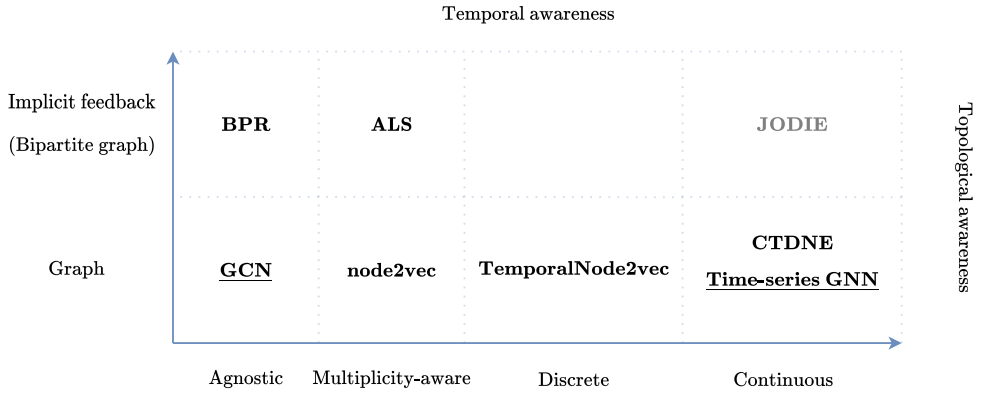


Figure 5.1: Taxonomy of the applied models. Approaches belonging to different classes "treat" a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ differently. For example, Alternating least squares is specifically designed for implicit feedback data (a special case of a bipartite graph). However, from the perspective of dynamics, it captures only the number of edges on the same vertices. Contrary, Time-series GNN is designed for a general dynamic graph but is able to capture continuous time. In the picture, graph neural networks are underlined.

I also want to mention the unfortunate attempts to apply some models to the considered task. In terms of this thesis, I implement the weg2vec (see Section 2.3.2.1) model, however it turns out that the construction of a weighted event graph is computationally infeasible due to the high density of the considered dynamic graphs. Note that I implement this model preserving its generality and do not employ any special steps to optimize it. Next, after multiple attempts, I was unlucky to install the DynGEM [95] library providing the implementation of several discrete-time graph machine learning models. Further, I had no success with JODIE[35] (see Section 2.5) application due to the limited GPU capacity.

[35] `https://github.com/srijankr/jodie`

Bayesian personalized ranking

To apply the Bayesian personalized ranking (BPR, see Section 1.5.1), for each dataset I construct the implicit feedback $\mathbf{R}$, as described in Section 4.2. I employ the existent implementation of BPR from the Implicit[36] library and run it with the default regularization parameter value $\lambda_1 = 0.01$, and $d = 15$ for both tasks based on the cross validation.

Alternating least squares

I use Alternating least squares (see Section 1.5.1) implementation from Implicit on the same constructed implicit feedback $\mathbf{R}$. ALS demonstrates the best performance with the same parameters as Bayesian personalized ranking. Note that the used implementation assumes $c_{ui} := \mathbf{R}[u, i]$.

Node2vec

I utilize the available node2vec (see Section 1.3.3) implementation[37] on the constructed multigraphs

$$G := \Big(\mathcal{V}, \{\{u, v\} \mid (u, v, t) \in \mathcal{E} \vee (v, u, t) \in \mathcal{E}\}\Big), \tag{5.4}$$

to capture the multiplicities of interactions. This model shows higher performance being set up to generate shorter random walks and use smaller sliding window of skip-gram architecture. To obtain the test scores I set walk length and window size to 2 with generating 200 walks per node for both tasks. The dimensionality d of embeddings is set to 64.

GCN

To run the graph convolutional neural network (GCN, see Section 1.4.2) I convert the dynamic graphs to single snapshots. I implement the model with PyTorch Geometric. Based on the cross-validation, I set the latent space dimensionality to 16, the number of layers to 1, and employ training with the learning rate set to 0.001 for 30 and 100 epochs on the maliciousness classification and foreign customer classification tasks respectively. Learning is done with Adam optimizer [96] with $\beta_1 = 0.9, \beta_2 = 0.999$.

TemporalNode2vec

To run TemporalNode2vec (see Section 2.3.1.3) I split the both dynamic graphs into 100 snapshots. The construction of **PPMI** matrices for each snapshot turns out to be too computationally expensive that is why I replace

[36] `https://github.com/benfred/implicit`
[37] `https://github.com/eliorc/node2vec`

them with the implicit feedback matrices **R** for each snapshot. It may be justified as an estimation of **PPMI** matrices constructed based on the random walks of length 2. I realize this model merging the mentioned implementation of node2vec and the original code of DynamicWord2vec[38]. In this case I set $d = 50$.

CTDNE

To apply CTDNE (see Section 2.3.2.1) I use the existent implementation[39]. Similarly as node2vec, on both tasks it achieves the best performance employing 2-nodes-long random walks. See Figure 5.2 for the illustration. The model shows better results utilizing the linear neighborhood sampling distribution. Thus, to test the model I finally set $l = 2$, $w = 200$, $d = 64$, exponential neighborhood sampling, and generation of 200 walks per node without random sampling. The used implementation assumes t to be the maximal timestamp.

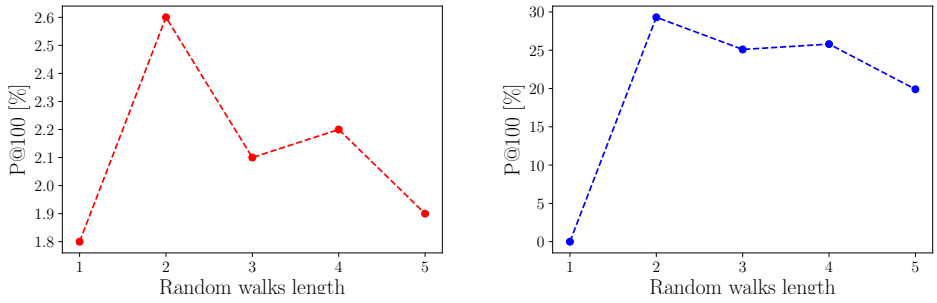


Figure 5.2: Influence of random walks length on the CTDNE validation performance on **(a)** the maliciousness classification task, **(b)** the foreign customer classification task. In both cases, the window length for skip-gram architecture is set to maximal.

Time-series GNN

In terms of this thesis, I implement a simple RNN-based layer of Time-series GNN (described at the end of Section 3.2), ad hoc for the considered problem, and leave the further study and experiments as future work.

I use one-hot-encoded node features to initialize the first hidden state of GNN. For the maliciousness classification task I set $d_2 = 30$ and for the foreign customer classification $d_2 = 15$. $d_1 = |V|$ is given by one-hot encoding. Next, for both problems 1-layer-deep RNN shows the best validation scores, which I use for the testing. I train the model with Adam optimizer [96]

[38] `https://github.com/yifan0sun/DynamicWord2Vec`
[39] `https://github.com/urielsinger/CTDNE`

($\beta_1 = 0.9, \beta_2 = 0.999$) for both problems with learning rate equal to 0.001 and 100 epochs.

5.4 Results

	All domains			Test domains				
	P@5	P@10	P@100	P@1	P@5	P@10	P@100	R@$\|\mathcal{T}\|$
BPR	**1.00**	**1.00**	**0.46**	**1.00**	**1.00**	**1.00**	**0.14**	0.58
ALS	**1.00**	**1.00**	**0.46**	**1.00**	**1.00**	**1.00**	**0.14**	**0.59**
Node2vec	**1.00**	**1.00**	0.33	**1.00**	0.20	0.10	0.02	0.05
GCN	**1.00**	0.90	0.40	0.00	0.80	0.70	0.13	**0.59**
TemporalNode2vec	**1.00**	**1.00**	0.36	**1.00**	0.36	0.22	0.04	0.11
CTDNE	**1.00**	**1.00**	0.34	**1.00**	0.20	0.10	0.02	0.05
Time-series GNN	**1.00**	**1.00**	0.45	**1.00**	**1.00**	**1.00**	0.13	**0.59**

Table 5.1: Models performance on the maliciousness classification task.

	All customers		Test customers						
	P@200	P@500	P@1	P@5	P@10	P@100	P@200	P@500	R@$\|\mathcal{T}\|$
BPR	**1.00**	0.75	**1.00**	**1.00**	**1.00**	0.93	0.59	0.28	0.68
ALS	**1.00**	0.76	**1.00**	**1.00**	**1.00**	0.94	0.64	0.29	0.75
Node2vec	**1.00**	**0.80**	**1.00**	**1.00**	**1.00**	**1.00**	**0.74**	**0.31**	0.87
GCN	**1.00**	0.65	**1.00**	**1.00**	**1.00**	0.94	0.59	0.30	**0.89**
TemporalNode2vec	**1.00**	0.64	**1.00**	0.88	0.82	0.46	0.33	0.19	0.34
CTDNE	**1.00**	**0.80**	**1.00**	**1.00**	**1.00**	**1.00**	**0.74**	**0.31**	0.87
Time-series GNN	**1.00**	0.76	**1.00**	**1.00**	**1.00**	0.97	0.66	0.28	0.83

Table 5.2: Models performance on the foreign customer classification task.

The results of experiments conducted on the considered problem (see Chapter 4 for the details) are summarized in Tables 5.1 and 5.2. Figures 5.3, 5.4, 5.5, 5.7, 5.8, 5.9 depict the latent spaces of user and domain embeddings. For the visualizations of the customer and product embeddings see Appendix A. The illustrated 2-dimensional vectors are obtained using dimensionality reduction of higher-dimensional latent spaces. I employ[40] UMAP [97] to achieve this. Note also that the colored markers are rendered in the order corresponding to their appearance in the legend. Consequently, for example, the low number of visible yellow and red points means that they are clustered well together. Remarkable are the clusters of malicious domains in the latent spaces of the models that demonstrate high performance. Moreover, note the difference between the embeddings produced in a two-step unsupervised manner and the ones learned supervisely by graph neural networks.

First of all, results demonstrate that the Cisco Cognitive maliciousness classification problem defined in Chapter 4 can be solved quite efficiently. BPR, ALS and Time-series GNN show P@10 = 1.00 = 100[%] on the test

[40] `https://umap-learn.readthedocs.io/en/latest/`

domains. It means that these models, given a task to infer 10 malicious domains based only on user connections, have chosen exactly some of the 22, hidden during the training, malicious domains out of 3364 candidates. Moreover, BPR, ALS and Time-series GNN demonstrate R@$|\mathcal{T}|$ = R@$|22|$ = 0.59 = 59[%], which means that 13 of 22 hidden malicious domains were in top-22. Similar observations can be done regarding the performance of models on the foreign customer classification task.

Next, we can observe the superior performance of implicit-feedback-based models on the domain classification problem, and random-walks-based techniques supremacy on the customer classification task. TemporalNode2vec consistently shows the worst scores and GCN demonstrates decent results on both problems. Time-series GNN demonstrates the stable second-best results, which I find satisfactory considering the conclusions discussed below.

