Master's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Behavioural classification of network devices

Bc. Vojtěch Outrata

Supervisor: Ing. Martin Kopp, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Outrata Vojtěch**	Personal ID number:	**474721**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence**		

II. Master’s thesis details

Master’s thesis title in English:

Behavioural classification of network devices

Master’s thesis title in Czech:

Behaviorální klasifikace síťových zařízení

Guidelines:

The goal of the thesis is to classify devices in a computer network by behavioural patterns observed in their network communication. Device behaviour will be extracted from connection logs collected on the endpoint devices.
1. Review current state-of-the-art of both static and dynamic network device classification.
2. Based on studied literature, select a baseline and propose a novel method or adapt existing state-of-the-art method for behavioural network device classification task.
3. Implement the proposed method. Test it against the selected baseline on the dataset provided by the supervisor. Analyse the results and discuss advantages of tested methods in real computer networks.

Bibliography / sources:

Rossi, Emanuele, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. "Temporal graph networks for deep learning on dynamic graphs." arXiv preprint arXiv:2006.10637 (2020).
Anglade, Thomas, Christophe Denis, and Thierry Berthier. "A novel embedding-based framework improving the User and Entity Behav-ior Analysis." (2019).
Velickovic, Petar, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. "Graph attention networks." stat 1050 (2017): 20.

Name and workplace of master’s thesis supervisor:

Ing. Martin Kopp, Ph.D. Cisco Systems

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

Date of master’s thesis assignment: **31.01.2023** Deadline for master's thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Martin Kopp, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

Acknowledgements

I am sincerely grateful to Ing. Martin Kopp, Ph.D., for his exceptional supervision, guidance, and invaluable expertise, as well as to Ing. Jaroslav Hlaváč for his valuable consultations and advice, attention to detail, and enduring my stubbornness. Their support and mentorship have been instrumental in shaping the research and refining its outcomes.

I would also like to convey my heartfelt gratitude to my girlfriend, Veronika, and my family for their unwavering support and encouragement throughout my entire studies.

Lastly, I would like to express my deep appreciation to my friends from university who have been with me throughout the entire academic journey. Our shared experiences, and late-night sessions, sometimes for the purpose of studying, have made the university years truly memorable and enjoyable.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 24, 2023

Abstract

My thesis focuses on developing a method for the behavioral classification of network devices using state-of-the-art machine learning methods. Since computer networks have natural graph structures, the implemented methods focus primarily on the graph representation of networks. I have conducted extensive research on node classification in static and dynamic graphs and developed a new model for classifying nodes in a dynamic network, the Snapshot GNN. Leveraging deep domain knowledge, I have crafted meaningful device representations and developed new node positional features that capture the global position of a node in the graph. The proposed node positional features are scalable to large graphs, and their effect is confirmed in an ablation study. The implemented models (three baseline models and graph neural networks) are then evaluated on three real-world networks and their devices. Experiments show that the models, in general, are able to learn the diverse device classes in the networks well. The graph-based models were the best-performing models across the evaluation. The results also show that the graph-based models need frequent retraining, especially in cases where graph structure is crucial for the classification task.

Keywords: network, device, telemetry, semi-supervised classification, graph neural network, static graph, dynamic graph, node classification

Supervisor: Ing. Martin Kopp, Ph.D.

Abstrakt

Má diplomová práce se zaměřuje na vývoj metody klasifikace chování síťových zařízení pomocí nejmodernějších metod strojového učení. Vzhledem k tomu, že počítačové sítě mají přirozenou grafovou strukturu, zaměřují se implementované metody především na grafovou reprezentaci sítí. Provedl jsem rozsáhlý výzkum klasifikace uzlů ve statických a dynamických grafech a vyvinul jsem nový model pro klasifikaci uzlů v dynamické síti, Snapshot GNN. S využitím doménových znalostí jsem vytvořil reprezentace zařízení a vyvinul nový způsob zachycení globální polohy uzlu v grafu. Navržená polohová charakterizace uzlů je škálovatelná na velké grafy a její účinek byl potvrzen v ablační studii. Implementované modely (tři základní modely a grafové neuronové sítě) jsou pak vyhodnoceny na třech reálných sítích a jejích zařízeních. Experimenty ukazují, že modely jsou schopny se dobře naučit různorodé třídy zařízení v sítích. V celkovém hodnocení se jako nejlépe fungujícími modely ukázaly ty založené na grafech. Výsledky také ukazují, že modely založené na grafech potřebují časté přetrénovávání, zejména v případech, kdy je pro klasifikační úlohu klíčová struktura grafu.

Klíčová slova: síť, zařízení, telemetrie, kombinace učení s učitelem a bez učitele, grafová neuronová síť, statický graf, dynamický graf, klasifikace uzlů

Překlad názvu: Behaviorální klasifikace síťových zařízení

Contents

Figures

Tables

Chapter 1

Introduction

Computers have played a significant role in our society since the second half of the previous century. Since the breakout of the internet, the number of interconnected devices of various types has started to grow exponentially. Currently, most companies heavily rely on the internet's incredible potential. For this purpose, many companies utilize their own large private networks critical for their operation. They must be carefully administered and monitored to prevent failures in the network that would lead to business disruption. A crucial part of this process is determining the role and importance of each device in the network. These devices can range from internet-connected light bulbs to servers handling financial transactions.

However, determining the type of a device is prohibitively difficult and time-consuming to do manually. It requires deep domain and network infrastructure knowledge, and it becomes impossible to manage properly in large-scale networks. The complexity comes from frequent fluctuations of devices, new devices joining the network, short-term virtual machines, etc. These issues also cause exact automated solutions to be extremely hard and expensive to configure and maintain. As a consequence, many devices within networks end up either mislabeled or not labeled at all.

In my thesis, I approach this issue as a semi-supervised classification problem and solve it by applying state-of-the-art machine learning methods, specifically graph neural networks. I have further combined the methods with knowledge of networking to develop and extensively test multiple methods. Models from the first family of methods classify a device only based on information about the device itself. The networking data, however, has a natural graph structure, which can be leveraged in multiple ways to better capture the behaviour of a device. Therefore, models leveraging the constructed static graph structure, as well as device information, form the second family of approaches. The inferred graph structure also changes throughout the day, along with changes in the communication patterns of devices in the network. These changes lead to a representation in the form of dynamic graphs, which change the graph structure and node features over time. Consequently, the third family of approaches that I utilize is designed for node classification in dynamic graphs.

In my thesis, I present research on state-of-the-art methods in each of these three families. Based on my research, I have designed new features for capturing important device information, as well as developed and implemented various suitable models. The models are then trained and tested on real-world, large-scale networks, which include various industries and numerous distinct device types. As a consequence, the model evaluation provides valid information on the usability of the models in the real world. Cisco Systems provided the dataset from real-world networks.

Chapter 2

Methods

The general task of device classification based on network traffic has yet to be thoroughly studied. Therefore, I employ methods that have been used on data with similar structures, which fall into three families of models. The first family includes methods focusing only on information about the device itself, as depicted in Figure 2.1. I use three commonly used models from this family of methods. The first of these algorithms is the Support Vector Machine (SVM) [1], which generally tries to find the optimal hyperplane for class separation based on various kernels representing feature vector distances. The second algorithm is the Random Forest (RF) [2], which is a well-known ensemble algorithm separating the feature space by orthogonal cuts. The last algorithm is the AdaBoost [3] with base learner Decision Tree, which belongs to the family of gradient boosting approaches. These three models serve as baseline models for more advanced ones throughout the thesis.

The more advanced algorithms, leveraging the graph structure, are described in-depth in the following two sections. The first section describes the numerous methods that work with the static graph structure.

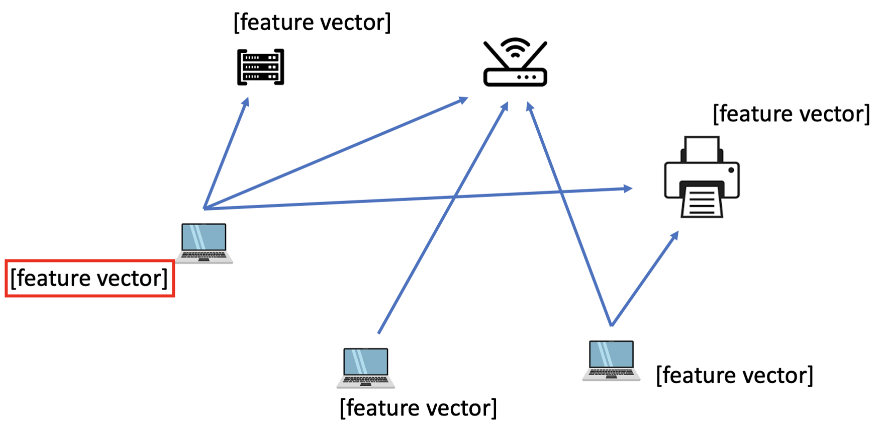


Figure 2.1: The baseline algorithms do not utilize the computer network structure and only evaluate the feature vector of a single device, highlighted by the red rectangle.

2.1 Static graph-based models

The second family of models focuses on graphs that do not evolve over time, so-called static graphs. The structure of networks naturally leads to a graph representation, which is heavily used in various manners by models described in this section. An undirected graph is represented as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes in the graph and $\mathcal{E} = \{\{i, j\} | i, j \in \mathcal{V}\}$ is the set of edges in the undirected graph G. In the constructed graph in this thesis, the nodes represent individual devices, and the edges represent connections among the devices. The detailed construction of the graph from raw data and reasons for choosing an undirected graph representation are discussed in detail in Section 4.3.

As per authors of the GraphSage architecture [4], there have been numerous methods designed for node classification on such graphs and can be generally sorted into the following three categories:

- Matrix factorization-based embedding approaches
- Node embeddings learned from random walk statistics
- Graph convolutional networks

The next three sections describe the three approaches respectively.

Matrix factorization-based node embedding approaches

This group of algorithms relies on node embeddings produced by the factorization of a matrix, which is induced by the graph structure. The information utilized by this approach is depicted in Figure 2.2.

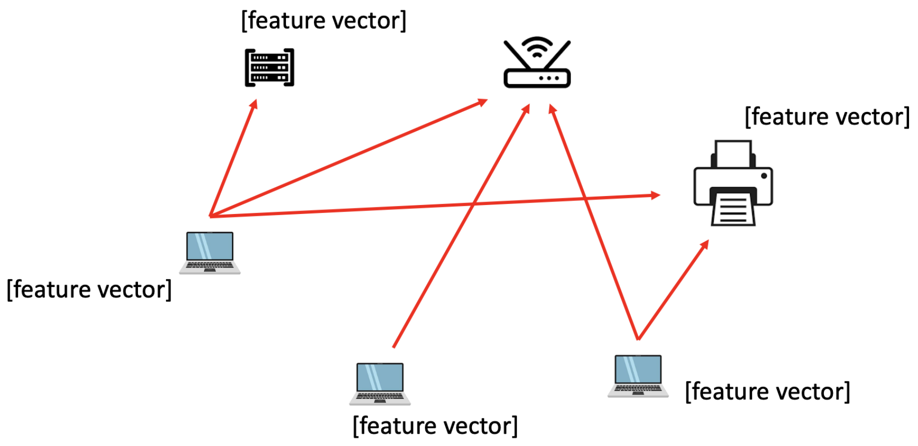


Figure 2.2: The matrix-factorization techniques utilize only the graph structure, highlighted in red. They do not use the feature vectors of devices to produce the node embeddings.

For the purpose of this thesis, it is enough to present the basic principles of this method, taken from [5].

For constructing the matrix that is factorized, first, the adjacency matrix A, degree matrix D, and Laplacian matrix L are constructed as follows:

$$
\begin{aligned}
A_{ij} &= \begin{cases} w_{ij}, & \{i,j\} \in \mathcal{E} \\ 0, & \text{otherwise} \end{cases} \\
D &= diag(d_1, \ldots, d_n) \\
L &= D - A
\end{aligned}
$$

Where d_i is the degree of node i and w_{ij} is the weight of the edge $\{i, j\}$ (equals 1 in unweighted graphs if there exists an edge between i and j, and 0 otherwise). The factorized matrix is then the normalized Laplacian matrix $\tilde{L} = D^{-1/2}LD^{-1/2}$. This matrix's factorization (eigendecomposition) produces the embedding for each node in the graph. This approach is not further used due to two main concerns. The first one is that the embeddings only contain information about the graph structure. In computer networks, the connections between devices can be enriched by important features that cannot be used by this approach. The second one is that matrix factorization is a costly operation. Consequently, the method is not feasible for large matrices induced by large graphs. Numerous methods were developed to speed up the process, such as the power iteration method [6], but the computational burden is still too high. The benchmarking [7] of clustering algorithms still shows that this method is one of the worst scaling ones and cannot be used for large graphs. Furthermore, if any new node appears in the graph (simulating the scenario of a new device connecting to the network), the node embeddings must be re-computed from scratch.

Due to the described concerns, this group of methods is not utilized in my thesis. A more recent group of methods, capturing similar information as the matrix-factorization-based approach, is described in the next section.

Node embeddings learned from random walks statistics

This approach is inspired by one of the most famous works within natural language processing, the word2vec algorithm [8]. Word2vec exploits the proximity of words within sentences to train the embedding vector of every word. This family of approaches uses the graph structure to obtain the node proximity by sampling random walks on the graph. The proximity of nodes within the random walks is then used to learn the node embeddings in a similar manner to the word2vec architecture. The random walks approach is incorporated, for example, by the DeepWalk [9], LINE [10], and node2vec [11] architectures. Since these approaches also only leverage the graph structure and not information about the node, Figure 2.2 still applies.

The node2vec architecture is the most recent and general architecture out of the listed ones and has been used to model networks previously [12], therefore, I dive deeper into its architecture. The model was designed for a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where the edges $\mathcal{E} = \{(i, j)| \ i, j \in \mathcal{V}\}$ is a set of tuples representing the oriented edges. As per authors of [11], the random walks of fixed length l are generated as follows. Given source node $u \in \mathcal{V}$, let c_i be $i-$th node in the walk starting with u. The next node is sampled according to the distribution:

$$P(c_i = x|c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} \text{ if } (v, x) \in \mathcal{E} \\ 0 \text{ otherwise} \end{cases}$$

where π_{vx} is the unnormalized transition probability between nodes v and x and Z is the normalizing constant. The walk is directed by hyperparameters p, q, controlling the tradeoff between BFS-like walks and DFS-like walks. Given that the previous traversed edge is (t, v), then the unnormalized transition probability computed as:

$$\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx}$$
$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} \text{ if } d_{tx} = 0 \\ 1 \text{ if } d_{tx} = 1 \\ \frac{1}{q} \text{ if } d_{tx} = 2 \end{cases}$$

where d_{tx} is the shortest-path distance between nodes t and x. The choice of the hyperparameters p, q then leads to the resulting embedding reflecting either the structural equivalence of nodes or the community indication (*homophily*). Once the random walks are generated, the fitting of the model follows. Let $f : \mathcal{V} \mapsto \mathbb{R}^d$ be the mapping function from node space to the embedding space with dimension d, and $N_S(u) \subset \mathcal{V}$ be the neighborhood of a node u generated through a neighborhood sampling strategy S (the random walks). Then the following objective function is optimized:

$$\max_f \sum_{u \in \mathcal{V}} \left[-\log Z_u + \sum_{n_i \in N_S(u)} f(n_i) \cdot f(u) \right]$$
$$Z_u = \sum_{v \in \mathcal{V}} \exp\left(f(u) \cdot f(v)\right)$$

The term Z_u is expensive to compute, and therefore a negative sampling approach is used. The presented objective function is then optimized through stochastic gradient descent, resulting in a learned matrix of embeddings. This approach has its advantages, but it suffers from similar issues as the matrix factorization-based approach described previously:

- The approach does not use node features.
- Once the graph changes or a new node is added, a costly training process must follow to obtain updated embeddings.
- The approach does not scale to graphs with hundreds of thousands of nodes.

Due to these disadvantages, I do not use this approach as one of the main classification methods. The architecture, however, can capture the community indication if the parameters of walks p, q are set accordingly. I use this setting of the algorithm in Section 4.3.5 to compare approaches that capture the global position of a node.

Graph convolutional networks

The disadvantages of the two previous approaches for machine learning on graphs have resulted in the introduction of a new family of approaches called graph convolutional networks. The initial paper, first introducing this concept, is accredited to authors of [13], where they introduce the GCN (Graph Convolutional Network) architecture. This innovative architecture incorporated both the features of nodes and the graph structure for propagating the node feature information to its neighborhood as depicted in Figure 2.3.

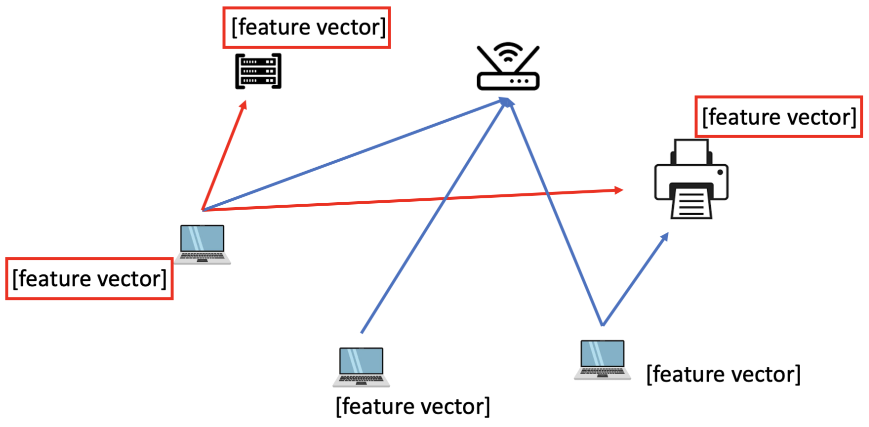


Figure 2.3: The graph convolutional neural networks utilize both the structure of the graph as well as the feature vectors of the devices. When classifying a node, the networks aggregate the feature vectors of neighbors defined by the graph structure, highlighted in red.

Let $\tilde{A} = A + I_N$ be the adjacency matrix of undirected graph $\mathcal{G}$ with self-connections I_N and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. With precalculated matrix $\hat{A} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$ the authors define two-layer GCN forward model as:

$$Z = f(X, A) = \text{softmax}\left(\hat{A}\ \text{ReLU}(\hat{A}XW^{(0)})W^{(1)}\right)$$

Where the $W^{(0)}, W^{(1)}$ are learnable matricies transforming the node feature matrix X. This approach, where the node feature information is distributed through the graph, so-called localized convolution in a graph, has sparked numerous architectures. One of the first architectures extending the GCN is the GraphSage architecture [4], defining various methods for aggregating neighbor information of the nodes in the graph. Let $\{x_i | \forall i \in \mathcal{V}\}$ be the set of node features, $W^k, \forall k = 1, \ldots, K$ be the weight matrix of the k-th layer, σ be the non-linearity, $AGGREGATE_k$, $\forall k = 1, \ldots, K$ be differentiable aggregator function of layer k, $\mathcal{N} : \mathcal{V} \mapsto 2^{\mathcal{V}}$ be the neighborhood function. The forward pass of the k-th layer of the GraphSage model of depth K is defined for all nodes $i \in \mathcal{V}$ as :

$$\begin{aligned}
&h^k_{\mathcal{N}(i)} \leftarrow AGGREGATE_k(h^{k-1}_j, \forall j \in \mathcal{N}(i)) \\
&h^k_i \leftarrow \sigma(W^k \cdot \left[h^{k-1}_i \| h^k_{\mathcal{N}(i)}\right]) \\
&h^k_i \leftarrow \frac{h^k_i}{\|h^k_i\|_2}
\end{aligned}$$

Where $h^0_i = x_i, \forall i \in \mathcal{V}$, and $[\cdot\|\cdot]$ represents concatenation. For a model with depth K, the final embedding of each node h^K_i contains aggregated information from the K-hop neighborhood of the node i. These embeddings can either directly represent class probabilities or can be further passed to a classification layer. Since the model utilizes local information only, the model scales much better than all of the previous approaches. The GraphSage has been succeeded on common benchmarks by other architectures stemming from the same schema, such as GAT [14], GATv2[15], P-GNN [16], and many more, each defining their own feature aggregating scheme. In my work, I employ one of the most recent architectures, the GATv2. Its authors identified flaws in the attention mechanism of the GAT architecture and resolved them while not worsening the scalability of the algorithm. The GAT architecture defines the forward propagation rule for node $i \in \mathcal{V}$ and one attention head as :

$$\begin{aligned}
&h'_i = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \cdot W h_j\right) \\
&\alpha_{ij} = \mathrm{softmax}_j(e(h_i, h_j)) = \frac{\exp\left(e(h_i, h_j)\right)}{\sum_{j' \in \mathcal{N}(i)} \exp\left(e(h_i, h'_j)\right)} \\
&e(h_i, h_j) = \mathrm{LeakyReLU}(a^T \cdot [W h_i \| W h_j])
\end{aligned}$$

Where $a \in \mathbb{R}^{2d}, W \in \mathbb{R}^{d' \times d}$ are learnable parameters, d is the dimension of node embeddings from the previous layer, and d' is the dimension of the resulting embedding form the forward layer. The attention mechanisms often have multiple heads in parallel as it gives more expressivity to the model. As authors of GATv2 show, the scoring function $e(\cdot,\cdot)$ is the root of the flaw of GAT architecture since the features of the two adjacent nodes are not compared against each other to obtain the attention score, as it is common in other attention mechanisms, for example in natural language processing. The GATv2 makes the following adjustment to the attention-scoring function e:

$$e(h_i, h_j) = a^T \mathrm{LeakyReLU}(W \cdot [h_i \| h_j])$$

Since the node features are first concatenated and then transformed by the learnable matrix W, the attention mechanism is far more expressive, as per authors of the GATv2 architecture. Further classification linear layer with

LeakyReLU activation layer coupled with the *softmax* layer is used for node class prediction:

$$s^i = \text{softmax}\left(\text{LeakyReLU}(W_p h'_i)\right)$$

And given the training node set $\mathcal{V}_{train} \subset \mathcal{V}$, the mean-reduced cross entropy loss is utilized for training the model's parameters. Given the label y of the training node i, the loss is computed as:

$$L = -\frac{1}{|\mathcal{V}_{train}|} \sum_{i \in \mathcal{V}_{train}} \sum_{j=1}^{k} \left[\mathbb{1}\left[j = y\right] \log(s^i_j) + (1 - \mathbb{1}\left[j = y\right]) \log(1 - s^i_j)\right]$$

The model utilizing the GATv2 aggregation scheme is further referred to as the *GNN* throughout the thesis.

In summary, the model's architecture enables learning even in tasks where the graph structure does not have to be relevant for the classification, thanks to utilizing the node features. However, the lack of information about the node's position in the global graph structure sometimes has to be compensated for, as demonstrated later in Section 4.3.5.

Also, the static nature of the underlying graph is not ensured in the networking domain. The graph structure changes throughout the day, along with the communication patterns of devices. Therefore, the following section focuses on methods that account for this change in the underlying structure.

2.2 Dynamic graph-based models

The third family of approaches focuses on node classification on data structured as timestamped interactions, so-called dynamic graphs.

As per [17], there are two main models for dynamic graphs:

- **Discrete-time dynamic graphs** (DTDG) represent dynamic graphs as sequences of static graph snapshots taken at intervals in time.
- **Continuous-time dynamic graphs** (CTDG), which are more general and represent a dynamic graph as a timestamped sequence of events, which could include node or edge addition and deletion or node and edge feature transformation.

I experimented with both approaches in my work and devoted the next two sections to describing architectures from these families. The following section introduces node classification methods leveraging the DTDG representation.

2.2.1 Snapshot architectures

The snapshot architectures deal with dynamic graphs with the help of discretization. The changing graph is captured by sampling static graph snapshots. For each snapshot, the graph structure, as well as node/edge features,

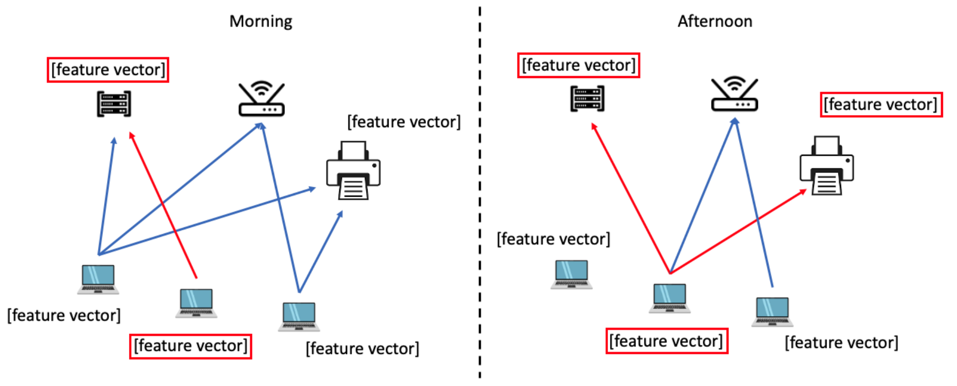


Figure 2.4: The dynamic graph-based approaches can track how the graph structure changes over time. They can account for new devices occurring in the network and detect changes in a device's behaviour within a day. The feature vectors attributed to the devices also change over time.

may be different. Formally, let $\mathbb{G} = \{\mathcal{G}^1, \ldots, \mathcal{G}^T\}$ be the set of observed attributed graph snapshots. Each snapshot $\mathcal{G}^j = (\mathcal{V}, \mathcal{E}^j)$ is undirected graph with edge set $\mathcal{E}^j$ and shared node set $\mathcal{V}$. The graph neural network model aims to learn output embeddings h_i^j for each node $i \in \mathcal{V}$ and each snapshot $\mathcal{G}^j \in \mathbb{G}$.

For example, the DySAT architecture [18] has two modules, structural and temporal attention modules. A variant of GAT architecture is utilized as a structural attention module producing node embedding for each node i and snapshot: $\{h_i^1, h_i^2, \ldots, h_i^T \; | h_i^j \in \mathbb{R}^d \}$, where d is the dimension of the node embedding. The structural embeddings are concatenated and passed through the temporal attention layer, depicted in Figure 2.5. As per [18], for final embedding z_i^j, the temporal attention only focuses on previous structural embeddings $\{h_i^k | \; k \leq j\}$, capturing the evolution of the node in the network. The temporal embedding z_i^j is then further used for a downstream task, e.g., node classification.

The primary concern regarding this approach is that it is computationally demanding in both time and space complexity. The forward pass shown in Figure 2.5 shows that during training, the entire computational graph of T graph snapshots must be stored in memory to make a prediction and back-propagate the loss. This scenario is not feasible for large graphs induced by large networks with hundreds of thousands of devices. Even the static GNNs are computationally expensive, and the snapshots multiply the computational burden by a factor of T. Other recent architectures from this family ([19], [20]) work similarly. They aggregate information across snapshots to make an inference, resulting in large computational graphs spanning across the snapshots.

However, the granularity that the snapshotting approach provides is still desired and can be achieved even for large-scale networks. Detecting a device behaving differently throughout the day may be an essential signal for

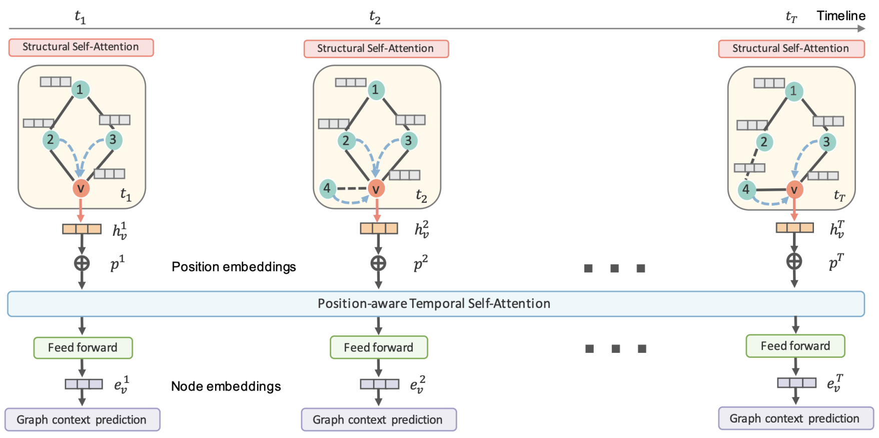


Figure 2.5: The forward pass of the DySAT [18] architecture. There are two attention modules, one for graph structure and the other for capturing temporal evolution across snapshots.

monitoring a network. The following section introduces the family of methods that do not use discretization for modeling dynamic graphs but represent the dynamic graph directly as a timestamped sequence of events.

2.2.2 Continuous-time dynamic graph-based approaches

The field of machine learning on the CTDG is very recent, and some examples of methods from this family are JODIE [21], DyRep [22], TGAT [23], and TGN [17]. The TGN (Temporal Graph Network) performs best on common benchmarks in both performance metrics and training speed. The authors of TGN showed that the rest of the cited architectures are just a particular instantiation of their framework. Therefore, I will further describe only the TGN architecture and the notation defined in the original paper. The TGN framework is based on processing the timestamped events, so-called a temporal multi-graph.

The temporal multi-graph (multiple edges between two nodes are allowed) is defined as a sequence of timestamped events $\mathcal{G} = \{x(t_1), x(t_2), \dots\}$. The event represents the addition/deletion of a node or interaction between a pair of nodes at the respective timestamps $0 \leq t_1 \leq t_2 \leq \dots$. The interaction of nodes i, j at time t is represented by a directed temporal edge $e_{ij}(t)$. During both of these types of events, the features of the node(s) involved in this event are updated. The temporal set of vertices and edges with respect to time interval T is defined as $\mathcal{V}(T) = \{i | \exists v_i(t) \in \mathcal{G}, t \in T\}$, where $v_i(t)$ is a node-wise event, and $\mathcal{E}(T) = \{(i,j) | \exists e_{ij}(t) \in \mathcal{G}, t \in T\}$. The temporal neighborhood is defined as $\mathcal{N}_i(T) = \{j | (i,j) \in \mathcal{E}(T)\}$ and the snapshot of temporal graph $\mathcal{G}$ at time t is $\mathcal{G}(t) = (\mathcal{V}([0,t]), \mathcal{E}(0,t))$ with $n(t)$ nodes. The main contribution of the TGN framework is that it can be applied to the sequence of timestamped events and produce embeddings of all the graph nodes $Z(t) = (z_1(t), \dots, z_{n(t)}(t))$ for any timestamp t. The

TGN framework is composed of core modules, each handling a different part of either keeping the model up to date with the data (**memory module, message function, message aggregator, memory updater**) or producing the temporal embeddings of nodes (**temporal embedding layer**).

Memory Module consists of vectors $s_i(t)$ for each node i the model has seen so far. The memory of a given node gets updated after every event the model has been involved in. It is supposed to represent the node's current state in condensed low-dimensional representation.

Message Function module serves the purpose of representing an event as a vector. In the case of an event $e_{ij}(t)$, two messages are computed:

$$m_i(t) = msg_s(s_i(t^-), s_j(t^-), \Delta t, e_{ij}(t))$$
$$m_j(t) = msg_d(s_j(t^-), s_i(t^-), \Delta t, e_{ij}(t))$$

where $e_{ij}(t)$ is the feature vector of the interaction, $s_i(t^-)$ is the memory vector of node i just before t and Δt is the time difference between t and timestamp of last event node i was involved in. For instance, the msg function can be an identity function or a multilayer perceptron.

Message Aggregator module is introduced due to the batch processing of interactions. Messages concerning a node in each batch are aggregated into one message to update the node's memory. The aggregation strategies may be, for instance, taking the last interaction in the batch or taking the mean of all interactions.

Memory Updater is the module used to update the node's memory vector s_i, once the messages are passed through the message aggregation function and message function:

$$s_i(t) = mem(\bar{m}_i(t), s_i(t^-))$$

where $\bar{m}_i(t)$ is the aggregated message. The mem function is usually instantiated as LSTM or GRU cell to capture the node's evolution over time.

Temporal embedding layer aggregates the node's temporal neighborhood $\mathcal{N}_i(T)$ into a temporal embedding. In the original paper introducing TGN, there are numerous embedding layers proposed. This can be, for example, the Temporal Graph Attention, first introduced in [23]. Unlike the GATv2 architecture described above, this layer computes the attention coefficients from the nodes' features, edge features connecting the nodes, and the elapsed time from the interaction. The current memory vector of the respective node represents each node's features in this aggregation layer. I skip the neighborhood aggregation's mathematical description since it is similar to the static GNN approach.

Once the temporal embeddings of all nodes are generated, they can be used for temporal link prediction by the classical approach of concatenating the embeddings of nodes involved in the potential interaction or directly for node classification. The training of this architecture for link prediction purposes is displayed and described in Figure 2.6 taken from the original paper. The

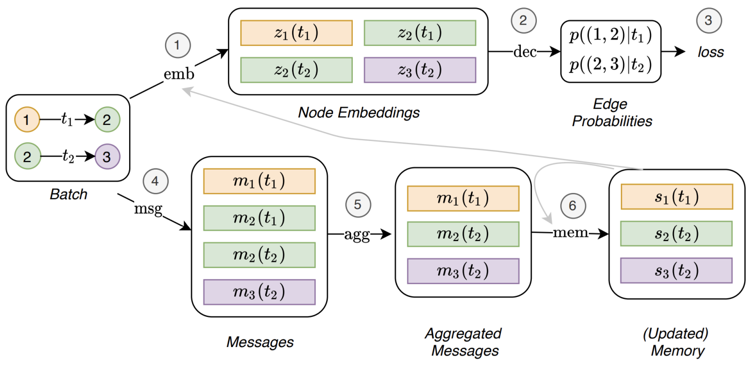


Figure 2.6: The training of TGN architecture for link prediction. The interactions are processed chronologically in batches. First, the embedding layer produces node embeddings for nodes involved in the batch interactions from the current temporal graph and memory state. Then the interactions in the batch are predicted based on the produced embeddings (with randomly sampled negative ones). The resulting loss is used to update each of the modules, and the interactions in the batch are used to update the memory of each node in the interactions.

link-prediction task is recommended for pre-training the model for further use in node-classification tasks.

The computational burden of these architectures is rather significant. The model processes the interaction events chronologically in batches, and large-scale networks have tens of millions of interactions per day. That results in very long training times, which are not rewarded with excellent results. In my experiments, the model could not learn the static device labels properly compared to other methods. One of the reasons may be that it also struggles to capture the node position in the global graph, which is valuable information for device classification tasks, as demonstrated later in this thesis. This task is in general even harder in dynamic graphs, where the graph structure is changing constantly with each batch of interactions. The architecture also evaluates a node based on the current state of the memory module, which is updated by an RNN cell. The RNN cell, however, may struggle to capture long-term information in the memory vector, which is vital for making inferences about a device type.

I have performed numerous experiments, first with the JODIE architecture, followed by extensive experiments with the TGN architecture. The experiments revealed that the models are outperformed even by the standard baseline algorithms with static features for the node classification based on static labels (the node's label does not change over time).

So far, both approaches for node classification on dynamic graphs have had significant shortcomings when applied to the networking domain. Therefore, I have developed a new method for node classification in large-scale dynamic graphs, called the *Snapshot GNN*.

2.2.3 Snapshot GNN

The model falls into the broad spectrum of snapshotting architectures and is one of the main contributions of my thesis. One of the critical attributes of this architecture is how the snapshots are generated from the raw networking data. This procedure is described in detail in Section 4.3.2. It ensures that the generated snapshots capture the most recent information with respect to the snapshot timestamp while subsampling the graph edges to lower the computational burden. This approach was inspired by experiments with the CTDG-based methods from the previous section.

The architecture also does not have a layer aggregating information across snapshots. The absence of this layer is crucial as it leads to each snapshot being evaluated in parallel. Consequently, only the computational graph of one snapshot can be kept in memory at a time, enabling this architecture to be used even on large-scale graphs scaling to hundreds of thousands of devices. Furthermore, this approach enables capturing the graph structure through node positional features (based on the graph of the respective snapshot) described in Section 4.3.4, which is one of the crucial components of my classification task.

I again use the GATv2 architecture to produce structural embeddings of nodes. These structural embeddings are then used by one final linear classification layer with a LeakyReLU activation layer paired with a *softmax* function:

$$s_i^j = \text{softmax}\left(\text{LeakyReLU}(W_p h_i^j)\right)$$

where $W_p \in \mathbb{R}^k \times d$ is the learnable matrix, k is the number of classes, and h_i^j is the structural embedding of dimension d from the GATv2 architecture applied to the snapshot $\mathcal{G}^j$. Furthermore, the cross entropy loss is employed for learning the model's parameters. Let $\mathcal{V}_{train} \subset \mathcal{V}$ be the set of training nodes, then the loss is summed across snapshots:

$$L = -\sum_{\mathcal{G}^j \in \mathbb{G}} \frac{1}{|\mathcal{V}_{train}|} \sum_{v \in \mathcal{V}_{train}} l_i^j$$

$$l_i^j(s_i^j, y) = \sum_{l=1}^{k} \left[\mathbb{1}\,[j = y] \log(s_{il}^j) + (1 - \mathbb{1}\,[j = y]) \log(1 - s_{il}^j)\right]$$

The loss function is optimized across the snapshots, ensuring the model can classify devices consistently.

The architecture is specifically suited to large-scale dynamic networks. The design decisions reflect this goal and may not be desirable for other use cases. For instance, this approach does not leverage the node evolution in time to make inferences, which is specific to the networking data. For example, the device identifier (for instance an IP address) may be used by a different device in the morning as compared to the evening. Furthermore, in classification tasks in dynamic graphs where graph structure information is not relevant, other approaches (such as the TGN) may be more suitable.

Chapter 3

Data description

This chapter describes the raw networking data I use in my thesis. It covers data collection, format, and labeling. The choice of the connection logs as the raw telemetry and the reasons for choosing this telemetry are described in the following sections.

3.1 Data acquisition

There are two main options for network telemetry with respect to where the communication is captured. The first option is capturing the communication at one of the endpoints of the communication, called endpoint connection logs. The other option is capturing the communication somewhere "along the way", often at the edge of the network. This kind of telemetry is called edge telemetry, and the acquisition of these types of telemetries is displayed in Figure 3.1.

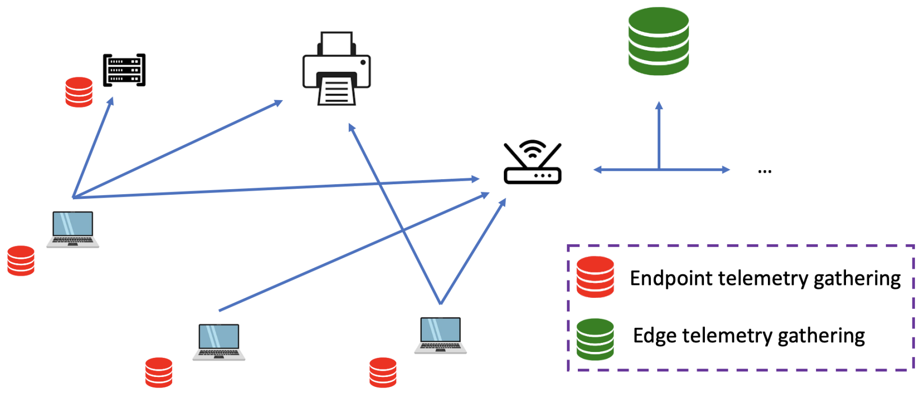


Figure 3.1: The endpoint telemetry is logged at the devices by an application indicated by the red database icon, while the edge telemetry is gathered at the border of the network, indicated by the green database icon.