5.5 Discussion

Comparison of the different approaches efficiency on the maliciousness classification task leads to the following generalized conclusions:

1. **Patterns determining the maliciousness of domains lie in their 1-hop neighborhoods.**

   Assume for example that malicious domains infect users. These users are then utilized to employ a distributed denial-of-service attack (DDoS attack[41]). To capture this malevolent pattern, one has to consider at least two-hop neighborhood of the attacker's domain: the infected users and the domains visited by these users. Another example of a deep structural pattern may be the presence of a lateral movement [42] — the process of malware propagation through the network. In this case, it is necessary to consider the temporal walks on a graph to express this process.

   Note that the CTDNE model described in Section 2.3.2.1 is exactly designed to accurately capture the processes of temporal information propagation in dynamic networks. However, Figure 5.2 demonstrates that the employment of longer temporal walks reduces the efficiency of the malicious domain exposure. It means that at least the majority of, present during the training, malicious domains are not identified by the malevolent patterns described above or any other deeper topological structures. This assumption is highly supported by the impressive performance of BPR and ALS models which are designed to focus on

[41] https://en.wikipedia.org/wiki/Denial-of-service_attack
[42] https://en.wikipedia.org/wiki/Network_Lateral_Movement

the direct user-item interactions. Moreover, GCN, being able to capture the node neighborhoods of arbitrary depth, demonstrates the best validation score initialized to be 1-layer deep.

2. **Patterns determining the maliciousness are not time-dependent.**

   Comparison of random-walk-based approaches leads to the interesting observation: node2vec, employing time-agnostic random walks achieves almost exactly the same performance as CTDNE that utilizes the walks that respect the natural ordering of interactions. Nguyen et al. — the authors of CTDNE — reasonably state that the time-agnostic walks are invalid when considering a dynamic graph. The equivalence of temporal and time-agnostic walks utilization means that the patterns identifying maliciousness do not exhibit time-related properties. Again, this assumption is significantly supported by the high performance of temporal-unaware baseline approaches.

The direct combination of two points discussed above leads to the following conclusion:

The maliciousness of a domain is determined by its visitors.

This conclusion is also supported by the experimental results of the foreign customer classification. On this problem, CTDNE and node2vec achieve identical performance as well. However, in this case, these models are superior to the other ones meaning that, regarding this task, the argument of temporal patterns absence is even more convincing. Interestingly, manual analysis of the considered e-commerce data shows that many foreign customers can be determined by the purchases of a specific product: the postage. Considering that this problem was introduced as a task related to the maliciousness classification, the fact that foreign customers are determined by the specific products leads another way to the conclusion above.

Importantly, the drawn conclusions hold regarding the defined problem and it is necessary to consider its possible important extensions which outline the possible future steps. First of all, the term maliciousness can be captured on its finer granularity, considering different types of threats. Probably, distinguishing between different kinds of maliciousness, one could observe the incapability of time-agnostic baseline methods to expose certain types of dangerous domains. However, in this case, the task becomes even more challenging due to the low amount of labeled data. Next, I want to remind that in this work I focused on the user-domain interactions, however it is also possible to consider full URLs and/or hostnames as well (see Table 4.1). The further important remark is given by the utilized data. In this work, I reduced

the number of unlabeled domains due to the limited computational capacity. Nevertheless, based on the discussed experimental results I assume that the consideration of the whole dataset would not significantly affect the results. Finally, I want to mention that it may be also important to study the other possible ways to collect the data in order to efficiently utilize the intrinsic dynamics of user-domain interactions.

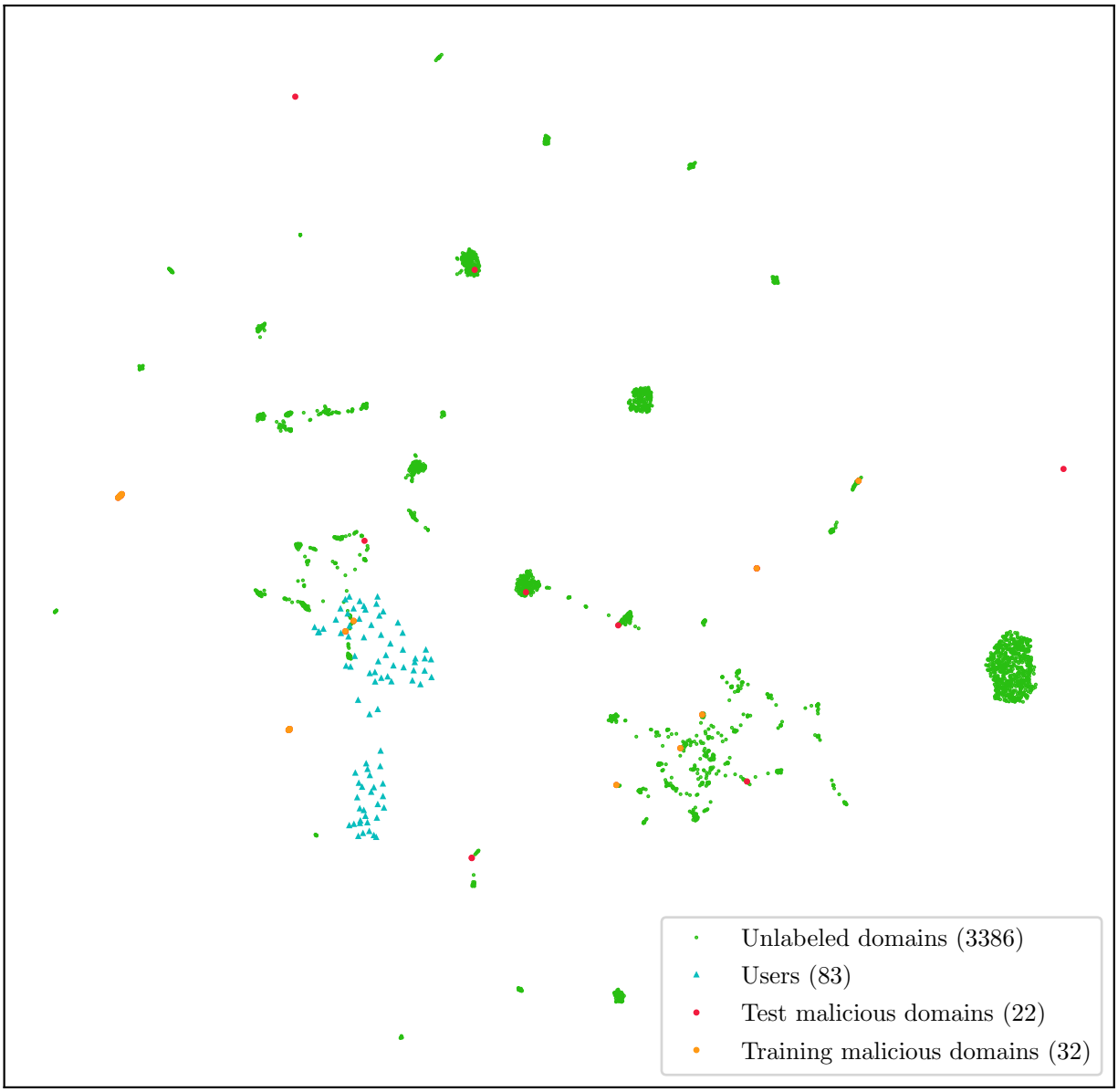


Figure 5.3: Unsupervized node embeddings with Bayesian Personalized Ranking (BPR) on the maliciousness classification task. Markers are rendered in the order corresponding to their appearance in the legend.

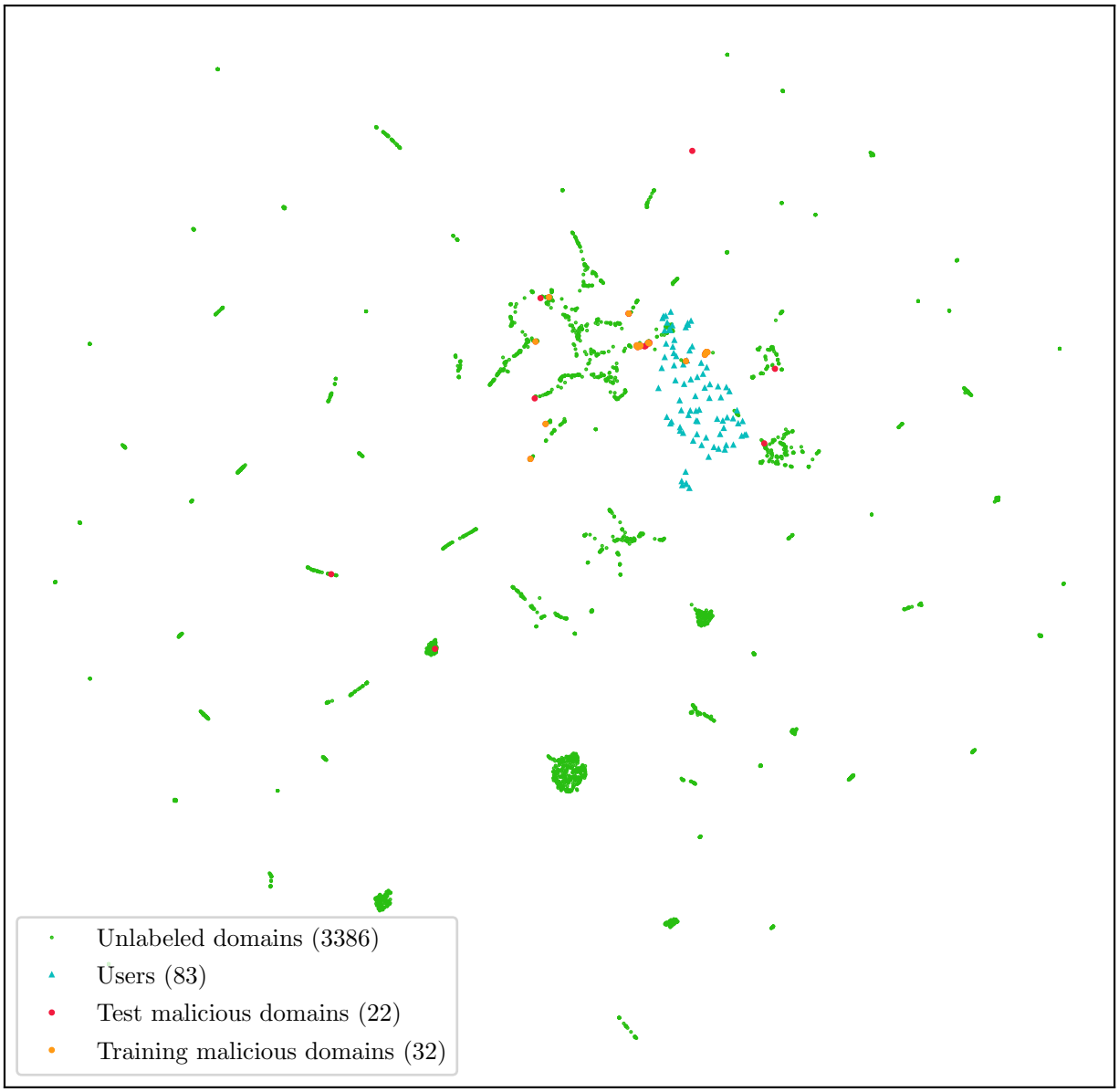


Figure 5.4: Unsupervized node embeddings with Alternating Least Squares (ALS) on the maliciousness classification task. Markers are rendered in the order corresponding to their appearance in the legend.