Both telemetry types require different data processing and vary in the captured information about the network:

- **Edge telemetry** provides information about all the outgoing or incoming communication from the public internet. The downside is that much of the information from internal communication within the private network may not be logged.

- **Endpoint telemetry** captures both the public and private communication of a device that is being monitored. Also, the device logging the events is uniquely identified in the logs (unlike in the edge telemetry). The unique identifier partially removes the instability of IP address space that is otherwise a natural part of the networking ecosystem. I elaborate on this in Section 4.1. A disadvantage of using this telemetry is that the private network can be heavily subsampled due to the distribution of monitored devices throughout the private network. A special software, which collects the computer telemetry, has to be installed on the computers in the private network. This is commonly not the case for all or even most computers in the private network. The individual private network endpoint collector coverage has a heavy impact on the portion of the network observed with this telemetry.

It is important to note that most of the data processing methods that I present are applicable to both types of telemetries and are therefore very universal. Based on the previous analysis, it was decided that I will be working with the endpoint telemetry to get this crucial insight into the private networks. The data does not capture low-level information, such as raw packet captures, but is much more similar to the NetFlow [24] data format. Since the data is collected by a lightweight application installed on the endpoint device, additional important information about the device is also captured. This includes, for instance, the process that handles the communications or unique identification of the device. The extra information from the endpoint is used for more stable device identification and to craft additional device features in later sections.

3.2 Endpoint connection logs

As stated before, a plethora of information can be captured at the endpoint of the communication. I select the information that would be present in endpoint connection logs from any source, not only the particular data source that I am utilizing. This further means that all of the methods applied in this thesis can be applied to endpoint data of any origin. After consultations with domain experts, I have selected the following log features:

- ***deviceId*** - Uniquely identifies the monitored device.
- ***source*** - The communication information of the device that initiated the communication.
  - ***IP*** - IP address.
  - ***p*** - Port number.

deviceId	source	destination	dir	proto	pHash	ts
017...	***ip***: '192.168.80.182', ***p***: 59496	***ip***: '192.168.0.199', ***p***: 80	1	TCP	'75C ...'	165...
47f...	***ip***: '10.4.22.129' ***p***: 64270	***ip***: '72.21.91.29' ***p***: 80	1	TCP	'333 ...'	165...

Table 3.1: Connection log examples. Direction value 1 indicates that the connection was initiated by the monitored device (outgoing connection).

- ***destination*** - The communication information of the target device of the communication.
  - ***IP*** - IP address.
  - ***p*** - Port number.
- ***dir*** - Direction of the communication from the perspective of the monitored device.
- ***proto*** - Constants indicating used protocol. (TCP, UDP, ...)
- ***pHash*** - The hash of the executable handling the communication.
- ***ts*** - Timestamps of when the communication was initiated.

I further want to emphasize that most of these fields are found in any connection logs, not just the endpoint data. The additional endpoint features are the ***deviceId*** and the ***pHash*** feature. The column ***deviceId*** uniquely identifies the device from which the log originates. This is much more advantageous to IP-to-IP communication logs because an IP address may change ownership over time and does not necessarily belong to one physical device over an extended period. The device, identified by the ***deviceId***, is also tied to the information in either the ***source*** or the ***destination***, depending on the direction (***dir*** field) of the communication. If the direction is outgoing, then the information in the ***source*** is tied to the monitored device, and if it is incoming, then the information in ***destination*** is tied to the device. The other device in the communication is identified only through the IP address in either the ***destination*** or the ***source***, respectively. Table 3.1 shows an example of the communication logs.

The following section describes the other input the models need to properly learn the device types, the device labels.

3.3 Device labels

The type of devices in each network varies greatly alongside the industry the company is operating in. Some networks may belong to hospitals, resulting in healthcare equipment devices, while others may be an IT company only having laptops. Because of this variance, there is no universal template for categorizing the devices within a private network. Consequently, the companies usually design the device classes themselves, resulting in device categories of various natures. One company may want to group the devices by their geographic location while another one by the function of the device in the network.

The models should therefore be trained and evaluated on labels of various types to assess the models' usability in a real-world scenario. I selected three private networks differing in size, industry, and device grouping to train and evaluate models on. The device counts from these networks, along with their labels, are presented in Table 3.2.

customer	**label**	**device count**	**device label coverage**
Customer A	City1	57	0.27
	City2	136	
	City3	136	
	City4	102	
	City5	246	
	City6	60	
	City7	51	
	City8	36	
Customer B	Domain Controller	11	0.52
	Protect	298	
	Protect - IT	20	
	Server	69	
Customer C	Loc. A - servers	514	0.67
	Loc. A - workers	1 022	
	Loc. A Building Services	10	
	Loc. A IS	24	
	Loc. A Lab	34	
	Loc. B - servers	26	
	Loc. B - workers	22 236	
	Loc. B App Packaging	15	
	Loc. B Cardiology EEG	20	
	Loc. B Cardiology PACS	176	
	Loc. B Medical Device	27	
	Loc. B Philips Software	49	
	Loc. B Radiology	84	
	Loc. C General	92	
	Loc. C General Srvs	27	

Table 3.2: Device labels for the three selected customers.

Customer A has its devices grouped by geographic location, Customer B by device function, and Customer C has a combination of both. The labels of these three customers already show that the device classes differ substantially based on the individual private network. The administrators of the private network not only decide how to group the devices but are also responsible for maintaining the assignments manually. This is a tedious and error-prone approach, resulting in two significant burdens. First, some of the devices may

new label	**merged labels**
Loc. A - servers	Loc. A Servers Audited Loc. A Srvs Protected
Loc. A - workers	Loc. A Protected with AV Wrks Loc. A Protected Wrks Loc. A Wrks Audit
Loc. B - workers	Loc. B General Wrks Loc. B Compromised Wrks Loc. B General Wrks Update
Loc. B - servers	Loc. B Servers Loc. B Srvs Fully Protected

Table 3.3: The difference between some classes is only the mode of the application collecting the logs. Since this does not affect the device's behaviour on the network, these labels are merged.

be assigned to the wrong class. The other one is that many devices within the private networks are not classified at all. The ratio of labeled devices within the network is displayed in Table 3.2 by the column **device label coverage**. The values further support the semi-supervised learning scenario, in which the models can extrapolate the learned knowledge to the rest of the devices in the network. The models learn on the labeled devices represented by the **device count** column. The column presents the number of devices within each class and shows a severe class imbalance for Customer B and Customer C. For both customers, special measures had to be taken to train and adequately evaluate the models. Due to the class imbalance issue and the high number of classes for Customer C, one of the measures is merging some of the labels. Several Customer C classes were merged, as presented in Table 3.3.

The reason for merging these labels is that the name suggests that the only difference is in the running mode of the application collecting logs on the computer. As this does not translate to any difference in the communication patterns of the devices, some of the labels were merged together. The third customer also has a small number of device categories with fewer than ten active devices, which were excluded from the main two experiments due to the scale of the network (more than 65000 devices).

Chapter 4

Data preparation and feature engineering

For the models to learn to classify devices, relevant information must be extracted from the raw data and attributed to the device. This chapter focuses on raw network telemetry preprocessing and feature engineering. To correctly attribute relevant information to a concrete device, the device must be reliably identified throughout the telemetry (Section 4.1). Once the device is consistently identified throughout the connection logs, features can be crafted and aggregated into the device representation (Section 4.2). The last part of this section (Section 4.3) describes graph construction for the graph-based models, presents detailed graph analysis of the constructed graphs, and introduces novel positional node features.

4.1 Device identification

In each connection log, two devices are to be identified, the source and the destination. Depending on the direction of the communication, one of these can be uniquely identified by the ***deviceId*** string. This identifier is time-invariant, and each connection log has this unique string of the device logged. However, the other end of the logged communication is identified only via an IP address. This device is often from the same private network and also has a ***deviceId*** identifier assigned. The implemented algorithm aims to replace the IP address with the ***deviceId*** identifier, where possible. Consequentially, more information can be attributed to the same device, resulting in a more accurate device representation.

The following sections cover the protocols responsible for the underlying issue with device identification by an IP address and then the algorithm partially solving this issue.

4.1.1 IP address space

This section is focused primarily on the IPv4 protocol since the IPv4 protocol is prevalent in the data, and the IPv6 protocol was explicitly designed to avoid the covered issues.

Internet Protocol is a standard defined in the early days of the internet to standardize routing on the whole internet. The IPv4 protocol uses 32-bit address space separated into blocks for various purposes such as multicast, broadcast, loopback, etc. One of the most crucial block separations is the public and private IP address space, which was introduced in [25]:

- **Public IP address** identifies the device uniquely to the entire internet. Devices with an assigned public IP address can communicate over the public internet.
- **Private IP address** identifies the device uniquely to the private network only. Devices with an assigned private IP address can directly communicate over the private network only. Private IP ranges are 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16.

The IPv4 addresses are only 32 bits long and the address space accounts for $2^{32} = 4,294,967,296$ unique addresses only. It was already becoming evident decades ago that this is not enough to uniquely identify every single device connected to the internet across the globe. It led to several attempts at solving the issue, out of which the most commonly adapted and used solution is the Network Address Translation (NAT) protocol.

4.1.2 Network Address Translation protocol

This protocol was designed to mitigate the IP address exhaustion problem. The diagram in Figure 4.1, taken from [26], shows a simplified example of how NAT works.

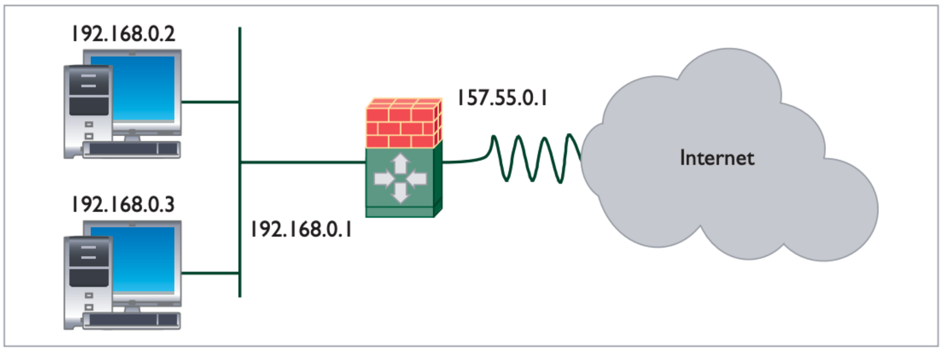


Figure 4.1: Example of how NAT protocols works. The computers have their private IP addresses for communication on the private network. The router translates the private IP address into a public one for communicating outside the private network.

As shown in Figure 4.1 and described in [26], the NAT protocol solves the IP address exhaustion problem in the following manner. When the computer with private IP address ***192.168.0.2*** wants to communicate to the public internet, it first sends the packet to the router with private IP address ***191.168.0.1***. The router then translates the source IP address in the packet to its public IP address ***157.55.0.1*** and sends it to the destination on the public internet. During this step, the router saves the translation table to keep the information about the IP address translation procedure. Once the router obtains a packet from the outside targeting IP address ***157.55.0.1***, it looks in the translation table and sends it to the appropriate device on the private network.

In general, devices connected to the private network have only been assigned a private IP address. If they want to communicate outside the private network, the router translates their private IP address into a public one. The router usually has a single public IP address or pool of public IP addresses provided by the ISP (Internet Service Provider). There are multiple versions of this protocol, such as the NAPT (Network Address Port Translation), which enables many devices on the private network to communicate with the public internet via a single public IP address. Consequently, multiple devices may be hidden behind a single (usually public) IP address in the network data. These devices cannot be distinguished from outside the private network behind the router employing NAT/NAPT protocols. The problem is usually even more severe as this single IP address may serve an entirely different pool of devices over time.

This is a typical scenario employed within most of the networking infrastructure worldwide. The last mechanism left to describe is the assignment of private IP addresses to the devices in the private network.

4.1.3 Dynamic Host Configuration Protocol

The DHCP (Dynamic Host Configuration Protocol) is a standard protocol introduced in [27]. This network management protocol is designed to assign IP addresses to devices connected to the network automatically. The inner implementation details of the DHCP are not crucial for my use case. However, the allocating mechanism of IP addresses is.

Depending on the implementation of the protocol, these mechanisms may be used:

- **Dynamic allocation** - The network administrator assigns a pool of IP addresses to the DHCP server. Each time a device is connected to LAN, it requests an IP address from the DHCP server (usually a router). The server is only lending these IP addresses for a given period of time and then reallocating them if the device does not renew them. The server allocates the IP addresses from an IP address pool assigned to the router by the network administrator or ISP.

- **Automatic allocation** - The DHCP server again assigns an IP address from the IP address pool. However, the server keeps a table of the history of IP address allocations and preferentially assigns the IP address the client previously had.

- **Manual allocation** - The network administrator manually maps a unique device identifier to an IP address, such as a MAC address.

The dynamic and automatic allocation mechanisms cause the most severe device identification issues. If these protocols are used, there is no guarantee that the device will obtain an identical private IP address, even if it repeatedly connects to the same private network. An additional issue is that after the leasing time of the allocated IP address is over, the device must request a new one, and it does not necessarily have to be the one previously owned. The leasing time also can be quite low, leading to frequent changes of IP addresses.

However, the information in the raw endpoint data can be used to obtain a partial solution to the problem caused by the instability of the IP address.

4.1.4 Device matching

This section describes an algorithm that partially solves these issues with the information contained in connection logs. To describe the basis of the algorithm, I present an **artificial sample of connection logs** with extracted communication devices.

Table 4.1 represents four communication flows logged by two devices ***id1*** and ***id2***. The value 1 of the **dir** field represents outgoing communication from the perspective of the monitored device. The **source** field contains the information about the device identified with the ***deviceId*** string as described in Section 3. The information about the currently assigned IP address to each device can be obtained and used to translate an IP address of the other device (in this case, the **destination**) into the ***deviceId*** identifier. In the case from Table 4.1, ***id1*** is currently assigned an IP address ***10.85.106.166*** as can be seen in the first and last sample. In the third connection log, ***id2*** communicates to this IP address as displayed in the **destination** field. Since the device this IP address currently belongs to is known, it can be translated into the ***id1*** resulting in the enriched connection logs in Table 4.2.

In general, the algorithm can be summarized in the following steps:

1. Collect sequence of tuples (***deviceId***, ***ts***) for each IP address tied to some ***deviceId***.

2. Create a mapping that maps the IP address to the appropriate ***deviceId*** for each timestamp from the collected sequence for each IP address. This is done by tracking the changes of ownership of each IP.

3. Apply the obtained mapping to the connection logs dataset, replacing the device's IP address on the other side of the connection with the ***deviceId***, where possible.

The success of this procedure depends on many factors, such as the portion of devices with the application installed or private IP address pools assigned to DHCP servers. In cases where the IP address is not translated, I model the IP address as a single device.

The translation must be done cautiously, as a single network usually contains multiple overlapping private networks. Therefore a single private IP address can be assigned to multiple devices simultaneously. The implementation must account for this possibility to avoid the misidentification of devices

device 1	source	destination	device 2	dir	ts
id1	***ip***: '10.85.106.166' ***p***: 53975	***ip***: '10.85.102.148', ***p***: 8530	'10.85.102.148'	1	1
id2	***ip***: '10.83.100.9', ***p***: 61008	***ip***: '10.2.2.2', ***p***: 433	'10.2.2.2'	1	2
id2	***ip***: '10.83.100.9', ***p***: 59496	***ip***: '10.85.106.166', ***p***: 8080	'10.85.106.166'	1	3
id1	***ip***: '10.85.106.166', ***p***: 8080	***ip***: '10.91.101.128', ***p***: 55208	'10.91.101.128'	1	4

Table 4.1: Artificial connection logs before the device matching procedure, **device 2** is always identified only by the IP address in the ***destination*** field.

device 1	source	destination	device 2	dir	ts
id1	***ip***: '10.85.106.166', ***p***: 53975	***ip***: '10.85.102.148', ***p***: 8530	'10.85.102.148'	1	1
id2	***ip***: '10.83.100.9', ***p***: 61008	***ip***: '10.2.2.2', ***p***: 433	'10.2.2.2'	1	2
id2	***ip***: '10.83.100.9', ***p***: 59496	***ip***: '10.85.106.166', ***p***: 8080	***id1***	1	3
id1	***ip***: '10.85.106.166', ***p***: 8080	***ip***: '10.91.101.128', ***p***: 55208	'10.91.101.128'	1	4

Table 4.2: The device matching procedure enriched the artificial log sample. The information from the **source** field is utilized in order to match the IP address of the **device 2** to the corresponding ***deviceId*** of the device currently using this address.

as much as possible. For implementation details, please refer to the provided code.

Once the device is consistently identified throughout the collected networking logs, aggregating information about the device follows. The next sections cover the feature engineering process on the connection logs and how the graph structure is constructed from the data.

4.2 Feature engineering

This section presents a feature engineering process aggregating various information for all the devices in the private network. In total, there are 122 features for each device: 60 port features (Section 4.2.2), 34 time features (Section 4.2.3), 25 networking features (Section 4.2.4), and three hash features (Section 4.2.5). Some of the formulated features require supporting data analysis provided in the respective section. This analysis relies upon the **set of 8 private networks**, which were selected to cover various types and scales of networks. To ensure variance in the underlying data, the selected customers operate in various industries such as healthcare, transportation, etc., and scale from a couple of thousand devices to hundreds of thousands of devices in a single network. The selection includes five networks, which are only used for data analysis, referred to as Network 1 through Network 5, and the three networks used for device classification experiments, referred to as Customer A, Customer B, and Customer C. The next section shortly introduces the notation that is necessary in order to manipulate the connection logs.

4.2.1 Connection logs notation

For comfortable manipulation with the connection logs, I define mathematical notation in this section. I refer to the dataset of connection logs as D. Each connection log is represented as an ordered tuple:

$$\begin{aligned} N &= \text{set of all devices in the network} \\ D &= \{(i, j, f) |\ i, j \in N; f \in \mathbb{R}^m\} \end{aligned}$$

Where $\boldsymbol{i}$ is the source of the connection, $\boldsymbol{j}$ is the destination of this connection, and f represents features associated with this connection log. This feature vector contains the timestamp feature ***ts***, source port ***port_i***, destination port ***port_j***, direction ***dir***, and protocol ***proto***. Some of the following sections create additional features of these logs or aggregate log features into device features. The notation for assigning value X to feature Y of feature vector f is:

$$f_Y = X$$

Moreover, the following definitions are utilized in the following sections. Given a device m:

$$\begin{aligned} In_m &= \{(i, m, f) |\ (i, m, f) \in D\} \\ Out_m &= \{(m, j, f) |\ (m, j, f) \in D\} \end{aligned}$$

In_m and Out_m indicate sets of logs where device $\boldsymbol{m}$ is the target and source respectively.

4.2.2 Port features

Port numbers are arguably one of the most potent sources of information characterizing a device's behaviour on a network, proven to work well with machine learning models, for example, in IoT devices classification in [28]. The first part of the analysis focuses on the possible port types that can be used. Figure 4.2 displays the available port types.

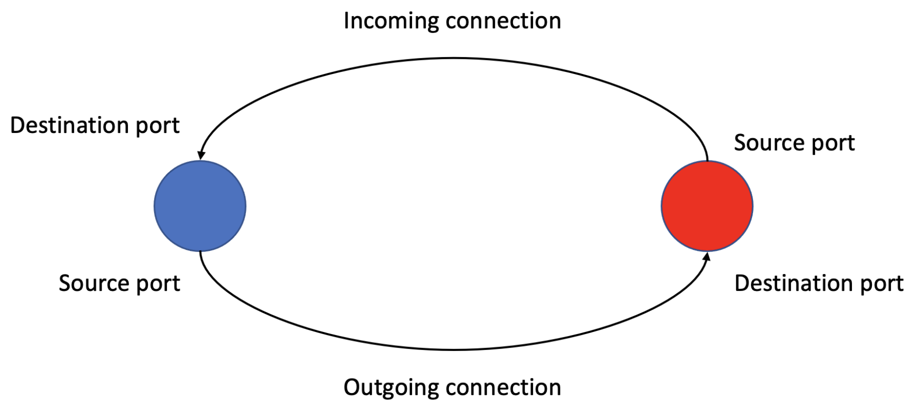


Figure 4.2: From the perspective of the blue device's communication, there are four port types to consider. Generally, the destination ports of both outgoing and incoming flows carry more meaningful information.