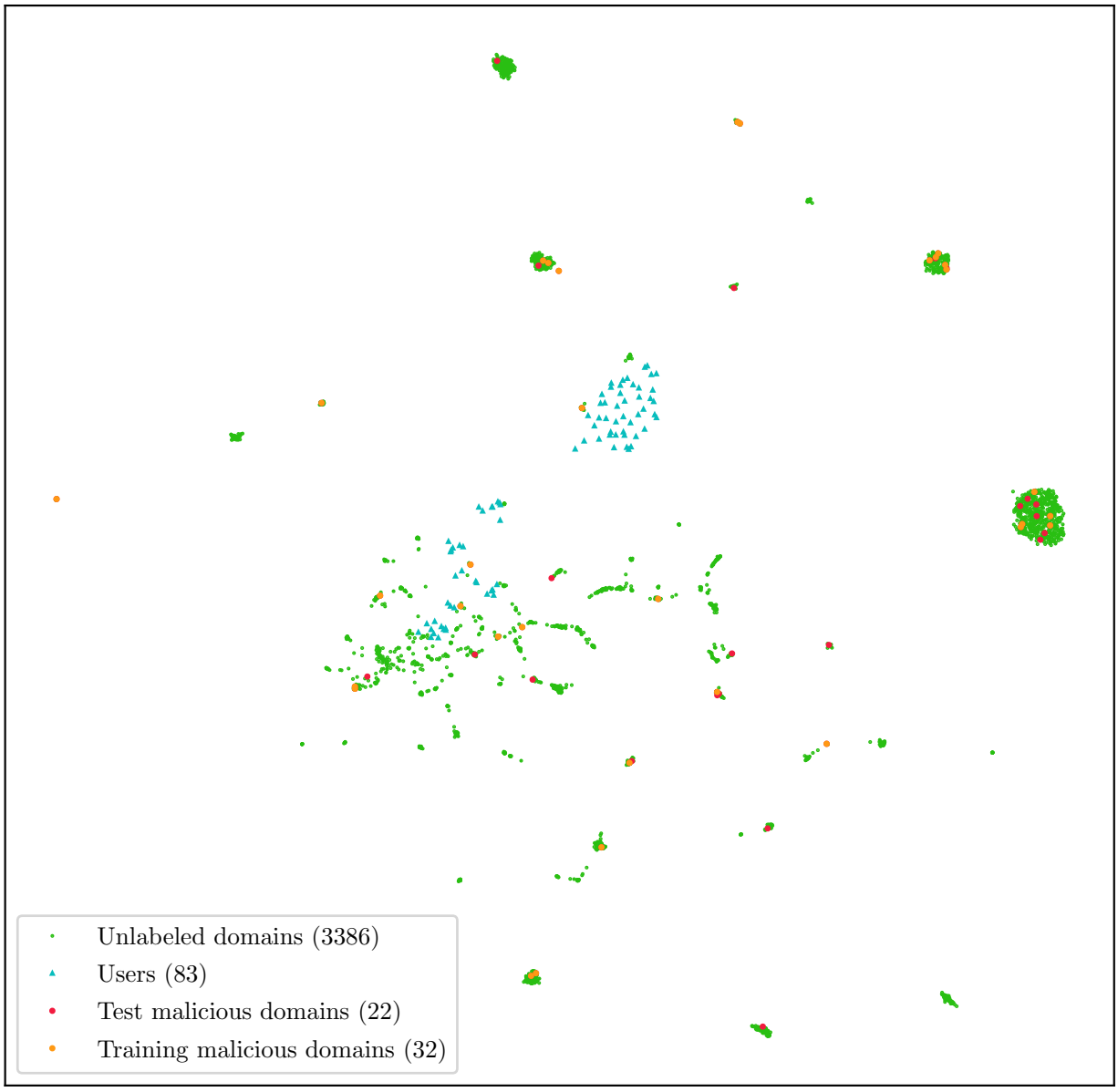


Figure 5.5: Unsupervized node embeddings with node2vec on the maliciousness classification task. Markers are rendered in the order corresponding to their appearance in the legend.

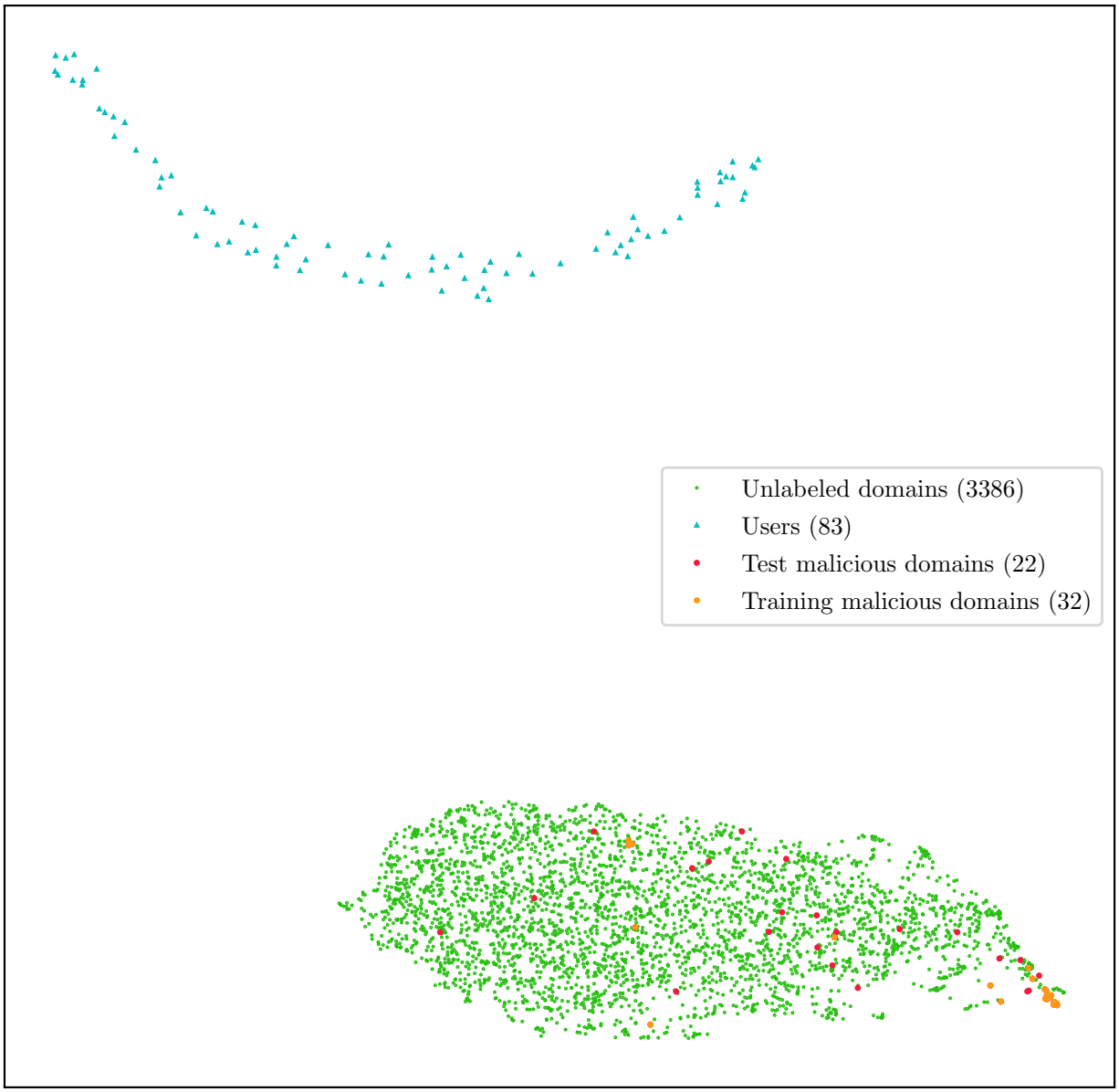


Figure 5.6: Supervized node embeddings with Graph convolutional neural network (GCN) on the maliciousness classification task. Markers are rendered in the order corresponding to their appearance in the legend.

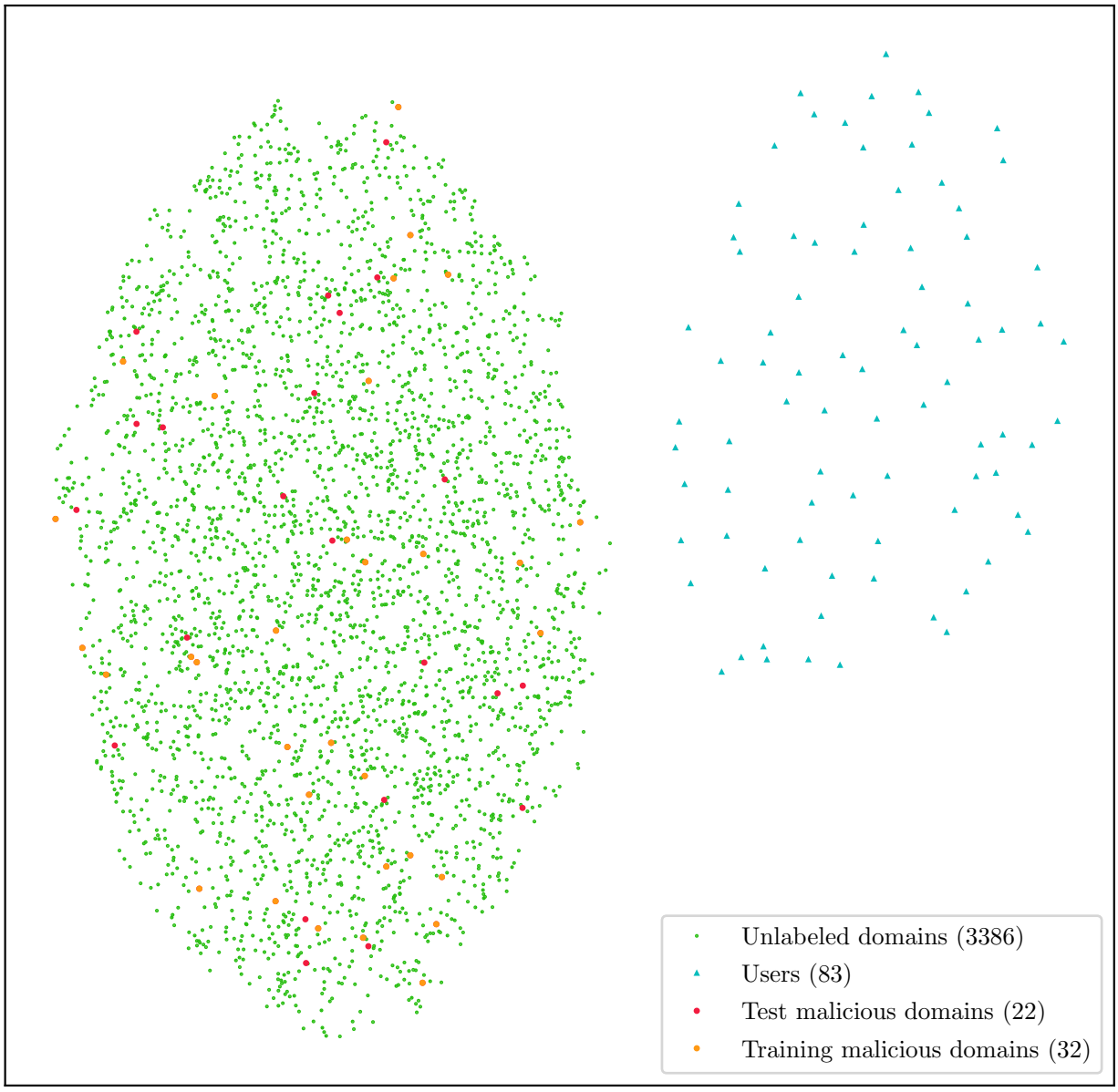


Figure 5.7: Unsupervized node embeddings with TemporalNode2vec on the maliciousness classification task. Markers are rendered in the order corresponding to their appearance in the legend.