The most important information that can be recovered from the port numbers is:

- Services running on the device.
- Services the device is accessing.

The first type of behaviour is captured by the **destination ports of incoming connections**, and the latter is represented by the **destination ports of outgoing connections**. I designed the port features to capture both separately because the number of outgoing logs dominates all the datasets. If the port statistics were collected altogether, ignoring the connection type, one of these types (usually the destination ports of outgoing connections) would dominate the statistics, and the information contained in the other important port type would get lost.

A feature design decision regarding which services and their ports to recognize as a feature naturally follows as the next part of the feature engineering process. As described in [29], the most important port numbers, ranging from 0-65535, are ports below the 1023 threshold. These ports are often called **reserved** ports since these port numbers are reserved for typical TCP/UDP applications and fall into the category of well-known ports.

Therefore the main focus of services recognized as features is on the ports under this threshold.

Table 4.3 presents a list of detected services. These services are recognized by the ports listed next to them in the table. The port numbers of these services are registered and handled by the Internet Assigned Numbers Authority (IANA) with the procedure introduced in [30].

service	**ports**
FTP	20,21
SSH	22
Telnet	23,992
SMTP	25
DNS	53
DHCP	67,68
HTTP	80, 8080, 8008, 8081
NetBios	135-140
BGP	179
LDAP	389
HTTPS	443
LDAP secure	636
FTP secure	989,990
SMB	445
Kerberos	88
SNMP	161
NTP	123
IPP	631
Certificate Management Protocol	829
ISAKMP	500
Sun RPC	111
RLZ DBase	635
webservice unassigned	81
SNPP	444
Multicast DNS	5353
SSDP	1900

Table 4.3: Detected services with their ports.

Most of these are well-known and established services except the "webservice unassigned" representing port 81, which is prevalent in the networking data but has no assigned service. They were either handpicked based on domain knowledge and consultation with domain experts or added after analysis of prevalent ports in the data. The port analysis is presented in Table 4.4. For each of the selected networks, the top 5 prevalent target ports (of either outgoing or incoming communication) in the communication logs are presented along with a detected service by the port selection presented above.

Based on the collected port statistics, I have included additional services and ports above the **reserved** ports threshold into the selected list in Table

network	most common target ports	port prevalence	service
Network 1	443	0.639	HTTPS
	80	0.144	HTTP
	389	0.072	LDAP
	53	0.018	DNS
	135	0.016	NetBios
Network 2	443	0.323	HTTPS
	80	0.248	HTTP
	8082	0.131	HTTP
	389	0.062	LDAP
	8080	0.032	HTTP
Network 3	443	0.505	HTTPS
	80	0.298	HTTP
	389	0.046	LDAP
	49152	0.032	Undetected
	5353	0.017	Multicast DNS
Network 4	8080	0.511	HTTP
	443	0.234	HTTPS
	389	0.048	LDAP
	80	0.044	HTTP
	5353	0.030	Multicast DNS
Network 5	8081	0.555	HTTP
	443	0.185	HTTPS
	80	0.088	HTTP
	8080	0.037	HTTP
	389	0.028	LDAP
Customer A	443	0.346	HTTPS
	80	0.302	HTTP
	8080	0.233	HTTP
	389	0.026	LDAP
	5353	0.015	Multicast DNS
Customer B	443	0.419	HTTPS
	80	0.410	HTTP
	389	0.047	LDAP
	135	0.028	NetBios
	49155	0.027	Undetected
Customer C	443	0.361	HTTPS
	9090	0.261	Undetected
	389	0.102	LDAP
	6060	0.074	Undetected
	80	0.072	HTTP

Table 4.4: Networks' port statistics.

4.3. The additional ports include the known alternative ports for HTTP: 8080, 8008, 8080, 8082, and two additional services noticeably prevalent for most customers: Multicast DNS (Domain Name System) and SSDP (Simple Service Discovery Protocol). The ports with "Undetected" service in Table 4.4 are not included in the feature space because no known stable services are running on these ports, and they are not prevalent in the rest of the networks.

The last part of this section focuses on creating device features based on the presented list of services. As described at the start of this section, each device is represented by two sets of features with respect to ports, which are computed based on features of individual connection logs. For each connection log, I define port features based on the **destination port of the connection** in the following manner. I denote $S = \{(service, ports) | \ ports = \{i | \ i \in \mathbb{N}\}\}$ as the set of services along with their ports as defined in Table 4.3. With the dataset of connection logs denoted D, the log features are encoded in a one-hot encoding fashion:

$$f_{service} = \begin{cases} 1 & \text{if } f_{port_j} \in ports \\ 0 & \text{if } f_{port_j} \notin ports \end{cases} \quad \forall (service, ports) \in S, \ \forall (i, j, f) \in D$$

Additionally, to the services listed in Table 4.3, I create three similar features representing undetected ports in the range 0−1023, 1024−10 000, and ports above 10 000, respectively. The **communication flow port features** can then be aggregated into **device port features** for each respective device.

Given a device m, the computation of the device's feature for $(service, ports) \in S$ of outgoing connections follows as:

$$m_{outgoing\_service} = \begin{cases} 0 & \text{if } |Out_m| = 0 \\ \frac{1}{|Out_m|} \sum_{(m,j,f) \in Out_m} f_{service} & \text{if } |Out_m| > 0 \end{cases}$$

The features represent mean operation over the one-hot encoded features of the communication flows. The device's port features of incoming connections are computed similarly:

$$m_{incoming\_service} = \begin{cases} 0 & \text{if } |In_m| = 0 \\ \frac{1}{|In_m|} \sum_{(i,m,f) \in In_m} f_{service} & \text{if } |In_m| > 0 \end{cases}$$

Furthermore, I designed two additional node features representing the number of unique ports the device is communicating to and the number of unique ports that are being accessed by other devices:

$$m_{outgoing\_unique} = |\{f_{port_j} | (m, j, f) \in Out_m\}|$$
$$m_{incoming\_unique} = |\{f_{port_m} | (i, m, f) \in In_m\}|$$

The port behaviour of the device is captured in the presented features. The following section focuses on another aspect of device behaviour, introducing features that capture when the device is active and its communication frequency.

4.2.3 Time features

I employ two types of time features, absolute time features and features capturing communication frequency of a device.

The first type of features, the absolute time features, captures when the device communicates during the day, for how long it was active, etc. To show that these features add value and information to the device's representation and that they are meaningful to each customer, I first provide a supporting analysis. Figure 4.3 displays the distribution of communication logs for each analyzed/selected network. The plots show a significant change in the communication log distribution with respect to the time of the day.

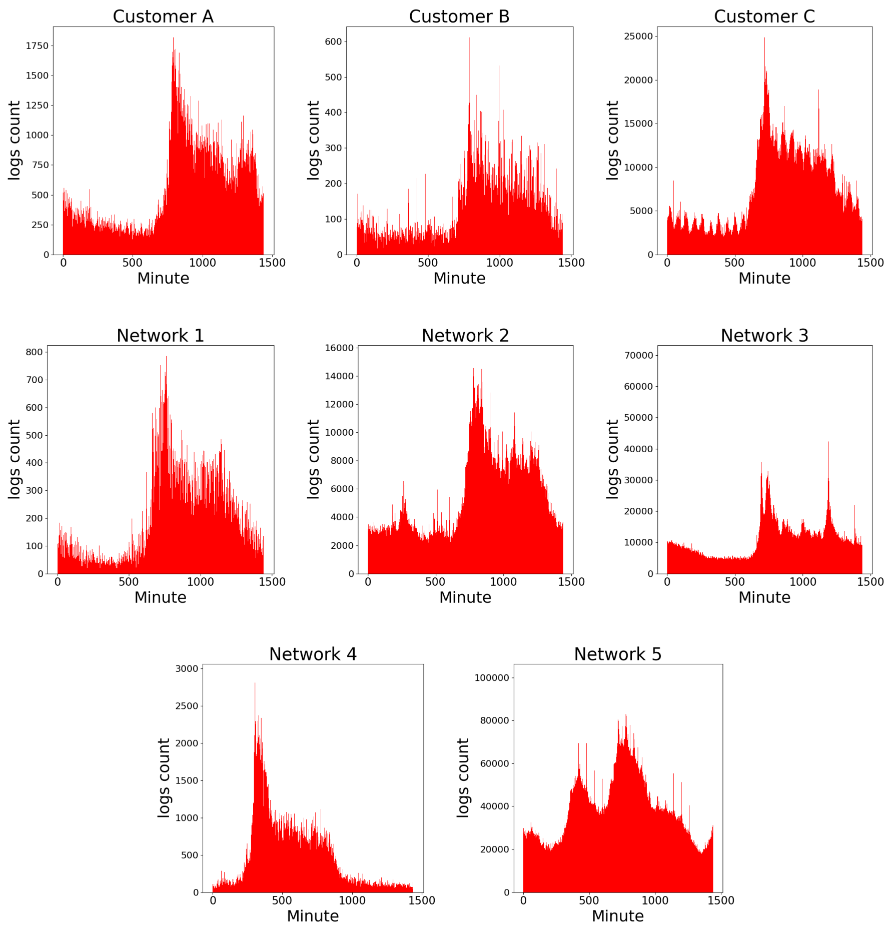


Figure 4.3: Communication logs distribution in a single day within each network. The resolution for communication logs count aggregation is one minute, resulting in 1440 minutes in a day.

For most customers, there is a spike in the figure indicating the start of the working day followed by an increased activity for 8-10 hours as expected. The working hours of Network 5 are not as clearly distinguished, which may be due to the fact that this network is large relative to other analyzed networks and may have branches operating in various time zones leading to working hours being shifted accordingly. The pattern emerging in these plots supports creating features capturing the device's active periods. The first set of features captures the distribution of device activity throughout the day, which is binned into 24 slots representing each respective hour. The features are computed similarly as port features in the previous section. First, the features of each communication log are computed followed by aggregation over the device's communication logs. With the set of hours in the day denoted H, the creation of the time features then follows:

$$f_{hour_i} = \begin{cases} 1 & \text{if } f_{ts} \in hour_i \\ 0 & \text{if } f_{ts} \notin hour_i \end{cases} \quad \forall hour_i \in H, \ \forall (i,j,f) \in D$$

This again creates one-hot encoding as the communication flow time feature, which is then aggregated to create a device feature. Again, given a device m and notation defined in Section 4.2.1:

$$m_{hour_i} = \frac{1}{|In_m \cup Out_m|} \sum_{(i,j,f) \in In_m \cup Out_m} f_{hour_i}, \ i = 0, \dots, 23$$

Two additional time features representing the number of hours in which the device was active and the total amount of time it was active for are computed as:

$$\begin{aligned} m_{hours\_active} &= \sum_{hour_i \in H} \lceil m_{hour_i} \rceil \\ m_{time\_active} &= \max_{(i,j,f) \in In_m \cup Out_m} f_{ts} - \min_{(i,j,f) \in In_m \cup Out_m} f_{ts} \end{aligned}$$

The two types of features conclude the first type of time features.

The second type of time feature computes the device's representation with respect to the device's communication frequency. First, an ordered sequence of communication timestamps is collected for each device:

$$t = (ts_i)_{i=1}^{|In_m \cup Out_m|}, \ ts_1 \leq ts_2 \leq \dots \leq ts_{|In_m \cup Out_m|}$$

Followed by collecting time differences between each consecutive pair of timestamps:

$$X = \{ts_i - ts_{i-1} | \ i = 2, \dots, |In_m \cup Out_m|\}$$

To describe the collected distribution, the following features over this set X are computed: **mean, variance, quantiles (0.25, 0.5, 0.75), skewness, kurtosis**. I included the last two features, skewness and kurtosis, based on their success in [31] as time series features. The skewness and kurtosis features are defined as:

$$\begin{aligned} m_{skewness} &= \frac{\sum_{x_i \in X} (x_i - \overline{x})^3 / |X|}{s^3} \\ m_{kurtosis} &= \frac{\sum_{x_i \in X} (x_i - \overline{x})^4 / |X|}{s^4} \end{aligned}$$

Where s stands for standard deviation, and $\overline{x}$ stands for mean. Skewness measures the asymmetry of distribution, while kurtosis measures how heavy-tailed the distribution is. The following section focuses on additional networking features that can be mined from the connection logs.

4.2.4 Networking features

The set of features covered in this section captures information about a device from the networking perspective. Since an insight into the customer's private network is crucial for the device classification, some of these features are computed twice, first for all communication flows the device is part of, and then again for private communication only.

First, as in previous sections, I define the features of each communication flow in the identical one-hot encoding manner as in previous sections and results in the following communication flow features: f_{TCP}, f_{UDP}, and f_{priv_comm}. The first two features represent the two protocols of the transport layer, and the third feature indicates whether the communication is over the private network. The following device networking features are collected:

- $m_{count} = |In_m \cup Out_m|$
- $m_{n\_out} = |Out_m|$
- $m_{n\_in} = |In_m|$
- $m_{dir\_mean} = \frac{|Out_m|}{|In_m \cup Out_m|}$
- $m_{TCP\_mean} = \frac{1}{|In_m \cup Out_m|} \sum_{(i,j,f) \in In_m \cup Out_m} f_{TCP}$
- $m_{UDP\_mean} = \frac{1}{|In_m \cup Out_m|} \sum_{(i,j,f) \in In_m \cup Out_m} f_{UDP}$
- $m_{priv\_comm} = \frac{1}{|In_m \cup Out_m|} \sum_{(i,j,f) \in In_m \cup Out_m} f_{priv\_comm}$
- $m_{n\_neighbours} = |\{j|\ (m,j,f) \in Out_m\} \cup \{i|\ (i,m,f) \in In_m\}|$
- $m_{out\_neighbours} = |\{j|\ (m,j,f) \in Out_m\}|$
- $m_{in\_neighbours} = |\{i|\ (i,m,f) \in In_m\}|$

The last three features capture the number of devices the device communicates with in total and in both directions.

4.2.5 Hash features

This last section covers the information that can be retrieved from the so far neglected field in connection logs, the hash field ***pHash***, which represents a hash of the program handling the communication at the monitored device.

The features capture the relative frequency of the logged hash. First, the unique hashes are ordered into a sequence by their count:

$$H = \{f_h | (i, j, f) \in D\}$$
$$count(h) = |\{(i, j, f)| \ (i, j, f) \in D, \ f_h = h\}|, \ h \in H$$

I collect three sets of hashes from a sequence ordered by this function in descending order: the top 10%, the set of hashes in the range 10% − 50%, and the rest. These three sets are then used for one-hot encoded features of the communication flows, which are then aggregated in the same fashion as it was previously done for port and time features.

4.3 Graph representation

This section shows how I build and use the communication graph structure in my thesis. I first describe how the graph is created from the communication logs (Section 4.3.1 and Section 4.3.2). Then I analyze the graph structure by standard graph analysis tools (Section 4.3.3). I then use the inferred knowledge from graph analysis to develop new positional node features (Section 4.3.4).

4.3.1 Graph construction

In order to leverage the graph structure, it must be first constructed from the raw data. The individual graphs are constructed from private communication only. There are two main reasons for this choice. Firstly, the previously described noise in device identification caused by the IP address assigning mechanisms propagates even more in the public domain. Secondly, reducing the communication to private only significantly reduces the number of edges and nodes in the resulting graphs, leading to much faster training and inference times.

The graph $G = (\mathcal{V}, \ \mathcal{E})$ is constructed from a day's worth of connection logs D in the following manner:

$$D_{priv} = \{(i, j, f)| \ f_{priv\_comm} = 1, (i, j, f) \in D\}$$
$$\mathcal{V} = \bigcup_{(i,j,f) \in D_{priv}} \{i, j\}$$
$$\mathcal{E} = \{\{i, j\}| \ (i, j, f) \in D_{priv}\}$$

Each node $v \in \mathcal{V}$ has an associated feature vector x_v, described in the previous sections. The constructed graph is undirected since the graph neural networks need to propagate the information on both sides of the communication, not just the destination. Furthermore, some nodes have only outgoing communication, so propagating only along the directed edges could result in no information aggregation for these nodes. The next section covers the snapshot generation procedure, one of the key points of the *Snapshot GNN* architecture.

4.3.2 Snapshot generation

The *Snapshot GNN* model, described in Section 2.2.3, operates on generated static graph snapshots $\mathbb{G} = \{\mathcal{G}^1, \dots, \mathcal{G}^T\}$, where $\mathcal{G}^i = (\mathcal{V}, \mathcal{E}^i)$ is an attributed graph snapshot. This section describes the snapshot generation procedure, which is one of the key components of this architecture. The shared set of nodes $\mathcal{V}$ is extracted as all devices present in the private network:

$$\begin{aligned} D_{priv} &= \{(i, j, f) | f_{priv} = 1,\ (i, j, f) \in D\} \\ \mathcal{V} &= \bigcup_{(i,j,f \in D_{priv})} \{i, j\} \end{aligned}$$

To construct the edge set for each snapshot, let's first define a set of communication logs that the node $v \in \mathcal{V}$ was part of up until the timestamp t_i (tied to the snapshot $\mathcal{G}^i$): $L_v^i = \{(i, j, f) | f_{ts} \leq t_i, i = v \vee j = v\}$. Then the considered edge set $\mathcal{E}_v^i$ for the node v and snapshot i are constructed from the N most recent connection logs in L_v^i. The edge set for the entire snapshot is then obtained as $\mathcal{E}^i = \bigcup_{v \in \mathcal{V}} \mathcal{E}_v^i$. Since each snapshot $\mathcal{G}^i$ is an attributed graph, a "temporal" set of features is computed $X^i = \{x_v | v \in \mathcal{V}\}$ out of the communication logs in set $L^i = \bigcup_{v \in \mathcal{V}} L_v^i$. To reduce the computational overhead of this architecture, the set of computed features is a subset of the features used for static classification. Therefore the communication frequency and hash features are omitted due to long computational times. There are still two parameters of the snapshot generation procedure left to decide: the number of recent edges N to consider and how to sample the snapshots throughout the day.

After consultation with domain experts, the distance between consecutive snapshots was determined to be at least 2-3 hours. Figure 4.4 shows a pattern that can further guide the sampling procedure.

The pattern of working hours indicates that sampling a snapshot within the first few hours of the day would give the model little information about the devices because the activity is very low up until the start of the working hours. Due to this pattern, no snapshots are sampled for each customer within the first 8 hours (this number was obtained experimentally). During the next 16 hours, six uniformly distributed graph snapshots are captured, resulting in one snapshot per ~ 2.6 hours.

The number of included most recent edges N was chosen out of 5 considered options: 10, 20, 30, 40, and 50. On the one hand, more edges result in more meaningful features, but on the other, the features become static, and temporality gets lost. Furthermore, the fewer edges each snapshot considers, the more sparse the graph is and the faster the training and inference are. The best option was experimentally determined to take the most recent 30 edges per node for each snapshot.

The last two sections covered constructing graphs representing the networks at hand. The following sections analyze the graph structures and introduce newly developed positional features based on that analysis.