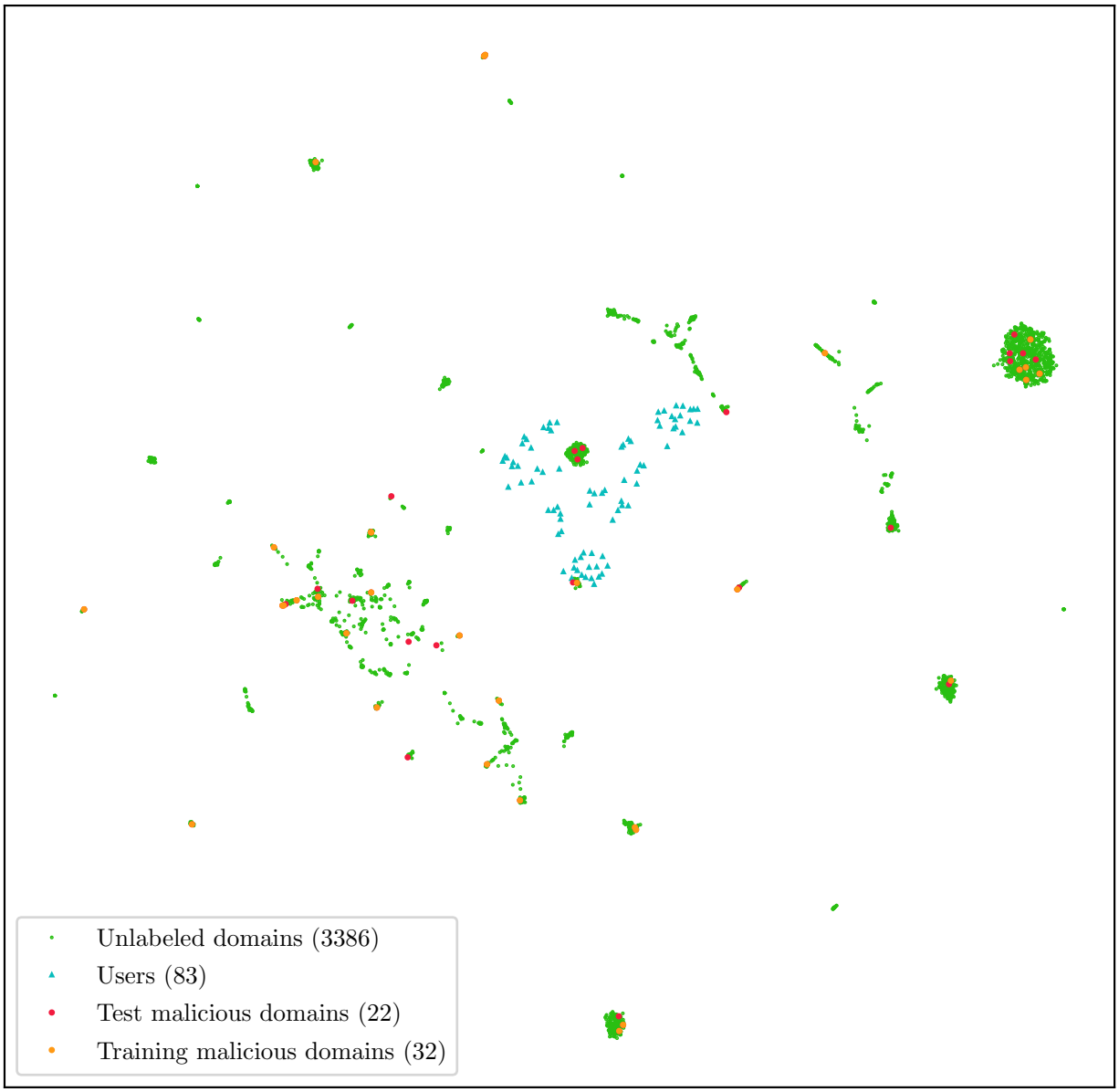


Figure 5.8: Unsupervized node embeddings with CTDNE on the maliciousness classification task. Markers are rendered in the order corresponding to their appearance in the legend.

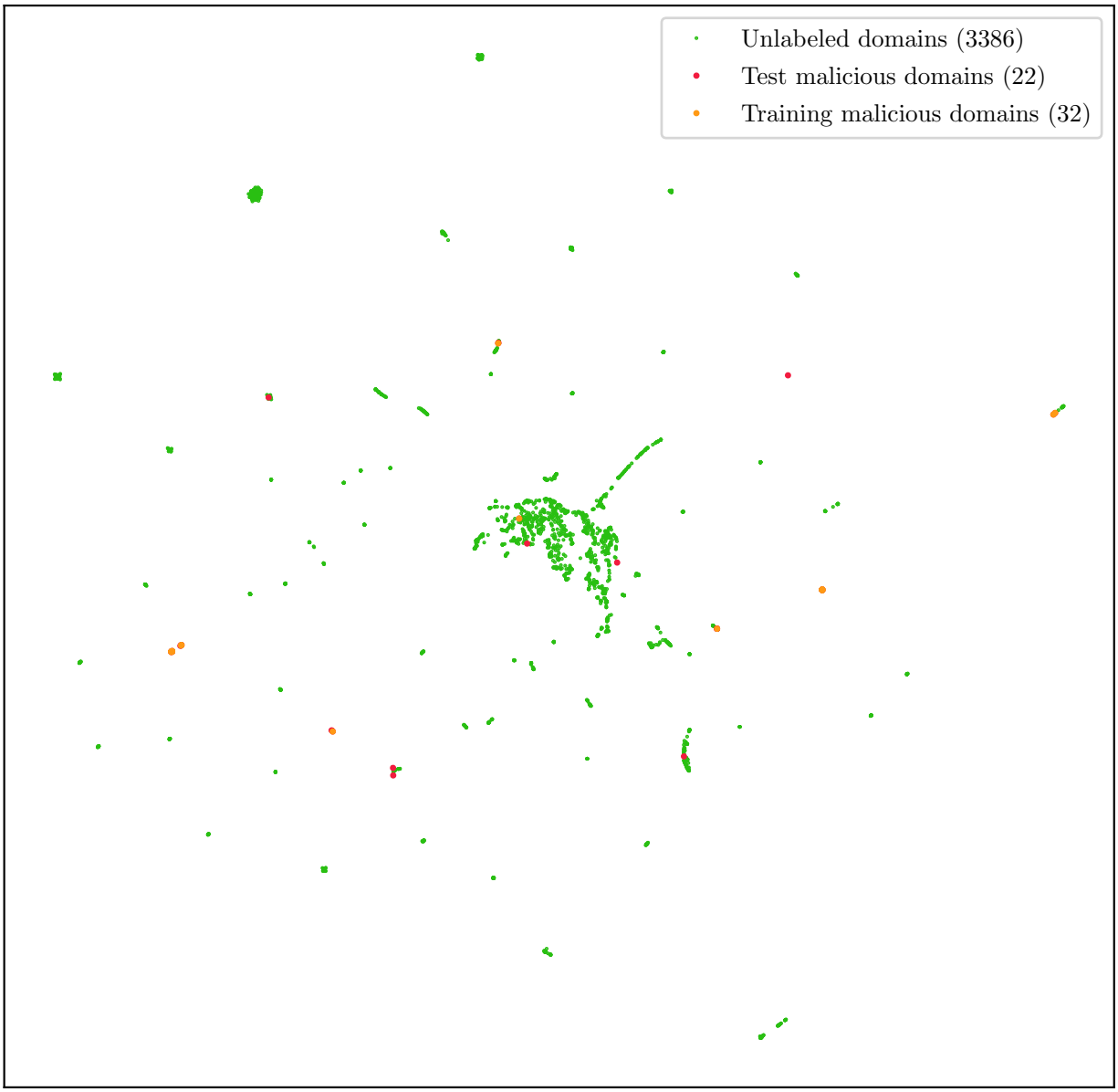


Figure 5.9: Supervized node embeddings with Time-series GNN on the maliciousness classification task. Markers are rendered in the order corresponding to their appearance in the legend.

Conclusion

In this thesis, I studied the Cisco Cognitive Intelligence maliciousness classification problem — the task of malicious Internet domain exposure based on the user-domain interactions — from the perspective of machine learning on dynamic graphs. I find this thesis a solid groundwork for potential future research and experiments.

In the first place, I conducted a study of the state of the art of machine learning on dynamic graphs. The outcome of this work is embodied in the proposed continuous-time dynamic graph neural network model. I believe that the designed approach can be useful in many real-world applications, however, the corresponding experiments and deeper research are out of the scope of this thesis and I leave it as an intriguing future work.

The second part of this thesis is a study of the temporal aspect awareness on the performance of machine learning models regarding the Cisco Cognitive Intelligence maliciousness classification problem. In this work, I showed that the considered task can be solved quite efficiently utilizing the different machine learning approaches, including the proposed continuous-time dynamic graph neural network model. The used approaches demonstrate the capability of revealing malicious domains and can be used in practice.

However, despite the intrinsic dynamics of user-domain interactions, the comparison of efficiency of multiple time-agnostic and temporal-aware approaches demonstrates that the maliciousness, expressed in the currently obtained data, does not exhibit dynamic properties. Thus, the utilization of temporal aspect does not improve the efficiency of machine learning applications to the considered problem. This fact suggests the possible future steps: (*i*) to study the used data collection process and conduct the experiments on differently collected records, (*ii*) to study the maliciousness on its finer granularity, considering, for instance different types of threats. I believe in significance of considering the temporal aspect and that this work is a step towards its efficient utilization.

Bibliography

[1] Amol Kapoor, Xue Ben, Luyang Liu, Bryan Perozzi, Matt Barnes, Martin Blais, and Shawn O'Banion. Examining covid-19 forecasting using spatio-temporal graph neural networks. *arXiv preprint arXiv:2007.03113*, 2020.

[2] Aynaz Taheri and Tanya Berger-Wolf. Predictive temporal embedding of dynamic graphs. In *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '19, page 57–64, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368681. doi: 10.1145/3341161.3342872. URL https://doi.org/10.1145/3341161.3342872.

[3] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. volume abs/1908.01207, 2019. URL http://arxiv.org/abs/1908.01207.