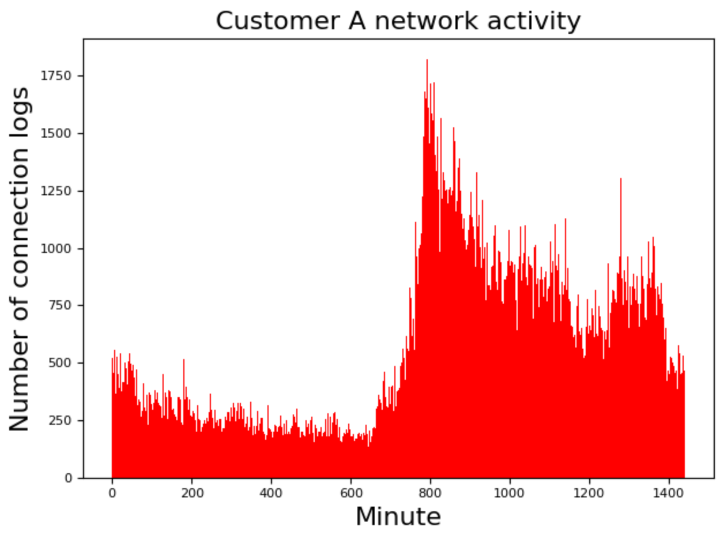


Figure 4.4: Distribution of connection logs throughout the day for Customer A. A clear pattern is visible, showing the start of the working hours followed by an elevated activity.

4.3.3 Graph analysis

This section analyzes graphs constructed in Section 4.3.1 from the entire day's worth of connection logs. Based on a benchmark performance analysis of graph libraries in [32], I chose the NetworKit library [33] due to the combination of numerous available algorithms and fast implementation. I use a directed version of the constructed graphs for this analysis as it represents the data most accurately. Table 4.5 presents basic statistics of constructed graphs.

network	log count	number of nodes	number of edges
Network 1	305 331	2 644	25 484
Network 2	8 279 277	52 948	2 208 026
Network 3	16 262 629	133 555	19 44 982
Network 4	738 305	6 337	80 666
Network 5	57 090 792	130 287	1 358 128
Customer A	839 324	5 134	39 216
Customer B	201 501	1 165	14 440
Customer C	11 214 942	65 892	1 932 538

Table 4.5: Network graphs statistics.

Even after reducing the graph sizes by constructing the graphs from private communication only, the node count in individual graphs still reaches above 100 000. The size of the graphs limits the use of methods relying on matrices derived from graphs. These methods include the spectral methods explained in Section 2.1, or even very recent methods relying on pairwise node distance matrix, such as PGNN [34] or DE-GNN [35].

Further insight into the network is gained by looking at the connected components of each network. The connected components tell us whether the networks are naturally segmented into components that do not interact with one another. Table 4.6 displays the connected component analysis.

customer	number of nodes	number of components	top 10 components
Network 1	2 644	17	2 586, 9, 8, 8, 5, 4, 4, 2, 2, 2
Network 2	52 948	59	52 791, 13, 9, 6, 4, 4, 4, 4, 3, 3
Network 3	133 555	425	132 387, 22, 14, 14, 14, 13, 12, 12, 12, 12
Network 4	6 337	11	6 302, 13, 5, 3, 2, 2, 2, 2, 2, 2
Network 5	130 287	1 485	126 757, 57, 45, 41, 37, 29, 23, 22, 19, 17
Customer A	5 134	14	5 099, 6, 4, 4, 3, 2, 2, 2, 2, 2
Customer B	1 165	4	1 157, 3, 3, 2
Customer C	65 892	4	65 886, 2, 2, 2

Table 4.6: Connected components analysis. The same pattern appears in all of the networks. Most nodes are contained within a single connected component, followed by many components of small cardinality.

The analysis shows a similar pattern for each customer. There is one single connected component, which includes more than 99% of nodes in the graph. This pattern removes the option to use the connected component analysis to classify cities for Customer A and shows that the task requires a more advanced approach.

The connected component analysis shows that the networks are interconnected, and most nodes belong to the same connected component. The traits of these interconnected networks can guide the configuration of the individual models. For graph neural networks, one of the crucial design decisions is how wide of a neighborhood the network aggregates for each node. A good base for this decision is the diameter of a graph, a measure defined as the length of the "longest shortest path" between any two vertices in a graph. The time complexity of computing the APSP (all pairs shortest path) problem is infeasible for graphs with hundreds of thousands of nodes. Therefore, I employ an algorithm [36] approximating the effective diameter, representing the 90th percentile of the shortest paths length distribution. Since it is a sampling-based approximation algorithm, it was run 100 times to ensure statistical relevance, and the results are reported in Figure 4.5.

The results show that the constructed graphs are very shallow. Consequently, models should not aggregate wide neighborhoods as, at that point, the model aggregates almost the entire graph into a single node representation.

I further analyze the properties of individual nodes rather than the graph as a whole. Especially the importance of nodes in the network, e.g., node

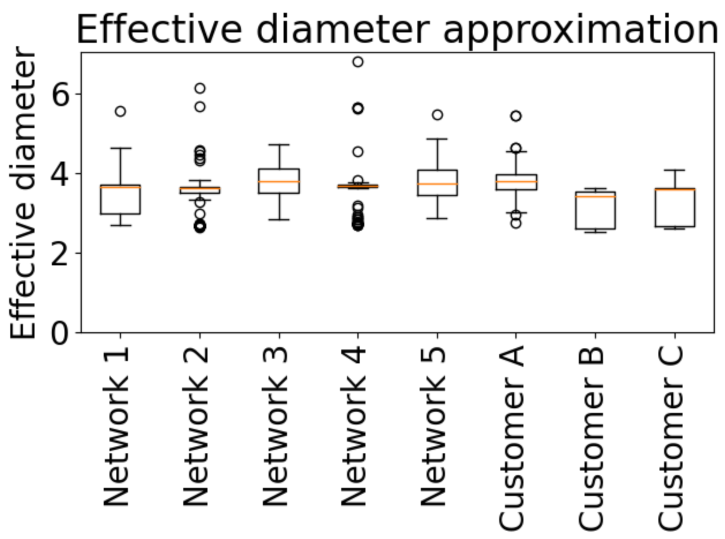


Figure 4.5: Approximation of effective diameters for each network based on 100 runs. The effective diameter represents the 90th percentile of the shortest paths in the graph.

centralities. I compute the following node centralities commonly used for graph analysis:

- **PageRank centrality** is computed according to the original paper [37] and was introduced to compute the relative importance of a web page.
- **Degree centrality** is naturally defined as the number of neighbors.
- **Betweenness centrality** is computed with the approximate algorithm introduced in [38] and represents how likely a node will occur on the shortest path between two random nodes.
- **Katz centrality** introduced in [39], computes the centrality of a node based on its degree and the degrees of its neighbors.

I further plot out the normalized distribution (to the interval [0,1], except for the degree centrality) of the centrality measures for each network. For each network, centrality values are sorted in descending order, and the top 50 values are plotted in Figure 4.6.

The centrality graphs show that in each graph, there are only a couple dozen of very influential nodes at maximum, while the rest is not that important with respect to these centralities. This result is what is to be expected in an internet network, as highly influential nodes (web servers, proxies, authentication servers, etc.) are scarce in the networks and absorb a lot of the traffic from the rest of the devices.

The computed centralities can further serve as a tool for identifying these critical and influential nodes. I utilized this approach to develop additional positional node features described in the next section.

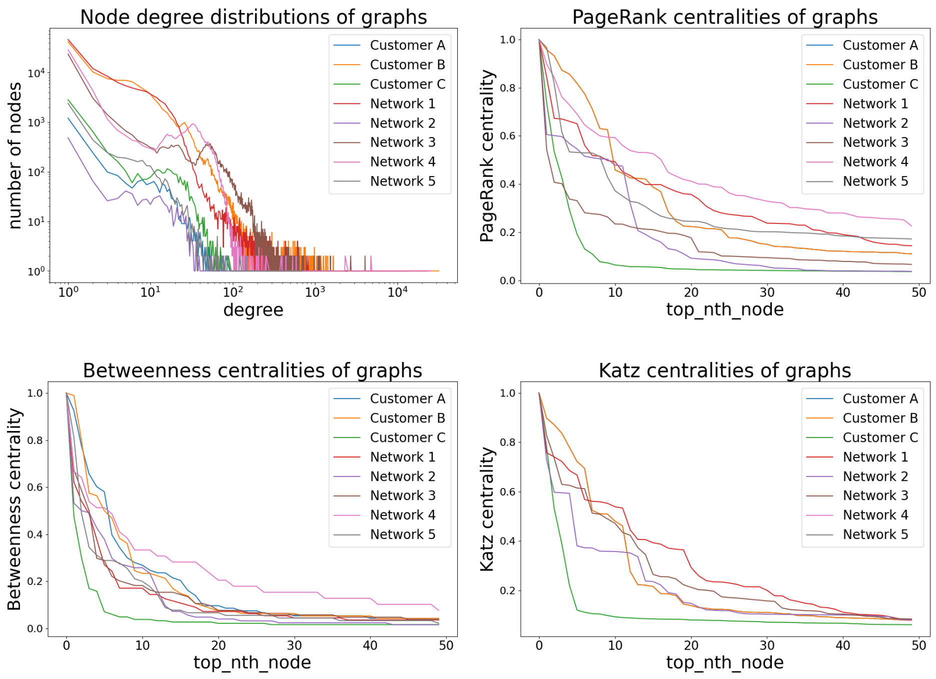


Figure 4.6: Node degree distribution and node centralities plots for each network. Each centrality plot compares scaled centrality values for the 50 most central nodes in the network.

4.3.4 New graph positional features

The device features presented in Section 4.2 carry information capturing mainly functional characteristics of a device, such as port statistics or how frequently the device communicates. Consequently, the models with only these features could not correctly classify Customer A devices into the respective cities. The experiments presented in Section 4.3.5 confirmed this issue. The general issue is to capture a node's position within the global graph structure. There exist numerous methods for this task.

As described in Section 2.1, the models utilizing random walks on the graph can produce embeddings capturing this information. I experimented with the node2vec model for this very purpose. The experiments with node2vec proved that the approach is working well, but unfortunately, the models utilizing the random walks do not scale to large networks and need to be frequently retrained.

The standard GNN models such as GAT or GraphSage have issues with this task, as confirmed theoretically and experimentally in [34]. The paper also proposed the PGNN architecture, which was later surpassed by the DE-GNN architecture [35] also targeting this task. However, both of these recent architectures suffer from computational complexity. The models rely either on pre-computed pairwise node distance matrix (PGNN) or powers of the adjacency matrix (DE-GNN), which are both computationally infeasible for large-scale graphs.

The last option is additional positional features of nodes based on the graph. The recent paper [16] proposed positional features used in combination with graph neural networks. Specifically, the authors considered additional node features of 4 types: random vector of high dimension (uniquely identifies node), one-hot encoding of nodes, node embeddings obtained by spectral decomposition (discussed in methods), and node embeddings obtained by DeepWalk algorithm (from random walks family of algorithms). The first two options do not capture the graph structure, while the latter two options cannot be efficiently used for the large-scale graph.

Due to the issues with state-of-the-art approaches for capturing the positional node information, I have developed new positional features capturing the position of a node within the graph. The proposed positional features are computationally very lightweight, scaling easily to large graphs while providing condensed information in a low-dimensional format. The effect of these features is confirmed by experiments in Section 4.3.5, where the features are used for city prediction of devices in the network of Customer A.

To construct the node features, I select a centrality measure and the top N nodes ordered by the respective centrality score. Distance to each of these nodes then represents a single feature for each node in the network. With a set of central nodes denoted S and a node m , the features are computed as:

$$m_{distance\_s} = \begin{cases} dist(m, s) & \text{if } m,\ s \text{ in the same connected component} \\ -1 & \text{otherwise} \end{cases} \forall s \in S$$

Since the constructed graphs are undirected and unweighted, the distance $dist(m, s)$ is computed by the BFS algorithm computing single source shortest paths. It is run only once from each of the selected N nodes, resulting in low computational times scalable to any network size.

4.3.5 Positional features ablation study

This section presents an ablation study on the positional features to evaluate the effect of the proposed positional node features. The device labels of Customer A represent the city where the device is located, which is a perfect classification task for testing the effect of positional node features. The results, presented in Section 5.2.2, show that the baseline algorithms achieved only underwhelming 40% accuracy. It was to be expected since the device features aggregated from connection logs do not contain any features that could distinguish two devices with similar communication patterns in different cities. It is confirmed when looking at the features' importance for the RF classifier displayed in Figure 4.7.

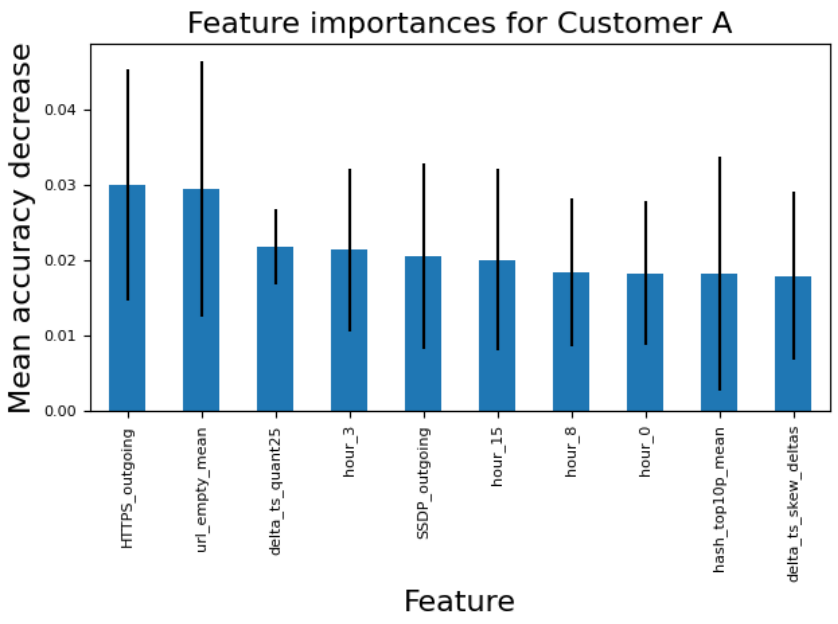


Figure 4.7: Feature importance for RF algorithm on predicting cities. The importance was computed by the feature permutation-based algorithm [2].

The feature importances show that the individual features have a high variance of mean accuracy decrease (the importance is unstable) and do not carry information for distinguishing geolocation. The following experiment confirms that leveraging the graph structure is the key to solving this classification task. One of the first graph neural networks, the node2vec architecture introduced in [11], can be set up to capture the necessary information for our classification. According to the authors, the node2vec algorithm can be set up in two ways, each capturing different information. The effects of the two main setups are shown in Figure 4.8.

For the task at hand, the node2vec embeddings should capture the community indication (*homophily*), which is displayed in the upper part of the figure. This is accomplished by controlling the parameters p, q of the random walks generation procedure, as described in Section 2.1. To capture this

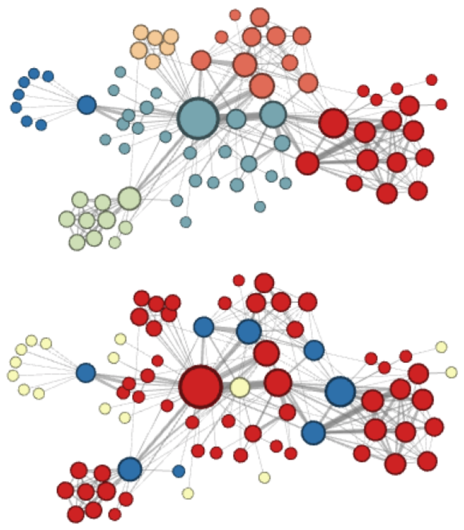


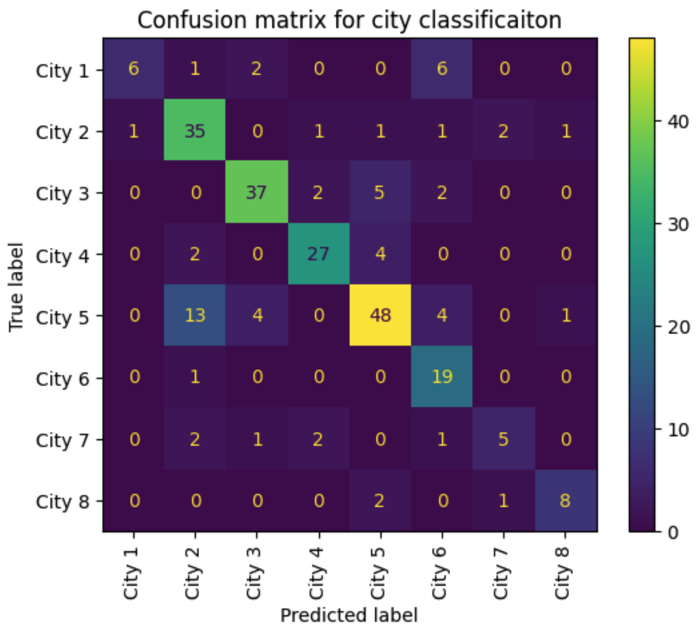
Figure 4.8: Two different setups of the node2vec algorithm. The node2vec node embeddings can capture either community indication (upper picture) or structural similarities (lower picture). The picture is taken from the original authors of node2vec [11].

information I selected $p = 1.5,\ q = 0.3$ as hyperparameters for the random walk generation. The node2vec algorithm runs in an unsupervised manner, producing embeddings for each node that reflect this goal. This approach, paired with the SVM classifier on top of the trained embeddings, produced the results in Figure 4.9:

The confusion matrix corresponds to 75% accuracy, significantly outperforming the baseline models and the GNN without the positional features (results in Figure 4.10). The results have proven that the graph structure can be exploited to capture the *homophily* information in this network. As discussed previously, this approach is not scalable to large networks, and therefore the next part of this section compares this approach to utilizing the positional features. I have designed an experiment in which I train the *GNN* model with two types of features available. The first type is the features mined from communication logs; these are always present. The second type of feature is the various types of positional features. I test these various types of positional features:

- **None** - The node features do not contain any positional features.
- **node2vec embeddings** - The node2vec node embeddings are used as additional positional features for the graph neural network. Embeddings of multiple sizes were trained to obtain the best performance.
- **Distance to randomly picked 50 nodes.**
- **Distance to 50 most central nodes**. (newly developed features)

The number of 50 most central nodes was selected based on the analysis presented later in this section. I consider three centralities for picking the



True label \ Predicted label	City 1	City 2	City 3	City 4	City 5	City 6	City 7	City 8
City 1	6	1	2	0	0	6	0	0
City 2	1	35	0	1	1	1	2	1
City 3	0	0	37	2	5	2	0	0
City 4	0	2	0	27	4	0	0	0
City 5	0	13	4	0	48	4	0	1
City 6	0	1	0	0	0	19	0	0
City 7	0	2	1	2	0	1	5	0
City 8	0	0	0	0	2	0	1	8

Figure 4.9: node2vec embeddings + SVM classifier results.

central nodes: Betweenness, PageRank, and Katz. For each of these options, I have trained and evaluated the *GNN* model with respective features, and the results are presented in Figure 4.10.

The results in Figure 4.10 clearly show that the new positional features (with respect to the PageRank centrality) improve the accuracy of the *GNN* model significantly. The features based on the distance to the central nodes w.r.t. the PageRank centrality improve the accuracy score by $\sim 10\%$ compared to the GNN without any positional features. In this setting, the model surpassed the approach with node2vec embeddings paired with an SVM classifier and achieved a similar result as *GNN* model paired with node2vec positional embeddings. The results show that the positional features capture the node position in the graph well, to a similar extent as the node2vec approach.

The results also show that the node2vec embedding dimension had to be raised to 100 for optimal performance. There was no further gain in performance when rasiing the dimension even further. Not only were the positional features able to provide similar information in lower-dimensional representation, but they were also incomparably faster to compute. For this graph, containing only around 5000 nodes, the training of node2vec embeddings took 5.43 minutes, while the newly developed features were computed in 0.0003 seconds (including the PageRank algorithm). Even for one of the large-scale networks, the network of Customer C, containing around 65000 nodes, the positional features take only ~ 1.7 seconds to compute. The final advantage of this approach is that if a new node appears in the graph or the graph structure changes, the re-computation of these features is almost instantaneous compared to other approaches.