[4] William L Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

[5] Nitin Gupta, Kapil Mangla, Anand Kumar Jha, and Md. Umar. Applying dijkstra's algorithm in routing process. *International Journal of New Technology and Research (IJNTR)*, 2:1–159, 2016. URL https://www.ijntr.org/download_data/IJNTR02050040.pdf.

[6] Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackermann, Victoria M. Tran, Anush Chiappino-Pepe, Ahmed H. Badran, Ian W. Andrews, Emma J. Chory, George M. Church, Eric D. Brown, Tommi S. Jaakkola, Regina Barzilay, and James J. Collins. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702.e13, 2020. ISSN 0092-

8674. doi: https://doi.org/10.1016/j.cell.2020.01.021. URL https://www.sciencedirect.com/science/article/pii/S0092867420301021.

[7] Medhini Narasimhan, Svetlana Lazebnik, and Alexander G. Schwing. Out of the box: Reasoning with graph convolution nets for factual visual question answering, 2018.

[8] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, page 585–591, Cambridge, MA, USA, 2001. MIT Press.

[9] Michelle L Rittenhouse. Properties and recent applications in spectral graph theory. *Virginia Commonwealth University*, 2008. URL https://scholarscompass.vcu.edu/etd/1126/.

[10] Tim Roughgarden and Gregory Valiant. The modern algorithmic toolbox lectures #11: Spectral graph theory, i. *Stanford University*, 2020. URL https://web.stanford.edu/class/cs168/l/l11.pdf.

[11] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[12] Daniel Lee and H. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401:788–91, 11 1999. doi: 10.1038/44565.

[13] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. URL https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf.

[14] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J. Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 37–48, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320351. doi: 10.1145/2488388.2488393. URL https://doi.org/10.1145/2488388.2488393.

[15] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, page 891–900, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337946. doi: 10.1145/2806416.2806512. URL https://doi.org/10.1145/2806416.2806512.

[16] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1105–1114, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939751. URL https://doi.org/10.1145/2939672.2939751.

[17] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[18] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks, 2016.

[19] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, Aug 2014. doi: 10.1145/2623330.2623732. URL http://dx.doi.org/10.1145/2623330.2623732.

[20] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line. *Proceedings of the 24th International Conference on World Wide Web*, May 2015. doi: 10.1145/2736277.2741093. URL http://dx.doi.org/10.1145/2736277.2741093.

[21] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. volume abs/1710.02971, 2017. URL http://arxiv.org/abs/1710.02971.

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[23] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013. URL https://arxiv.org/abs/1312.6203.

[24] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL http://arxiv.org/abs/1609.02907.

[25] Jure Leskovec. Lecture 8 graph neural networks. *Stanford CS224W - Machine Learning with graphs - Fall 2019*, Autumn 2019. URL https://www.youtube.com/watch?v=LdK9HzBAR8c&list=PL-Y8zK4dwCrQyASidb2mjj_itW2-YYx6-&index=8.

[26] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[27] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[28] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. 2021.

[29] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE International Conference on Data Mining*, pages 263–272. Ieee, 2008. URL `http://yifanhu.net/PUB/cf.pdf`.

[30] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618*, 2012.

[31] Yulong Pei, Jianpeng Zhang, GH Fletcher, and Mykola Pechenizkiy. Node classification in dynamic social networks. *Proceedings of AALTD*, page 54, 2016. URL `https://kulak.kuleuven.be/benelearn/papers/Benelearn_2016_paper_48.pdf`.

[32] Ryan Rossi, Brian Gallagher, Jennifer Neville, and Keith Henderson. Modeling dynamic behavior in large evolving graphs. 02 2013. doi: 10.1145/2433396.2433479.

[33] Yuqi Li, Weizheng Chen, and Hongfei Yan. Learning graph-based embedding for time-aware product recommendation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, page 2163–2166, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349185. doi: 10.1145/3132847.3133060. URL `https://doi.org/10.1145/3132847.3133060`.

[34] Yupu Guo, Yanxiang Ling, and Honghui Chen. A time-aware graph neural network for session-based recommendation. *IEEE Access*, 8:167371–167382, 2020. doi: 10.1109/ACCESS.2020.3023685.

[35] Wenchao Yu, Charu C. Aggarwal, and Wei Wang. Temporally factorized network modeling for evolutionary network analysis. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, page 455–464, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346757. doi: 10.1145/3018661.3018669. URL `https://doi.org/10.1145/3018661.3018669`.

[36] Maddalena Torricelli, Márton Karsai, and Laetitia Gauvin. weg2vec: Event embedding for temporal networks, 2019.

[37] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *CoRR*, abs/1012.0009, 2010. URL http://arxiv.org/abs/1012.0009.

[38] Barbara Guidi, Andrea Michienzi, Laura Ricci, Chrysanthi Iakovidou, and Symeon Papadopoulos. Define a time-dependent social graph. *Networks*, 4:2, 2019. URL https://helios-h2020.eu/wp-content/uploads/2020/05/D4.2_Define-a-time-dependent-social-graph.pdf.

[39] Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *CoRR*, abs/1710.04073, 2017. URL http://arxiv.org/abs/1710.04073.

[40] F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–87, 1997. ISSN 0895-7177. doi: https://doi.org/10.1016/S0895-7177(97)00050-2. URL https://www.sciencedirect.com/science/article/pii/S0895717797000502.

[41] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. 2020.

[42] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. Foundations and modelling of dynamic networks using dynamic graph neural networks: A survey. *CoRR*, abs/2005.07496, 2020. URL https://arxiv.org/abs/2005.07496.

[43] Claudio DT Barros, Matheus RF Mendonça, Alex B Vieira, and Artur Ziviani. A survey on embedding dynamic graphs. *arXiv preprint arXiv:2101.01229*, 2021.

[44] Yuanhua Lv and ChengXiang Zhai. Positional language models for information retrieval. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 299–306, 2009.

[45] Mathieu G'enois and Alain Barrat. Can co-location be used as a proxy for face-to-face contacts? *EPJ Data Science*, 7(1):11, May 2018. ISSN 2193-1127. doi: 10.1140/epjds/s13688-018-0140-1. URL https://doi.org/10.1140/epjds/s13688-018-0140-1.

[46] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. page 556–559, 2003. doi: 10.1145/956863.956972. URL https://doi.org/10.1145/956863.956972.

[47] Nahla Mohamed Ahmed, Ling Chen, Yulong Wang, Bin Li, Yun Li, and Wei Liu. Sampling-based algorithm for link prediction in temporal networks. *Inf. Sci.*, 374(C):1–14, December 2016. ISSN 0020-0255. doi: 10.1016/j.ins.2016.09.029. URL https://doi.org/10.1016/j.ins.2016.09.029.

[48] Nahla Mohamed Ibrahim and Ling Chen. Link prediction in dynamic social networks by integrating different types of information. *Applied Intelligence*, 42(4):738–750, June 2015. ISSN 0924-669X. doi: 10.1007/s10489-014-0631-0. URL https://doi.org/10.1007/s10489-014-0631-0.

[49] Lin Yao, Luning Wang, Lv Pan, and Kai Yao. Link prediction based on common-neighbors for dynamic social network. *Procedia Computer Science*, 83:82–89, 12 2016. doi: 10.1016/j.procs.2016.04.102.

[50] Linhong Zhu, Dong Guo, Junming Yin, Greg Ver Steeg, and Aram Galstyan. Scalable temporal latent space inference for link prediction in dynamic social networks. *IEEE Transactions on Knowledge and Data Engineering*, 28(10):2765–2777, 2016. doi: 10.1109/TKDE.2016.2591009.

[51] İsmail Güneş, Sule Gunduz Oguducu, and Zehra Cataltepe. Link prediction using time series of neighborhood-based node similarity scores. *Data Mining and Knowledge Discovery*, 30, 02 2015. doi: 10.1007/s10618-015-0407-0.

[52] Mounir Haddad, Cécile Bothorel, Philippe Lenca, and Dominique Bedart. Temporalnode2vec: Temporal node embedding in temporal networks. In Hocine Cherifi, Sabrina Gaito, José Fernendo Mendes, Esteban Moro, and Luis Mateus Rocha, editors, *Complex Networks and Their Applications VIII*, pages 891–902, Cham, 2020. Springer International Publishing. URL https://hal.archives-ouvertes.fr/hal-02332080/document.

[53] Carlos Henrique Gomes Ferreira, Fabricio Murai Ferreira, Breno de Sousa Matos, and Jussara Marques de Almeida. Modeling dynamic ideological behavior in political networks. *The Journal of Web Science*, 7, 2019.

[54] Zijun Yao, Yifan Sun, Weicong Ding, Nikhil Rao, and Hui Xiong. Discovery of evolving semantics through dynamic word embedding learning. volume abs/1703.00607, 2017. URL http://arxiv.org/abs/1703.00607.

[55] Daniel M. Dunlavy, Tamara G. Kolda, and Evrim Acar. Temporal link prediction using matrix and tensor factorizations. *ACM Trans. Knowl. Discov. Data*, 5(2), February 2011. ISSN 1556-4681. doi: 10.1145/1921632.1921636. URL https://doi.org/10.1145/1921632.1921636.

[56] Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. Introduction to tensor decompositions and their applications in machine learning, 2017.

[57] Giang Nguyen, John Lee, Ryan Rossi, Nesreen Ahmed, Eunyee Koh, and Sungchul Kim. Continuous-time dynamic network embeddings. pages 969–976, 04 2018. doi: 10.1145/3184558.3191526.

[58] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *CoRR*, abs/1809.04356, 2018. URL `http://arxiv.org/abs/1809.04356`.

[59] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. Structured sequence modeling with graph convolutional recurrent networks. 2016.

[60] Franco Manessi, Alessandro Rozza, and Mario Manzo. Dynamic graph convolutional networks. *Pattern Recognition*, 97:107000, 2020. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2019.107000. URL `https://www.sciencedirect.com/science/article/pii/S0031320319303036`.

[61] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, and Charles E. Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. *CoRR*, abs/1902.10191, 2019. URL `http://arxiv.org/abs/1902.10191`.

[62] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[63] Jinyin Chen, Xuanheng Xu, Yangyang Wu, and Haibin Zheng. GC-LSTM: graph convolution embedded LSTM for dynamic link prediction. *CoRR*, abs/1812.04206, 2018. URL `http://arxiv.org/abs/1812.04206`.

[64] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. Streaming graph neural networks. SIGIR '20, page 719–728, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380164. doi: 10.1145/3397271.3401092. URL `https://doi.org/10.1145/3397271.3401092`.

[65] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. Dyrep: Learning representations over dynamic graphs. In *International Conference on Learning Representations*, 2019. URL `https://par.nsf.gov/servlets/purl/10119521`.

[66] Dimitrios Rafailidis and Alexandros Nanopoulos. Modeling the dynamics of user preferences in coupled tensor factorization. RecSys '14, page 321–324, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326681. doi: 10.1145/2645710.2645758. URL https://doi.org/10.1145/2645710.2645758.

[67] Xiaomin Fang, Rong Pan, Guoxiang Cao, Xiuqiang He, and Wenyuan Dai. Personalized tag recommendation through nonlinear tensor factorization using gaussian kernel. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 439–445. AAAI Press, 2015. ISBN 0262511290.