To further validate how many central nodes are necessary to capture

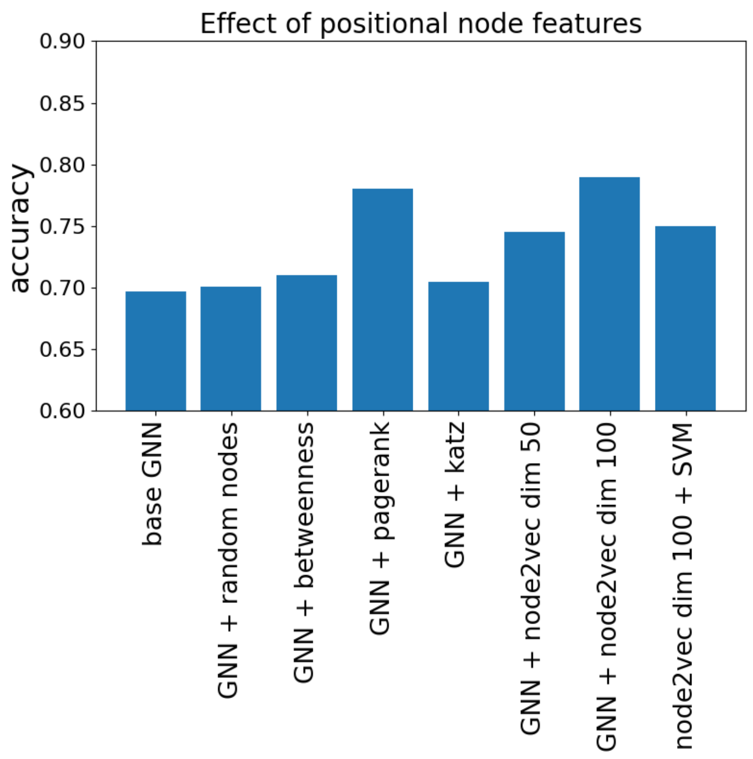


Figure 4.10: Comparison of the effect of positional node features on prediction accuracy. The positional node features are compared to the *GNN* without them and the node2vec embedding+SVM approach. I report the best-achieved result for each method.

this information in this particular network, Figure 4.11 studies the effect of choosing various numbers of central nodes in this scenario.

The plots suggest that including distances to less than 20 central nodes (w.r.t. PageRank) is insufficient and that the model's performance is close to the model's results with no positional features. The steep incline at 20-30 most central nodes and stagnation at 50-60 central nodes indicates that the number of central nodes could potentially be reduced, depending on the network. However, to capture as much information as possible, the setup of including the 50 most central nodes w.r.t. to the PageRank centrality was used in the experiments in the following sections.

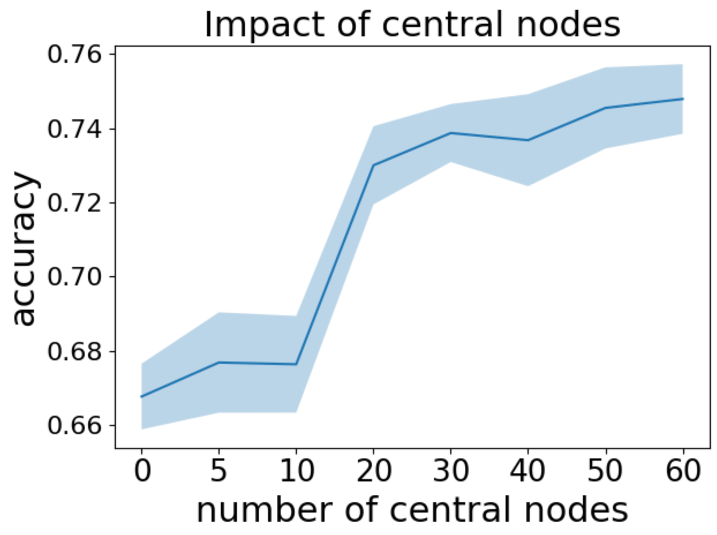


Figure 4.11: The effects of choosing the number of central nodes N to compute the distances to. To ensure statistical relevance, ten runs were performed for each number of nodes, and 95% confidence intervals based on the Student's t-distribution are displayed.

Chapter 5

Experiments

The implemented models and representations were tested in-depth to evaluate the usability of proposed solutions in the real world. Two main experiments were designed to capture the necessary information for this evaluation. A detailed analysis of results is provided after describing each experiment. The experiments are followed by a section presenting the required computational resources for each method, one of the critical aspects of model usability in the real world. However, to run the experiments, all the methods need to be configured for the training procedure, which is described in the following section.

5.1 Models configuration

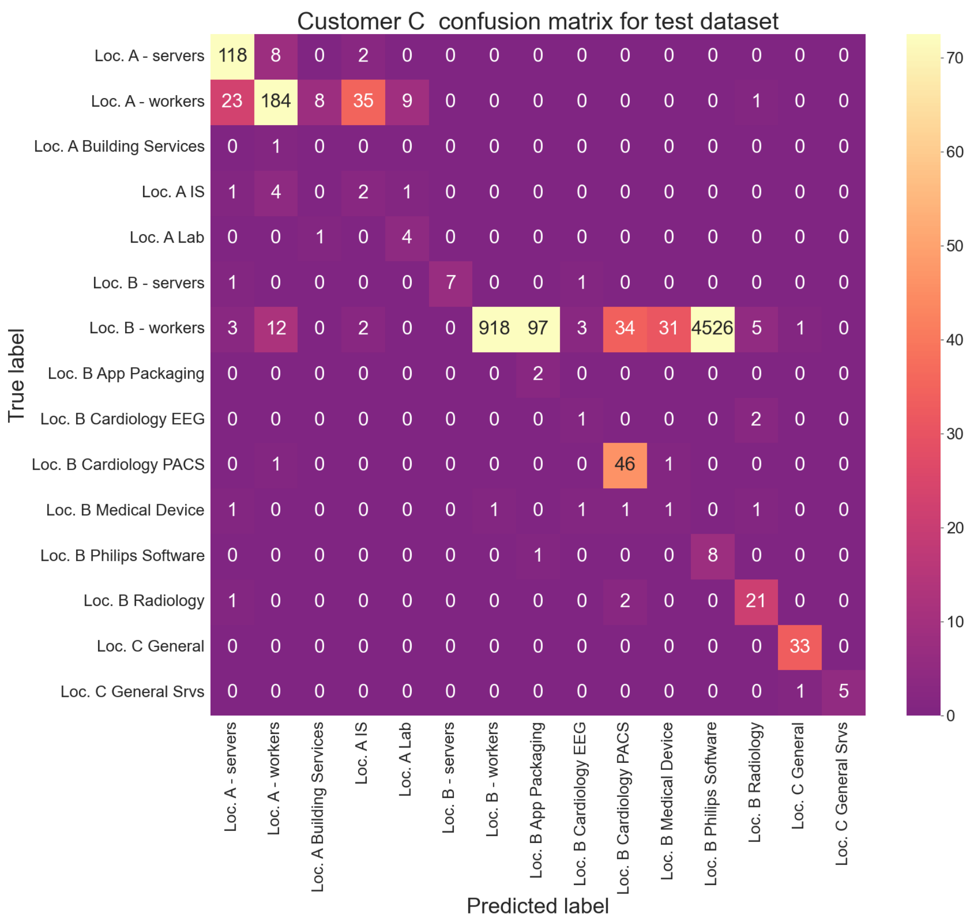
Each model has a set of hyperparameters that need to be configured for the training procedure. This section shortly covers the configuration of these hyperparameters and the specifics of the training process.

One of the substantial challenges in the training process was the class imbalance issue. Customer A has a minor class imbalance problem among its eight classes, and the models were able to perform on this customer without special measures for solving class imbalance.

To address the class imbalance present in the dataset of Customer B and the heavy class imbalance issue of Customer C, I have experimented with three approaches: class weights, subsampling, and no adjustments.

The class imbalance issue of Customer B was handled by applying class weights [40] to the cross entropy loss function for the graph neural network models. The same weights were applied to the training procedure of the baseline models RF and SVM. The AdaBoost classifier performed well even without the class weights applied to the training.

In the most severe imbalance case, the Customer C dataset, one majority class (among 15) makes up most of the devices ($\sim 91\%$). I had to perform numerous experiments for this customer to set up the learning procedure appropriately. First, if the device counts in the training set were proportionally equivalent, the models generally just classified all the devices as the one majority class. This issue could not be solved by applying class weights because the weight for the majority class was so low that the models did not



True label \ Predicted label	Loc. A - servers	Loc. A - workers	Loc. A Building Services	Loc. A IS	Loc. A Lab	Loc. B - servers	Loc. B - workers	Loc. B App Packaging	Loc. B Cardiology EEG	Loc. B Cardiology PACS	Loc. B Medical Device	Loc. B Philips Software	Loc. B Radiology	Loc. C General	Loc. C General Srvs
Loc. A - servers	118	8	0	2	0	0	0	0	0	0	0	0	0	0	0
Loc. A - workers	23	184	8	35	9	0	0	0	0	0	0	0	1	0	0
Loc. A Building Services	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Loc. A IS	1	4	0	2	1	0	0	0	0	0	0	0	0	0	0
Loc. A Lab	0	0	1	0	4	0	0	0	0	0	0	0	0	0	0
Loc. B - servers	1	0	0	0	0	7	0	0	1	0	0	0	0	0	0
Loc. B - workers	3	12	0	2	0	0	918	97	3	34	31	4526	5	1	0
Loc. B App Packaging	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0
Loc. B Cardiology EEG	0	0	0	0	0	0	0	0	1	0	0	0	2	0	0
Loc. B Cardiology PACS	0	1	0	0	0	0	0	0	0	46	1	0	0	0	0
Loc. B Medical Device	1	0	0	0	0	0	1	0	1	1	1	0	1	0	0
Loc. B Philips Software	0	0	0	0	0	0	0	1	0	0	0	8	0	0	0
Loc. B Radiology	1	0	0	0	0	0	0	0	0	2	0	0	21	0	0
Loc. C General	0	0	0	0	0	0	0	0	0	0	0	0	0	33	0
Loc. C General Srvs	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5

Figure 5.1: Confusion matrix of the test dataset from a *GNN* model trained on a training dataset with the majority class subsampled and class weights applied to the loss function.

consider it at all. Therefore, as a countermeasure that helped significantly with this issue, I used heavy subsampling of the majority class in the training dataset. After subsampling, the number of devices from the majority class in the training dataset approximately matches the number of devices from the second-most cardinal class. However, the class imbalance among 15 classes still remains problematic in the dataset. After the subsampling procedure, I tested the class weight approach, but the results suggest that it is not a good fit for this task. Figure 5.1 presents the results of the *GNN* model when class weights were applied to the training.

The plot shows a significant issue in the predictions; the **Loc. B-workers** row shows that the majority class is not well classified. The predictions for the devices in the majority class are spread among similar device classes of low cardinality. This issue is caused by the class weights that modify the loss function to assign the same importance for each class irrespective of its cardinality. A similar, less severe issue can be seen in the **Loc. A-workers** row. Overall, the classification accuracy drops, and most of the devices in the test dataset are not correctly classified. These issues have led me not to apply the class weights after the subsampling procedure, which resulted in much better results. The results of the same model trained without the class weights are displayed in Figure 5.6. The confusion matrix shows that the model was able to classify well not only the cardinal classes but also a lot of the lower cardinality classes as well. Further discussion of the results is provided in the respective section.

After specifying the general training setup for all the models, the specific configuration of each model is discussed in the following sections, starting with the baseline models.

5.1.1 Configuration of baseline models

The baseline models include the SVM, Random Forest, and AdaBoost algorithms, which are all relatively simple and cheap models to train. Therefore, all three are set up so that each training procedure includes a search over the hyperparameter grid and picks the best set of hyperparameters based on a 3-fold cross-validation. The selection of the hyperparameter grid, based on the study [41], is presented in Table 5.1.

model	**hyperparameter**	**options**	**number of combinations**
SVM	kernel gamma C	rbf, sigmoid 0.1,0.01,0.001 0.1,1,10,100	24
RF	n_estimators min_samples_leaf max_features	75, 150, 200 1,5,10 0.3,0.6, None	36
AdaBoost	n_estimators learning_rate algorithm base_estimator_depth	75,150 0.01,0.1,1 SAMME, SAMME.R 3,5,10	36

Table 5.1: The grid-search for hyperparameters of each of the baseline algorithms. The base estimator used for the AdaBoost algorithm was the commonly used DecisionTree model.

The SVM model further requires feature scaling before training and evaluation. Experiments showed that there is minimal difference between normalization approaches. The min-max scaling was used in further experiments.

5.1.2 Configuration of graph-based models

Both architectures, the *GNN* and the *Snapshot GNN* are graph neural networks operating either on one attributed graph (*GNN*) or on a set of attributed graph snapshots (*Snapshot GNN*). Scaling the node features is also beneficial for neural network models in general; therefore, I also use the min-max scaling for these models. Initial experiments also revealed that a one-layer network aggregating a 1-hop neighborhood is sufficient. Deeper architectures resulted in worse performance, longer training times, and an even more exhaustive hyperparameter-optimization process. One could also argue that even a 1-layer graph convolutional network aggregates information from the 2-hop neighborhood since the crafted node features contain aggregated information from the edges of a given node.

Since the training procedure for the *GNN* model is more expensive than for the baseline models, the set of optimal hyperparameters was established separately before each network's training procedures. The obtained set of hyperparameters for the *GNN* model is presented in Table 5.2 for each customer.

hyperparameter	**Customer A**	**Customer B**	**Customer C**
learning rate	0.003	0.01	0.004
num_heads	8	4	10
embedding_dim	4	4	40
l2_reg	0.004	0.04	0.001
dropout	0.4	0.2	0.1
n_epochs	400	200	250

Table 5.2: The obtained set o optimal hyperparameters for the *GNN* model for each customer. These hyperparameters were then used for each model training in the following experiments.

The *num_heads* represents the number of heads in the attention mechanism of the GATv2 architecture, and the *embedding_dim* represents the dimension of the final embedding that is further used by the classification layer. The *Snapshot GNN* architecture uses the same *GNN* model for the inference; therefore, a very similar set of hyperparameters was obtained for this model and is omitted for the sake of brevity. Both of the models were implemented in PyTorch Geometric [42] framework and trained by the Adam [43] algorithm. The Tune framework [44] was employed for the initial hyperparameter search for both of the models.

5.2 Experiment 1 - Training stability

This experiment is designed to evaluate the models' performance from two perspectives. The first is how well the models can learn the diverse classes. The diversity of the classes was assured by the selection of three networks for classification containing various device types as well as various scales and topologies of networks. The second part of this experiment is to assess whether the models' training procedure is stable throughout the week. The stability of the training process is an essential metric for potential real-world use since, generally, models need to be retrained to follow the current data distribution.

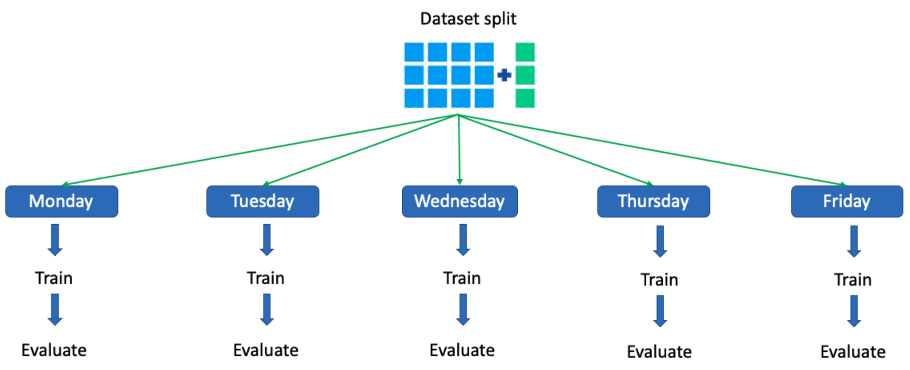


Figure 5.2: Experiment setup to evaluate the performance of models throughout the whole working week and to assess the stability of the training process throughout the week.

5.2.1 Experiment setup

The models are designed to be operated on one day's worth of data. To validate the models' training stability, they are trained and evaluated on each day of the working week, Monday through Friday. For classification results collected from various days to be comparable, the train/test split of the devices remains identical throughout the week. Let's denote $\mathbb{N}^l_{Monday}$ set of labeled devices occurring in the data on Monday. Then the device dataset is computed as the intersection of devices occurring on each respective day:

$$\mathbb{N}^l = \mathbb{N}^l_{Monday} \cap \mathbb{N}^l_{Tuesday} \cap \mathbb{N}^l_{Wednesday} \cap \mathbb{N}^l_{Thursday} \cap \mathbb{N}^l_{Friday}$$

For the experiment to have any statistical significance, numerous repetitions of splitting $\mathbb{N}^l$ into the train/test split were performed, resulting in 10 datasets. Once the 10 dataset splits of devices are obtained, the training and evaluation procedures follow on data from each respective day Monday-Friday, for each split as shown in Figure 5.2.

The experiment setup leads to 10 datasets for each day of the working week; therefore, there are 50 training and evaluation phases for every single model.

5.2.2 Results

Having ten classification results per day allows for constructing confidence intervals for more reliable guarantees of the models' performance. This small sample allows me to infer the confidence intervals based on the Student's t-distribution [45], commonly used for estimating the mean of a normally distributed population with a small sample size and unknown standard deviation. All the plots I provide have the typical 95% confidence interval plotted for each applied classifier.

I utilize common standard metrics for classification analysis used in machine learning, defined, for instance, in [46]. The confusion matrix for the classification in the binary case defines the basic elements for constructing multi-class classification metrics:

	actual positive class	**actual negative class**
predicted positive class	True positive (tp)	False positive (fp)
predicted negative class	False negative (fn)	True negative (tn)

Table 5.3: Confusion matrix for binary classification.

The derived classification metrics that can be constructed for each respective class in a one-versus-all manner are expressed as:

$$\text{Accuracy} = \frac{tp + tn}{tp + fp + tn + fn}$$
$$\text{Precision} = \frac{tp}{tp + fp}$$
$$\text{Recall} = \frac{tp}{tp + fn}$$
$$\text{F1-measure} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Precision, *recall*, and *F1-measure* are defined for each respective class and can be averaged over classes in a macro manner, defined in [46], to obtain the metric for the entire testing dataset:

$$\text{Average recall} = \frac{\sum_{i=1}^{l} \text{recall}_i}{l}$$

Where $recall_i$ refers to the *recall* of the class i. The same macro average is computed to obtain the metric for *precision* and *F1-score*.

The analysis of classification results must be done carefully due to the heavy class imbalance present for Customer B and Customer C. The achieved accuracy for each customer is plotted in Figure 5.3.

Since Customer A is the only one not suffering from severe class imbalance issues, the accuracy metric is meaningful for this customer only. The results for this customer show that the graph-based approaches achieve much better classification results than the baseline models consistently throughout the week. The top performer is the *GNN* model, while the *Snapshot GNN* achieved similar results.

Due to class imbalance issues for Customer B and Customer C, I present the macro average recall and precision in Figure 5.4 for these two customers.

For Customer B, the wide confidence intervals suggest that the model's recall and precision are highly unstable. However, a deeper analysis is necessary to understand why these macro measures are inconsistent. For Customer C, the recall and precision metrics, much lower than the accuracy, might suggest that the model correctly classifies primarily the majority class. However, again an even deeper insight into the results suggests otherwise. In the following two upcoming sections, I provide an in-depth classification results analysis for both customers.