[68] Serkan Kiranyaz, Onur Avci, Osama Abdeljaber, Turker Ince, Moncef Gabbouj, and Daniel J. Inman. 1d convolutional neural networks and applications: A survey. 2019.

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[70] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *Ndss*, pages 1–17, 2011.

[71] Yury Zhauniarovich, Issa Khalil, Ting Yu, and Marc Dacier. A survey on malicious domains detection through DNS data analysis. *CoRR*, abs/1805.08426, 2018. URL http://arxiv.org/abs/1805.08426.

[72] Kazumichi Sato, Keisuke Ishibashi, Tsuyoshi Toyono, and Nobuhisa Miyake. Extending black domain name list by using co-occurrence relation between dns queries. *IEICE Transactions on Communications*, E95B:8–8, 03 2012. doi: 10.1587/transcom.E95.B.794.

[73] Daiki Chiba, Takeshi Yagi, Mitsuaki Akiyama, Toshiki Shibahara, Takeshi Yada, Tatsuya Mori, and Shigeki Goto. Domainprofiler: Discovering domain names abused in future. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 491–502, 2016. doi: 10.1109/DSN.2016.51.

[74] Futai Zou, Siyu Zhang, Weixiong Rao, and Ping yi. Detecting malware based on dns graph mining. *International Journal of Distributed Sensor Networks*, 2015:1–12, 10 2015. doi: 10.1155/2015/102687.

[75] Nizar Kheir, Frédéric Tran, Pierre Caron, and Nicolas Deschamps. Mentor: Positive dns reputation to skim-off benign domains in botnet c&c blacklists. In Nora Cuppens-Boulahia, Frédéric Cuppens, Sushil Jajodia, Anas Abou El Kalam, and Thierry Sans, editors, *ICT Systems Security*

and Privacy Protection, pages 1–14, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-55415-5.

[76] Matija Stevanovic, Jens Pedersen, Alessandro D'Alconzo, Stefan Ruehrup, and Andreas Berger. On the ground truth problem of malicious dns traffic analysis. *Computers & Security*, 55, 09 2015. doi: 10.1016/j.cose.2015.09.004.

[77] Xuhong Wang, Baihong Jin, Ying Du, Ping Cui, and Yupu Yang. One-class graph neural networks for anomaly detection in attributed networks. *arXiv preprint arXiv:2002.09594*, 2020.

[78] Xian Wu, Baoxu Shi, Yuxiao Dong, Chao Huang, and Nitesh V. Chawla. Neural tensor factorization for temporal interaction learning. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, page 537–545, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359405. doi: 10.1145/3289600.3290998. URL `https://doi.org/10.1145/3289600.3290998`.

[79] Xiaohan Li, Mengqi Zhang, Shu Wu, Zheng Liu, Liang Wang, and Philip S Yu. Dynamic graph collaborative filtering. *arXiv preprint arXiv:2101.02844*, 2020.

[80] Esther Rodrigo Bonet, Duc Minh Nguyen, and Nikos Deligiannis. Temporal collaborative filtering with graph convolutional neural networks. *arXiv preprint arXiv:2010.06425*, 2020.

[81] Karthik Raja Kalaiselvi Bhaskar, Deepa Kundur, and Yuri Lawryshyn. Implicit feedback deep collaborative filtering product recommendation system. *arXiv preprint arXiv:2009.08950*, 2020.

[82] Guillaume Lemaître, Fernando Nogueira, and Christos K Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research*, 18(1):559–563, 2017. URL `https://www.jmlr.org/papers/volume18/16-365/16-365.pdf`.

[83] Justin Johnson and Taghi Khoshgoftaar. Survey on deep learning with class imbalance. *Journal of Big Data*, 6:27, 03 2019. doi: 10.1186/s40537-019-0192-5.

[84] Shoujin Wang, Wei Liu, Jia Wu, Longbing Cao, Qinxue Meng, and Paul J. Kennedy. Training deep neural networks on imbalanced data sets. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 4368–4374, 2016. doi: 10.1109/IJCNN.2016.7727770.

[85] Roi Blanco and Christina Lioma. Graph-based term weighting for information retrieval. *Inf. Retr.*, 15(1):54–92, February 2012. ISSN 1386-4564. doi: 10.1007/s10791-011-9172-x. URL https://doi.org/10.1007/s10791-011-9172-x.

[86] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[87] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL http://networkrepository.com.

[88] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[89] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[90] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL https://doi.org/10.5281/zenodo.3509134.

[91] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[92] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia

Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[93] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[94] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[95] Palash Goyal, Sujit Rokka Chhetri, Ninareh Mehrabi, Emilio Ferrara, and Arquimedes Canedo. Dynamicgem: A library for dynamic graph embedding methods. *arXiv preprint arXiv:1811.10734*, 2018.

[96] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1412.6980`.

[97] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020. URL `https://umap-learn.readthedocs.io/en/latest/`.

Appendix A

Visualization of node embeddings on the foreign customer classification problem

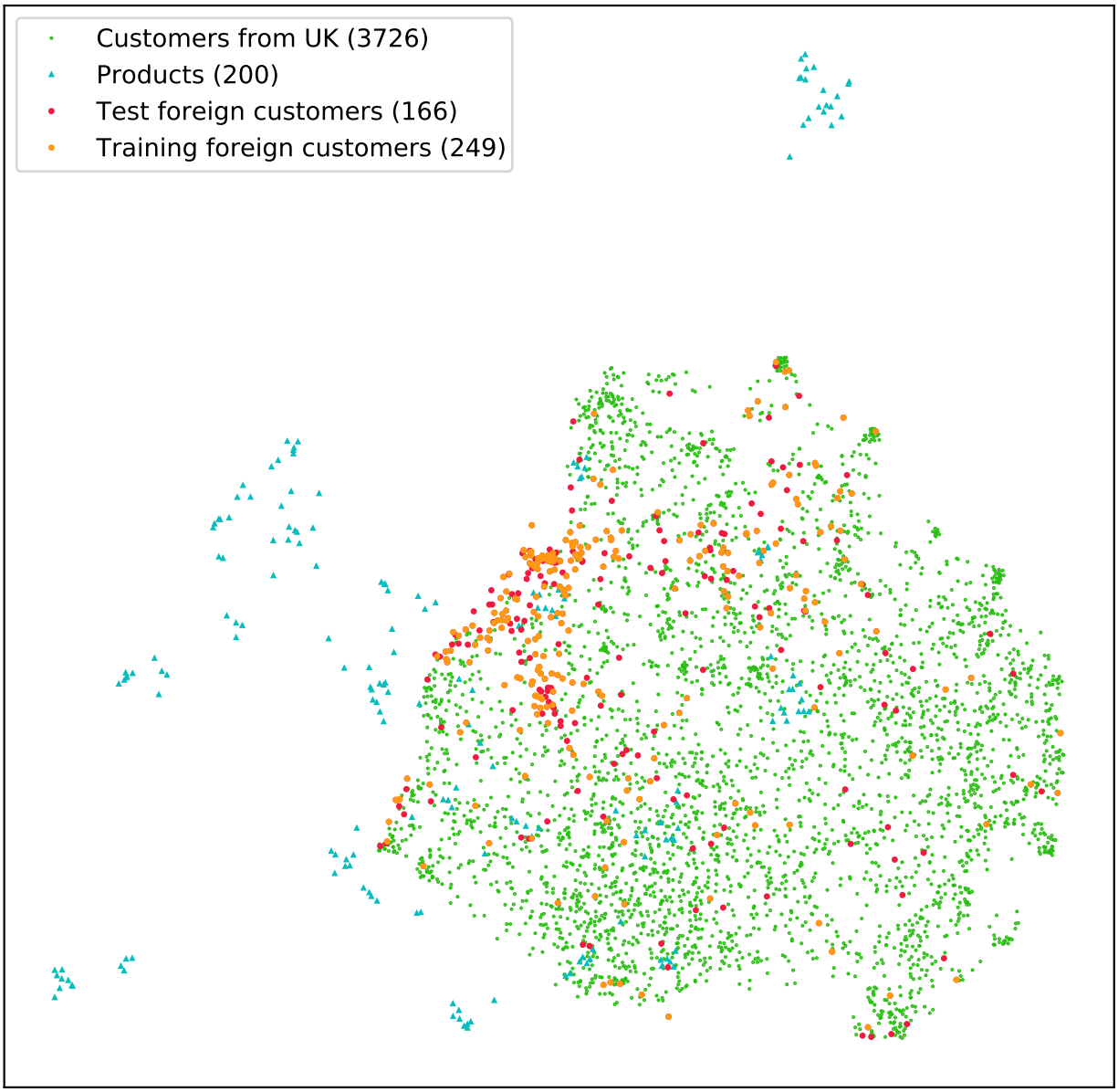


Figure A.1: Unsupervized node embeddings with Bayesian Personalized Ranking (BPR) on the foreign customer classification task. Markers are rendered in the order corresponding to their appearance in the legend.

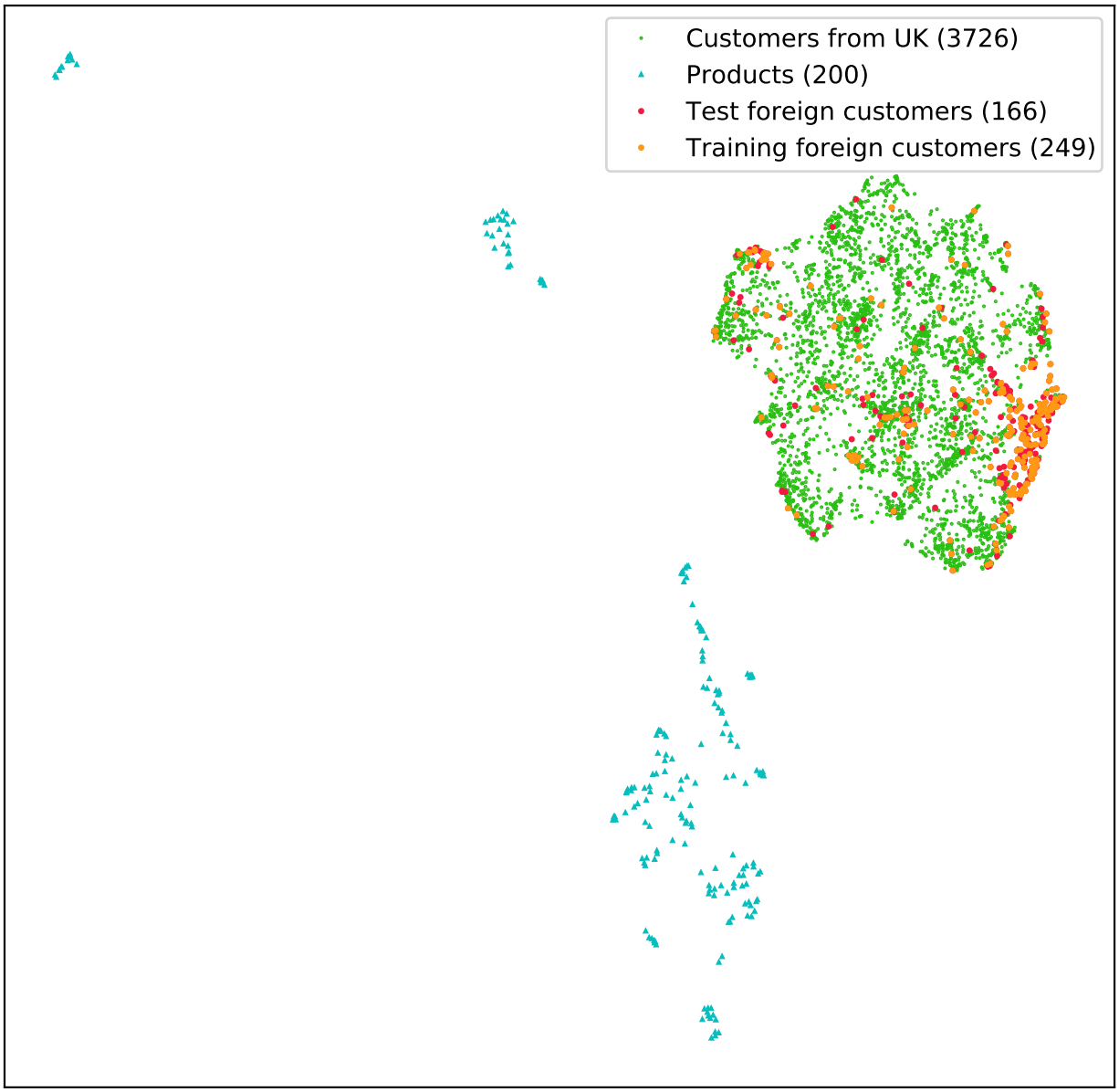


Figure A.2: Unsupervized node embeddings with Alternating Least Squares (ALS) on the foreign customer classification task. Markers are rendered in the order corresponding to their appearance in the legend.

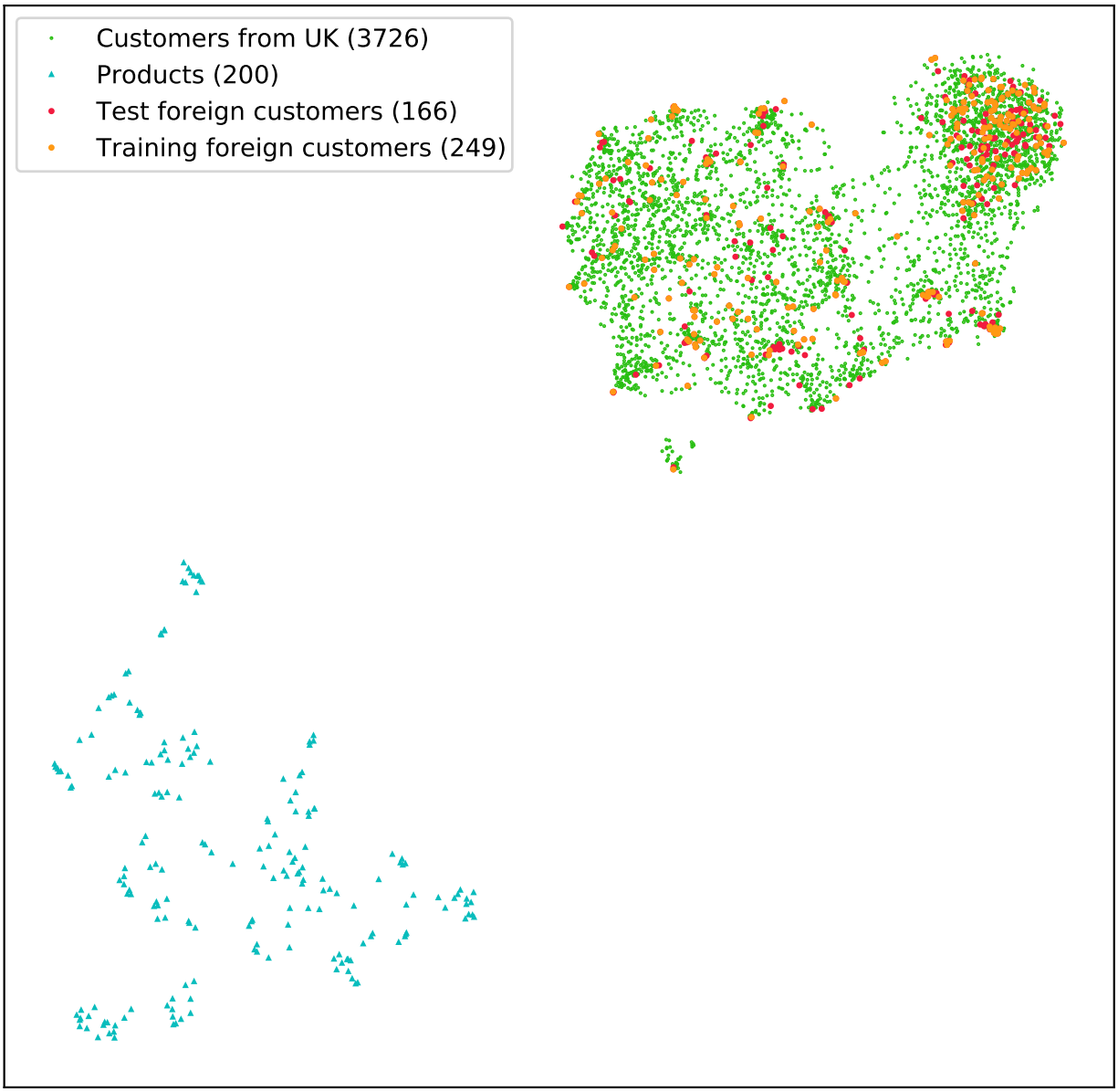


Figure A.3: Unsupervized node embeddings with node2vec on the foreign customer classification task. Markers are rendered in the order corresponding to their appearance in the legend.

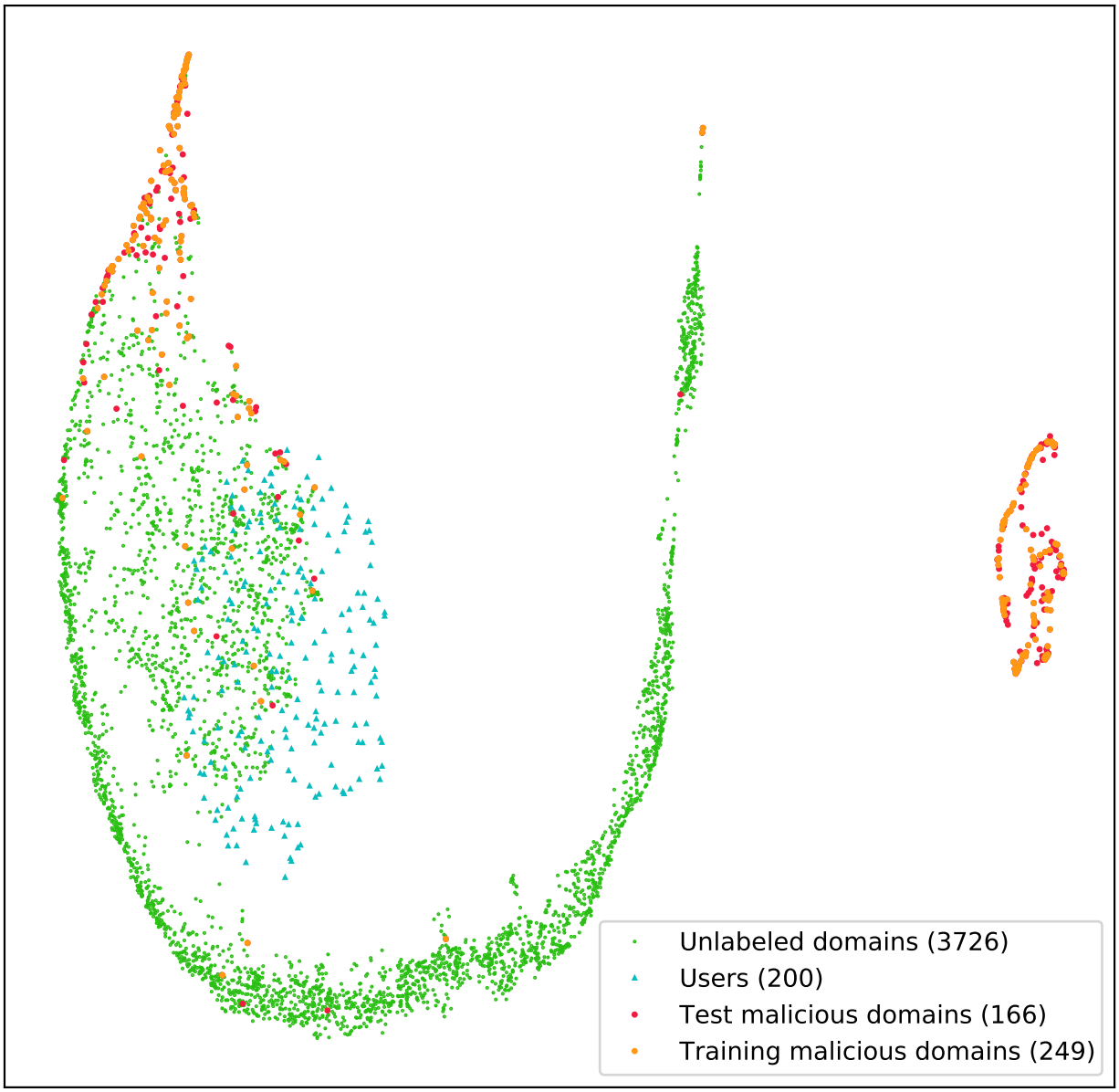


Figure A.4: Supervized node embeddings with Graph convolutional neural network (GCN) on the foreign customer classification task. Markers are rendered in the order corresponding to their appearance in the legend.