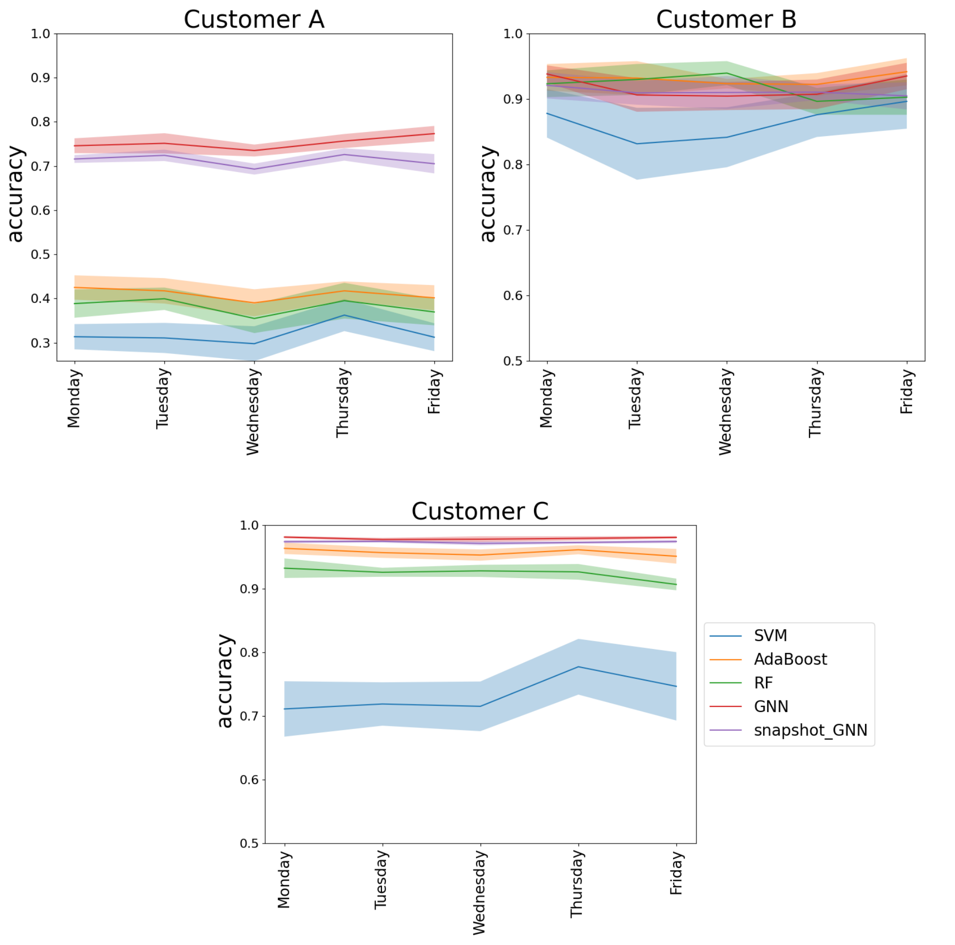


Figure 5.3: Classification accuracy results. For each method, the 95% confidence interval for the accuracy metric is displayed.

Customer B results analysis

One of the key elements for classification results analysis is the class imbalance of this customer. Since the test datasets account only for 25% of devices, the resulting device counts in both the train and test datasets usually look similar to the ones in Table 5.4.

The issue with unstable macro averaged metrics comes from the counts of devices in the test dataset. In Figure 5.5 and Table 5.5, I present a single result of the GNN model to highlight the problem.

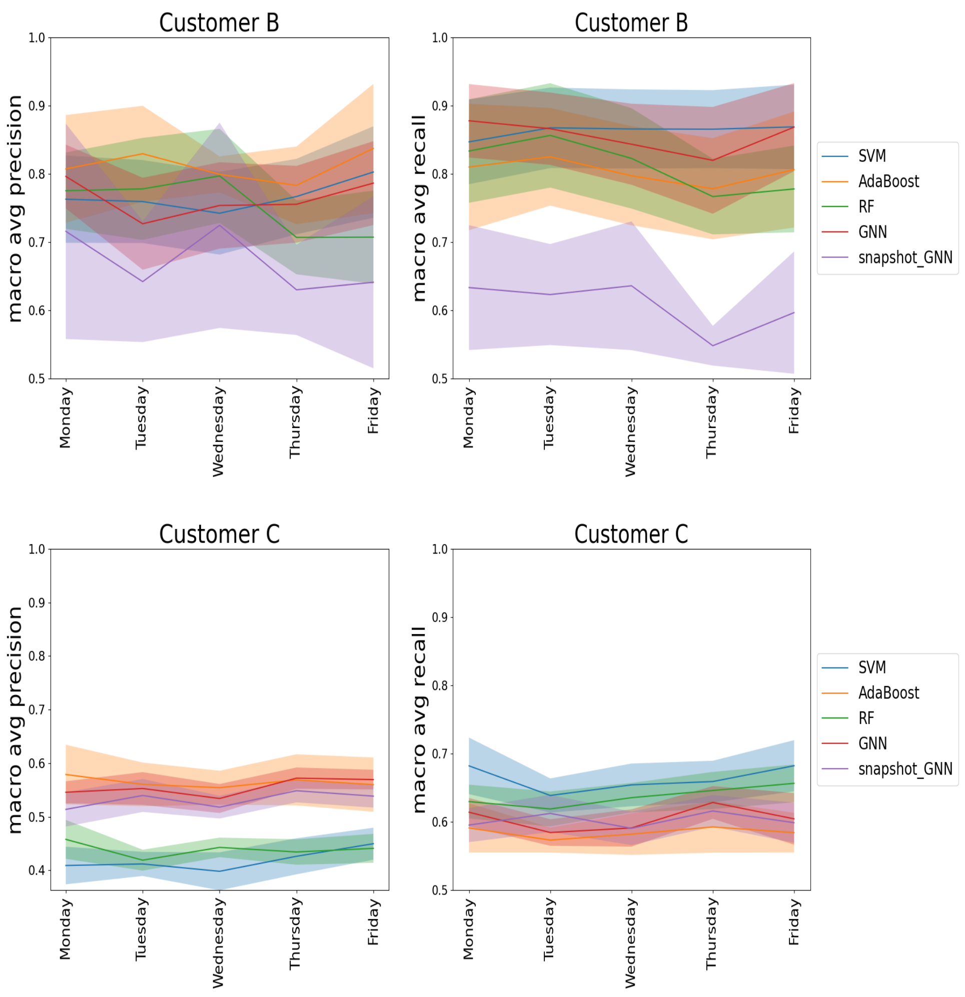


Figure 5.4: Classification recall and precision metrics along with confidence intervals for each classifier.

label	device count	train dataset count	test dataset count
Domain Controller	11	6	5
Protect	298	229	69
Protect - IT	20	15	5
Server	69	56	13

Table 5.4: Customer B label count for train/test dataset.

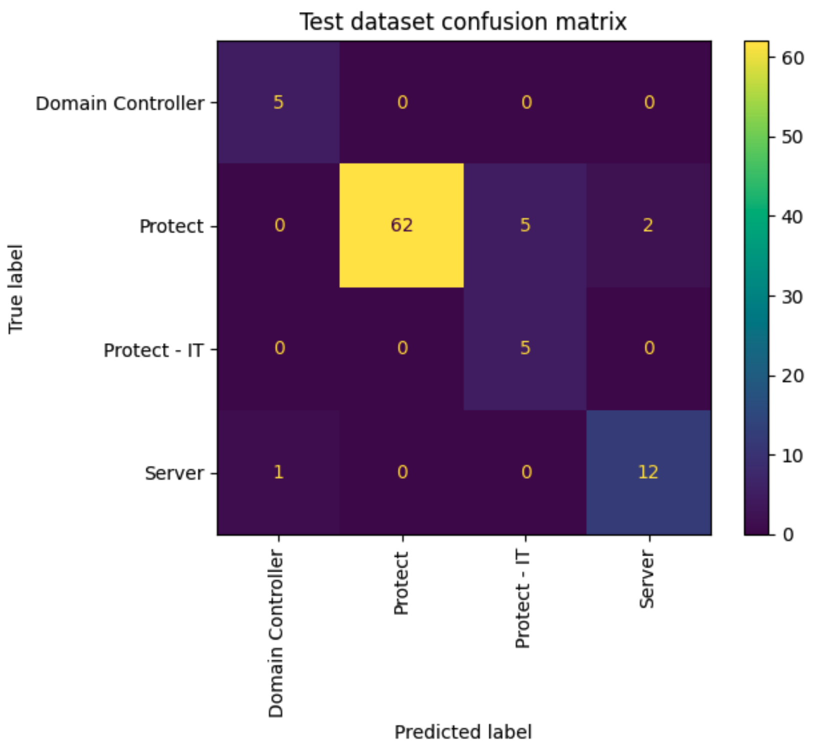


Figure 5.5: Confusion matrix of a *GNN* model on the test dataset of Customer B.

	precision	**recall**	**F1-score**	**support**
Domain Controller	0.83	1.00	0.91	5
Protect	1.00	0.90	0.95	69
Protect-IT	0.50	1.00	0.67	5
Server	0.86	0.92	0.89	13
macro avg	0.80	0.96	0.85	
accuracy	0.91			

Table 5.5: Derived performance metrics from the confusion matrix in Figure 5.5.

In this case, the macro average for precision and recall is reported to be relatively high. However, it would be enough to classify only one domain controller incorrectly and one device from **Protect-IT** class incorrectly, and the reported macro average for recall would fall by more than 10% to 0.855. This significant change caused by only two devices is caused by the low number of devices within the two classes. The lower the number of devices in the class, the more impact each misclassified device from this class has on the metric.

This issue is even worse than in this case since the testing dataset usually has fewer than five devices in the **Domain Controller** class. Therefore, one misclassified device from this class (out of 92 total devices) significantly impacts the macro averaged metrics, leading to the confidence intervals being very broad and the reported macro averaged score unstable. There is another option to report the averages weighted by the support of each respective class, but then the majority class dominates the metrics.

Because of this issue, the confidence interval plots can give only limited information on the classification performance, and a more profound analysis with a confusion matrix and classification metrics for each respective class is necessary.

The confusion matrix shows that the *GNN* model mostly makes a good guess for the few misclassified devices. For example, the predicted **Protect-IT** devices fall either into **Protect-IT** or **Protect** class, which both represent workstations. The one incorrect prediction of **Domain Controller** for a server is also a sound suggestion since the predicted class is closest to the correct one out of the rest. Furthermore, as discussed in Section 3.3, the devices may be assigned to the wrong class, and therefore the models' predictions may even be valid.

Since the class imbalance is even more severe for Customer C, the following section focuses on a similar deeper analysis of the results for this customer.

Customer C result analysis

The classification analysis for this customer has to be even more careful, as there are 15 classes for the devices in this network, many of which also have very low cardinality. This low cardinality causes similar issues with the average macro scores as it does for Customer B, although the variation is lower due to the high count of classes. For this customer, I have picked one of the train/test splits to show the distribution of devices:

label	**device count**	**train dataset count**	**test dataset count**
Loc. A - servers	514	389	125
Loc. A - workers	1 022	805	217
Loc. A Building Services	10	7	3
Loc. A IS	24	16	8
Loc. A Lab	34	26	8
Loc. B - servers	26	19	7
Loc. B - workers	22 236	784	4 811
Loc. B App Packaging	15	9	6
Loc. B Cardiology EEG	20	19	1
Loc. B Cardiology PACS	176	130	46
Loc. B Medical Device	27	23	4
Loc. B Philips Software	49	38	11
Loc. B Radiology	84	60	24
Loc. C General	92	71	21
Loc. C General Srvs	27	22	5

Table 5.6: Customer C label count for train/test dataset. The **Loc. B - workers** class is heavily subsampled in the training dataset.

The high amount of low cardinality classes in Table 5.6 already hints at a similar issue in the macro averaged performance metrics that were described in the last section. The analysis again focuses on the presented classification results in Table 5.7 and the confusion matrix in Figure 5.6.

	precision	**recall**	**F1-score**	**support**
Loc. A - servers	0.93	0.97	0.95	125
Loc. A - workers	0.89	0.96	0.92	217
Loc. A Building Services	0.0	0.0	0.0	3
Loc. A IS	0.0	0.0	0.0	8
Loc. A Lab	0.0	0.0	0.0	8
Loc. B - servers	1.0	0.71	0.83	7
Loc. B - workers	1.0	0.99	1.0	4811
Loc. B App Packaging	1.00	0.17	0.29	6
Loc. B Cardiology EEG	0.14	1.0	0.25	1
Loc. B Cardiology PACS	0.70	1.00	0.82	46
Loc. B Medical Device	0.14	0.25	0.18	4
Loc. B Philips Software	0.0	0.0	0.0	11
Loc. B Radiology	0.95	0.83	0.89	24
Loc. C General	0.95	0.95	0.95	21
Loc. C General Srvs	0.80	0.80	0.80	5
macro avg	0.57	0.58	0.53	
accuracy	0.98			

Table 5.7: Derived performance metrics from confusion matrix in Figure 5.6.

The performance results in Table 5.7 for respective classes seem to fall into two extremes. Either the model can distinguish the class well or almost not at all. The focus of this analysis is then on the classes that the model struggles with. The confusion matrix in Figure 5.6 provides the necessary detail for analyzing the GNN's confusion.

For example, the confusion matrix displays that the model classified all devices from classes **Loc. A Lab**, **Loc. A IS**, **and Loc. A Building Services** entirely into the **Loc. A−workers** class. However, this is a correct assessment as both classes fall within the **Loc. A−workers** class, the customer had just separated a subsection of this class into a separate class. Similar results from other models indicate that there is not a distinct difference among these three classes, so the models assign the majority class out of these similar ones.

A similar situation occurs for misclassified devices from low cardinality classes **Loc. B App Packaging**, **Loc. B Cardiology PACS**, **Loc. B Medical Device**, **Loc. B Philips Software**, and **Loc. B Radiology**.

All devices from the **Loc. B Philips Software** class were classified into the **Loc. B−workers** category, which again is not an entirely wrong assessment since the presence of different software does not necessarily translate into different behaviour on the network and it still falls into the broader category of **Loc. B−workers**. The confusion matrix for the other listed classes shows that the devices were categorized as the general class **Loc. B−workers** or the other subcategories of workers from **Loc. B**.

The classes representing servers were usually classified very well, and overall, the results show that the model was able to learn the characteristics of the

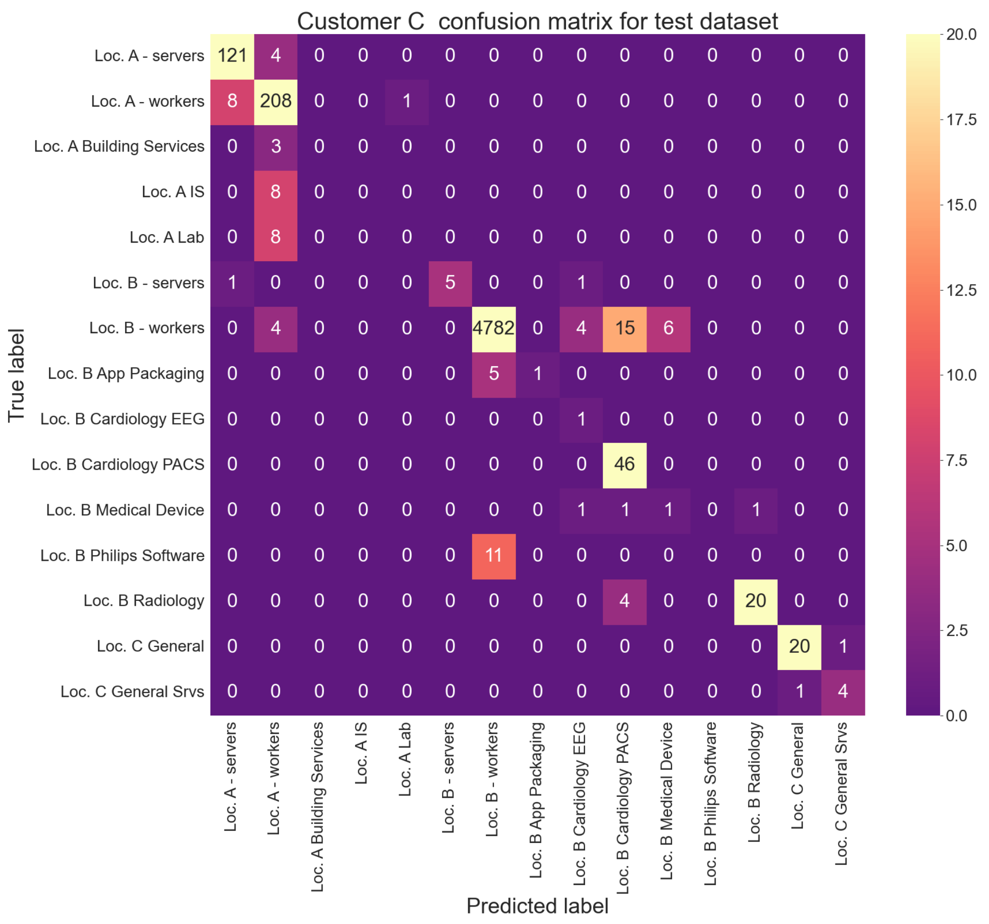


Figure 5.6: Confusion matrix of the test dataset.

classes with higher cardinality, and even for the classes with few devices only, the model is making either correct or at least sensible predictions.

Experiment summary

The first experiment and the following analysis have confirmed that the device representation captures the necessary information for classifying the diverse device types. The baseline models performed significantly worse for Customer A, where the graph structure became a crucial component. The graph structure has proven highly beneficial and enabled the graph-based models to learn the specifics of the classes very well. The two graph-based models are the best performers across the evaluation. The newly developed model, the *Snapshot GNN* model, has achieved similar results as the *GNN* model while providing the ability to work with dynamic networks and therefore track the devices throughout the day.

Furthermore, the results demonstrated that the training procedure is stable throughout the week, allowing for retraining the models when necessary. The following experiment evaluates whether the models need the retraining frequently and examines how the performance of already trained models degrades over time.

5.3 Experiment 2 - Prediction stability

The second experiment logically follows the first one, testing how frequently, if at all, the models must be retrained in order to retain their predictive abilities.

5.3.1 Experiment setup

Since this experiment aims to test how long the models retain their performance, all of the models are trained on a single day and evaluated on six consecutive days. Wednesday was decided to be the initial training day for the models since, usually, the data during the middle of the week contain the most common traffic.

The devices occurring on Wednesday were again split into the train/test datasets, similarly to the first experiment. To obtain statistical boundaries for the models' results, ten randomized train/test splits were again performed. Once the splits are obtained the training and evaluation phases follow, as displayed in Figure 5.7.

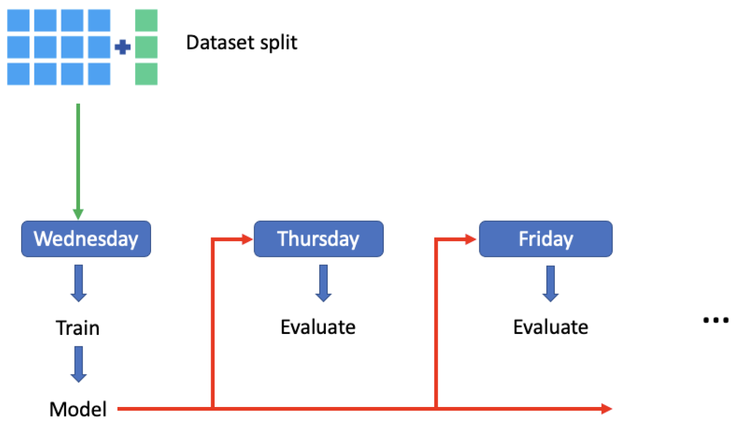


Figure 5.7: Setup of the second experiment

On Wednesday, the models are evaluated on the test split of the dataset, and for the following six days (Thursday - Tuesday), the models are evaluated on **all labeled devices** occurring on each respective day.