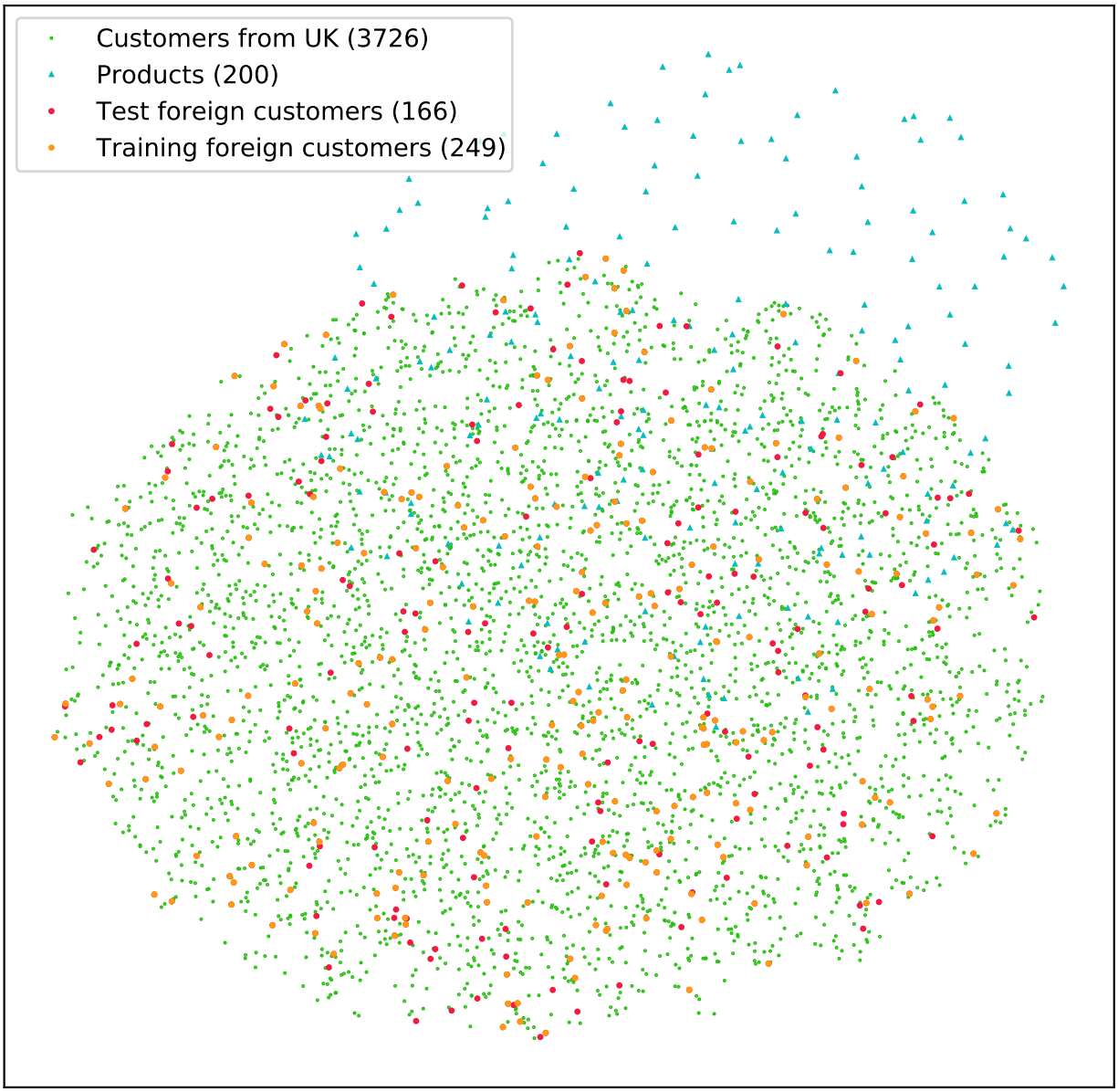


Figure A.5: Unsupervized node embeddings with TemporalNode2vec on the foreign customer classification task. Markers are rendered in the order corresponding to their appearance in the legend.

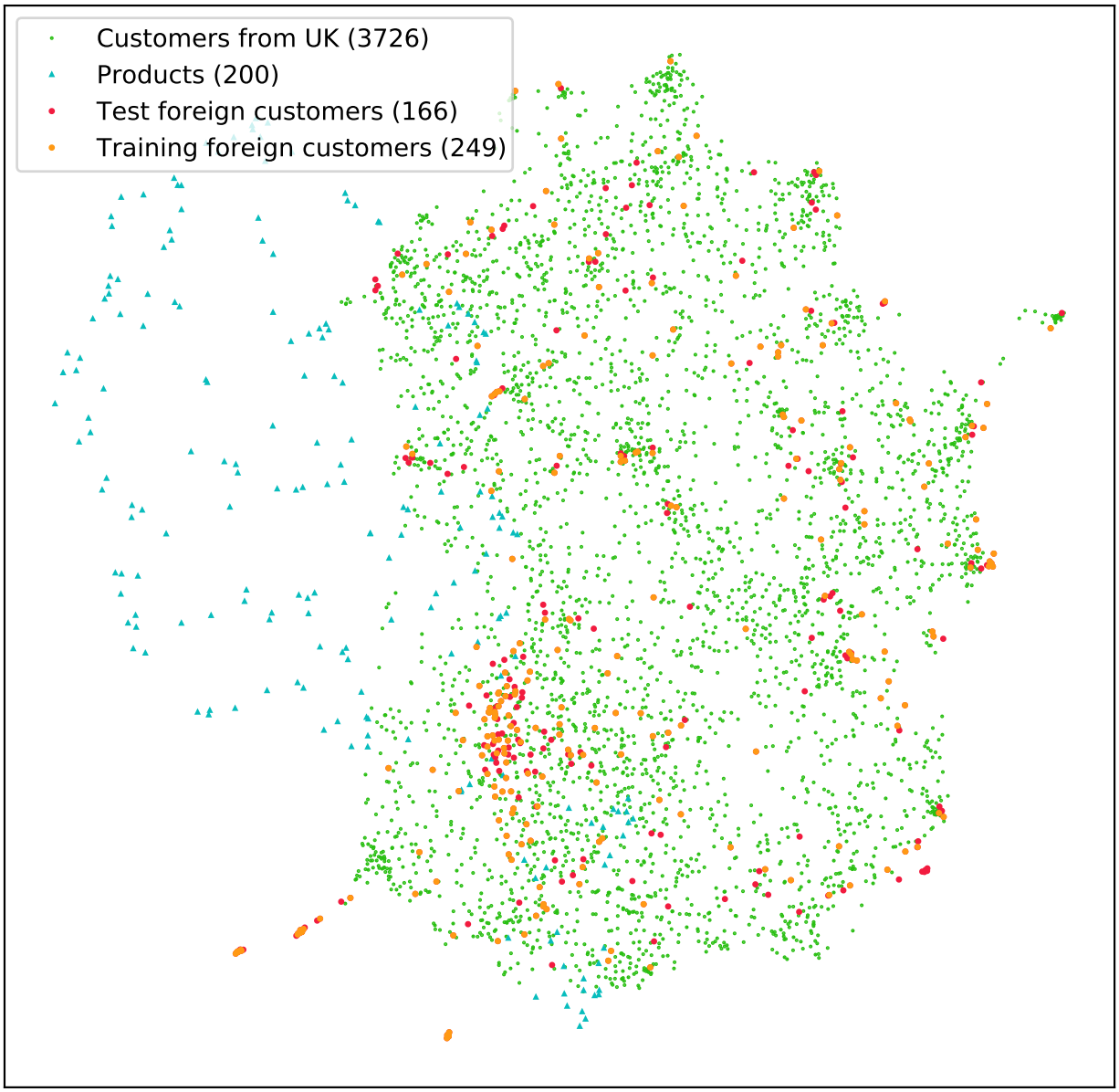


Figure A.6: Unsupervized node embeddings with CTDNE on the foreign customer classification task. Markers are rendered in the order corresponding to their appearance in the legend.

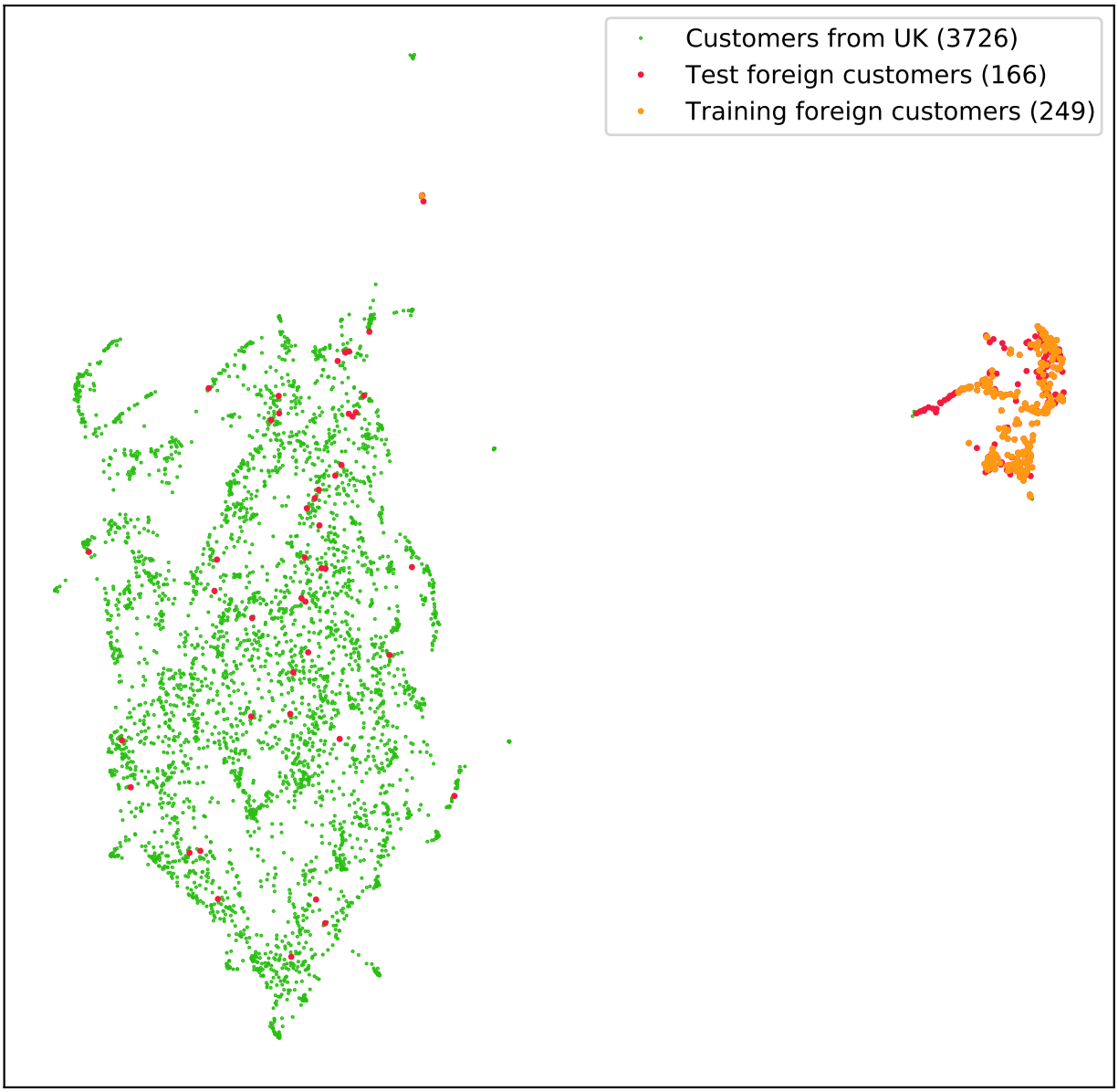


Figure A.7: Supervized node embeddings with Time-series GNN on the foreign customer classification task. Markers are rendered in the order corresponding to their appearance in the legend.

Appendix B

Acronyms

ALS Alternating Least Squares

BFS Breadth First Search

BPR Bayesian Personalized Ranking

CNN Convolutional Neural Network

CTDNE Continuous-Time Dynamic Network Embedding model

DFS Depth First Search

DGNN Dynamic Graph Neural Network

GCN Graph Convolutional Neural Network

GNN Graph Neural Network

GRU Gated Recurrent Unit

LSTM Long Short-Term Memory

NLP Natural Language Processing

RNN Recurrent Neural Network

Appendix C

Contents of enclosed CD

`thesis` ..... the directory with contents of enclosed CD
- `experiments` ..... the directory with a source code of experiments
- `tex` ..... the directory of LaTeX source codes of the thesis
- `thesis.pdf` ..... the text of the thesis in PDF format