Graph-based methods setup

The methods relying on positional features must be carefully set up for this experiment. As explained in Section 4.3.4, the positional features mined from graph structure compute the distance to a set of central nodes present in the graph. Based on the ablation study in Section 4.3.5, an amount of 50 central nodes was selected for this purpose. If there are N devices in the data, the resulting positional feature matrix $P \in \mathbb{R}^{N\times 50}$ has one dimension for each specific central node. Since on the rest of the days, Thursday through

Tuesday, models are only evaluated and not trained, the meaning of these features must be the same as in its training phase, and the initial set of central nodes must be relied upon for each of the evaluation phases, assuming that the central nodes are active through rest of the week. I provide plots showing how many central nodes remain in the network for the rest of the evaluating days for each of these three customers:

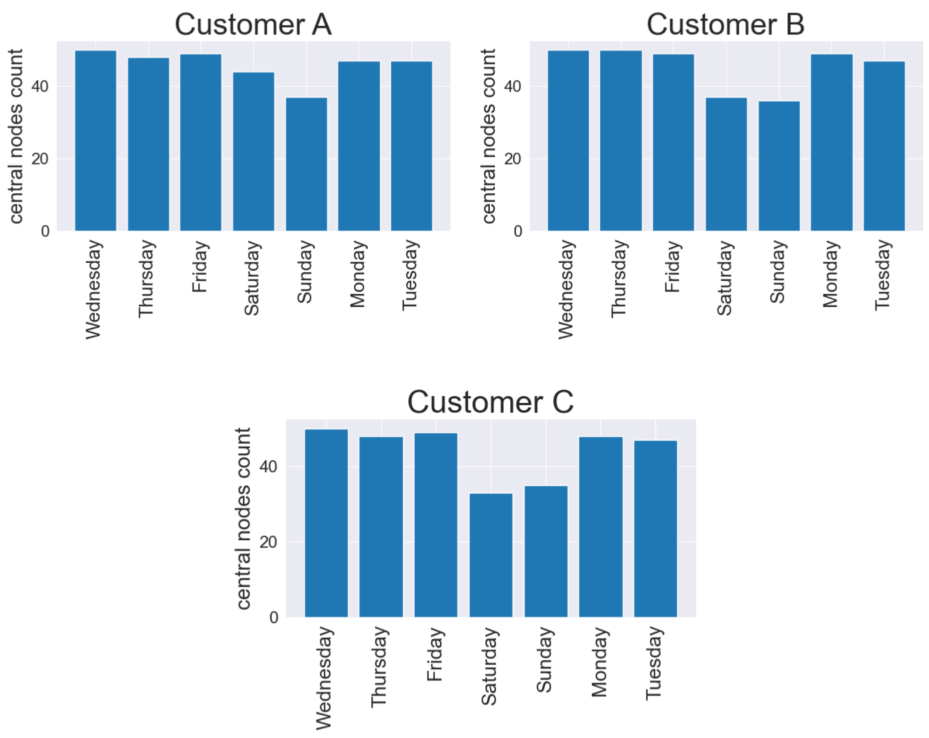


Figure 5.8: Central nodes presence for each of the customers for each of the evaluation days.

Figure 5.8 shows expected behaviour, that for each customer, the number of detected central nodes drops significantly during the weekend. Since the nodes are not present in the data, the distance features cannot be computed and are set to -1 (which represents unreachable). I suspect that to be one of the reasons for the poor performance of the graph-based models on weekend data, as discussed in upcoming sections.

5.3.2 Results

Since all the devices are evaluated for each customer on days Thursday through Tuesday, I first examine the occurrences of each class for Customer B in Figure 5.9.

The figure shows an expected pattern, the general class Protect shrinks the most during the weekend. The other classes, representing servers and IT workstations, remain much more stable even throughout the weekend. This pattern is the expected behaviour and holds for the other two customers.

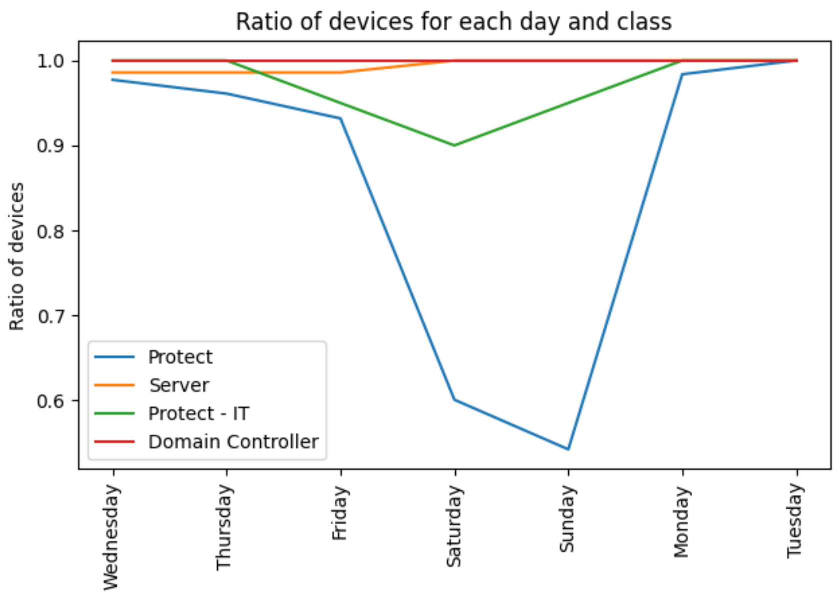


Figure 5.9: Customer B ratio of occurring devices for each class in each respective day. The ratios are obtained by normalizing the number of devices by the maximum of devices observed for each class.

The first reported metric for all customers and methods is reported in Figure 5.10.

The accuracy results for Customer A indicate that the performance of graph-based models, which were the best-performing models for this customer by a large margin in the previous experiment, rapidly deteriorates and even falls below the performance of the baseline models. One of the key factors contributing to performance issues with the graph-based methods could be the changes in the graph structure, especially on weekends, as shown previously.

Again, since the *accuracy* metric is only a meaningful metric for Customer A, I also report the *macro averages* for *recall* and *precision* for customers B and C in Figure 5.11.

Since the testing dataset is 100% (not just 25%) of devices on Thursday through Tuesday, the lower-cardinality classes have more devices, and the issue with unstable confidence intervals from the previous experiment does not occur. Consequently, the confidence intervals for Customer B are much narrower, and the metrics are more meaningful. For Customer B, the models generally have a similar drop in performance on the weekend but retain their performance for the rest of the days.

For Customer C, the performance of the graph-based methods again degrades significantly faster than that of the baseline methods that do not utilize the graph structure. It may be because Customer C has devices grouped partially by location, similar to Customer A. The graph models rely on the graph structure for predictions and cannot cope with changes in the graph structure. As a consequence, the performance of the models degrades quickly as for Customer A (similar pattern visible in Figure 5.10). This claim is fur-

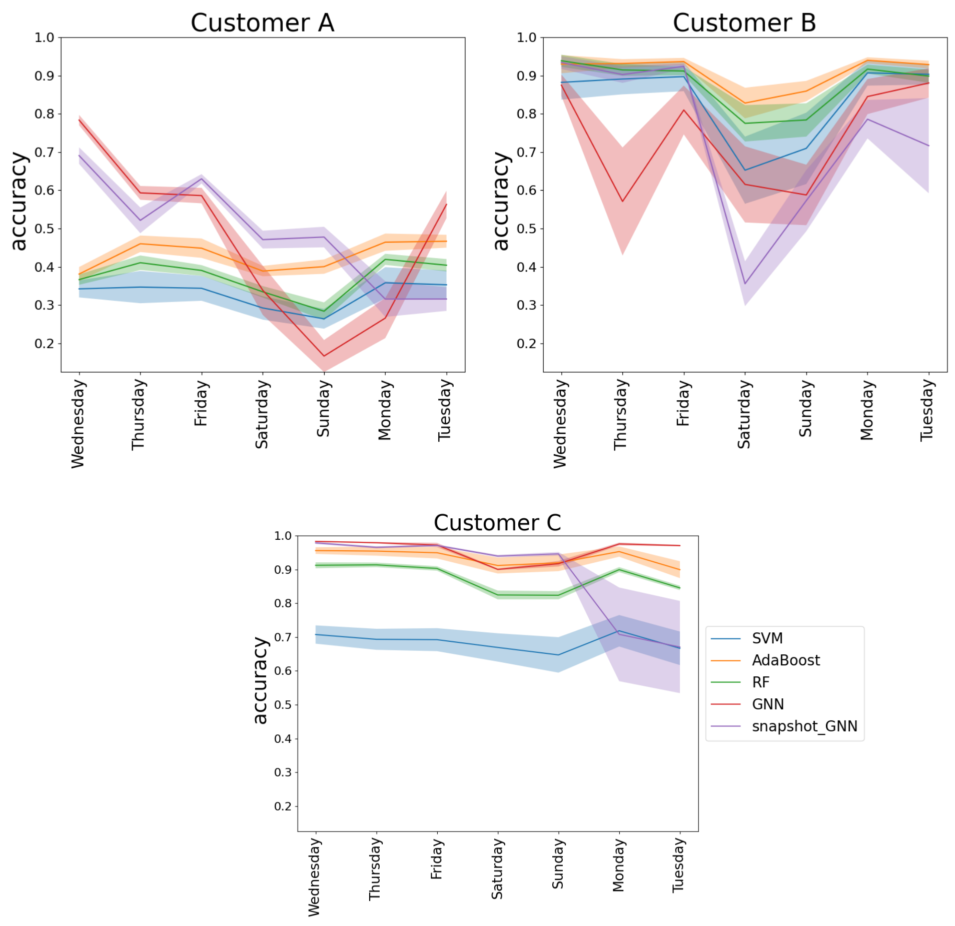


Figure 5.10: Models' accuracy confidence intervals for each customer and method.

ther supported by the graph-based models not having this issue for Customer B, who has devices grouped by function and not location. Consequently, the models do not suffer as much because they do not predict the labels based on the changing graph structure but on the crafted networking features.

Experiment summary

Overall, the presented results indicate that the period the models retain their performance depends on the individual network, the nature of labels, and the individual model. For Customer A, the by-far best graph neural network models leverage the graph structure for predictions, but they do not cope well with the change of the graph structure on consecutive days. For Customer B, the models' performance drops during the weekend but on the rest of the days the models retain their performance. For Customer C, the graph-based models degrade, while the other models retain their performance for a longer period.

This experiment concludes that while the graph-based models are usually

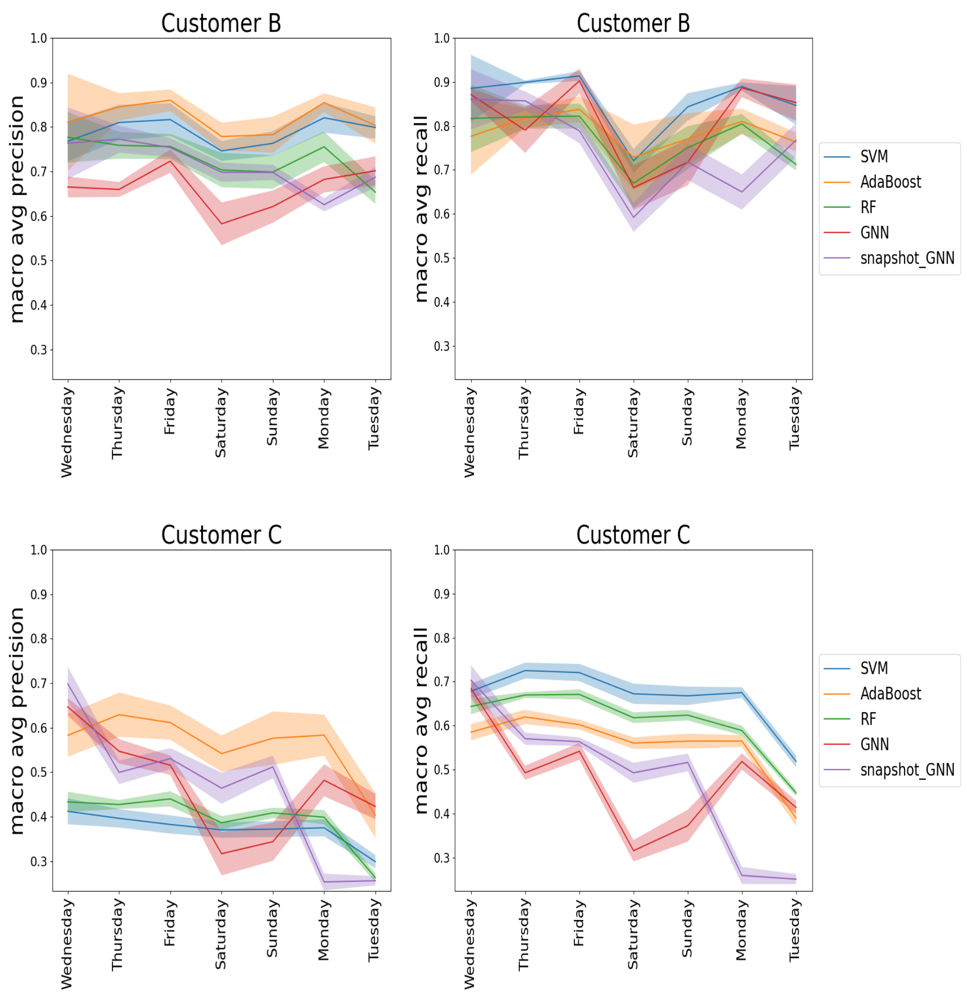


Figure 5.11: Classification recall and precision results for the second experiment.

the best-performing ones for the evaluation on the training day, the graph structure change among the various days of telemetry leads to a degradation of their performance if the graph structure is crucial for the classification task at hand. For Customer B, the graph models retained their performance, which may be because the graph structure or the positional node features are not as relevant for the device classes of this customer. The three baseline models are more stable regarding the prediction metrics during the week and less prone to change in the underlying data structure.

5.4 Models' training resources comparison

This section provides an analysis of the resources each method requires, which considers the retraining requirements and the necessary resources for individual training of each model.

Not every method uses the data in the same format; therefore, preparing the data into the necessary format should also be accounted for when measuring the required time for training. The following section focuses on the data preprocessing for each method.

5.4.1 Data processing pipeline

When evaluating the resources each method requires, it is necessary to consider the entire pipeline processing the raw data for the five models. Figure 5.12 presents a diagram showing the data pipeline for each model.

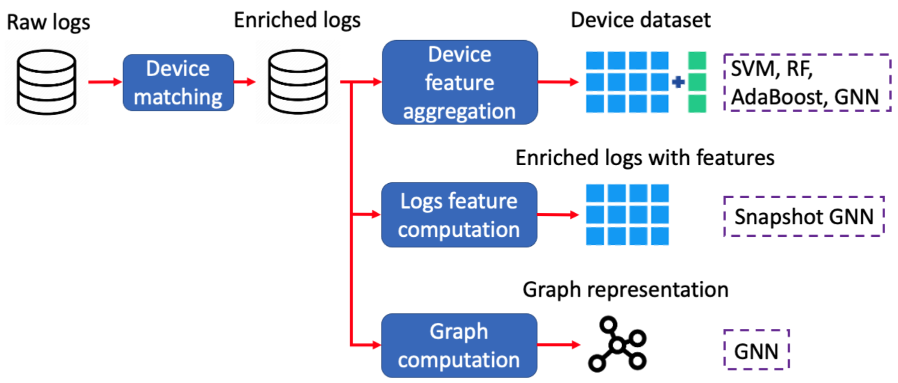


Figure 5.12: Data processing pipeline for each model. The baseline models and the *GNN* model utilize the static device features. The *GNN* model also needs to mine the graph structure from the logs, while the enriched communication logs are processed by the *Snapshot GNN*.

Figure 5.12 shows that each method's data sources vary, and the time to create each resource must be included in comparing the models.

The pipeline architecture is considered in the next section, which presents the time comparison of each method.

5.4.2 End-to-end time comparison of the methods

Before comparing the entire processing times of each method, I would like to revisit how every model was trained to interpret the results correctly. The baseline models (SVM, RF, AdaBoost) were trained with hyperparameter grid search every time due to being computationally lightweight. In contrast, the neural networks were trained with already pre-determined hyperparameters. This puts the baseline models at a disadvantage in the time resources comparison, which was compensated for by the grid search being parallelized on a 32-core CPU - AMD EPYC 7571.

Figure 5.13 displays the standalone training times for all models. The times for graph neural networks are reported for training on the 32-core CPU and on one of the fastest GPUs available for AI inference - NVIDIA T4 Tensor Core GPU.

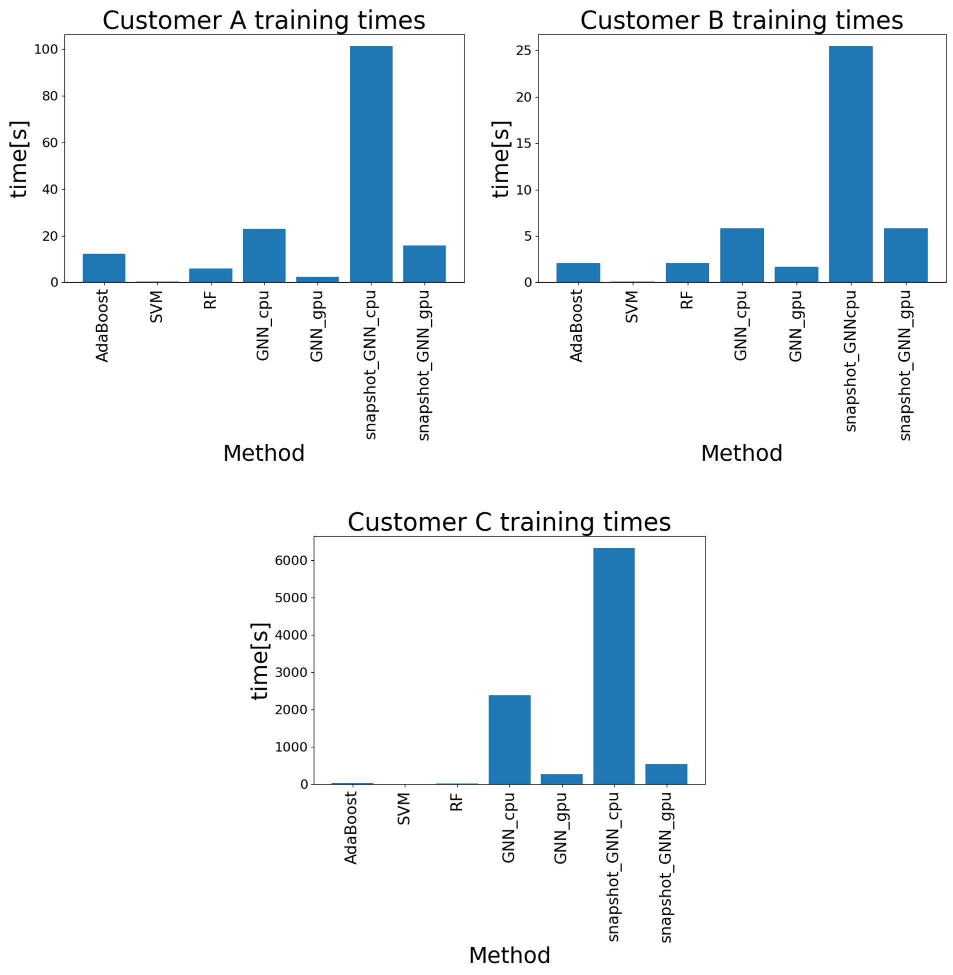


Figure 5.13: Training times of each method for each customer. The training times were averaged over ten runs (except for *Snapshot GNN* on CPU, which had only five repetitions due to long training times) to provide better statistical information.

The GPU training, marked by suffix *_gpu*, helped to significantly reduce the training times for graph neural networks, as shown in the respective figure. The training on the GPU is around five times faster compared to the training on a CPU. Since the GPU training is much faster, I report only the training times measured on the GPU in further plots.

There is a significant difference in resource consumption among the methods, but to obtain a better overall assessment, all of the data processing must be taken into account. Figure 5.14 shows the end-to-end time it takes for each of the methods to process raw logs, create the respective dataset, and train the model.

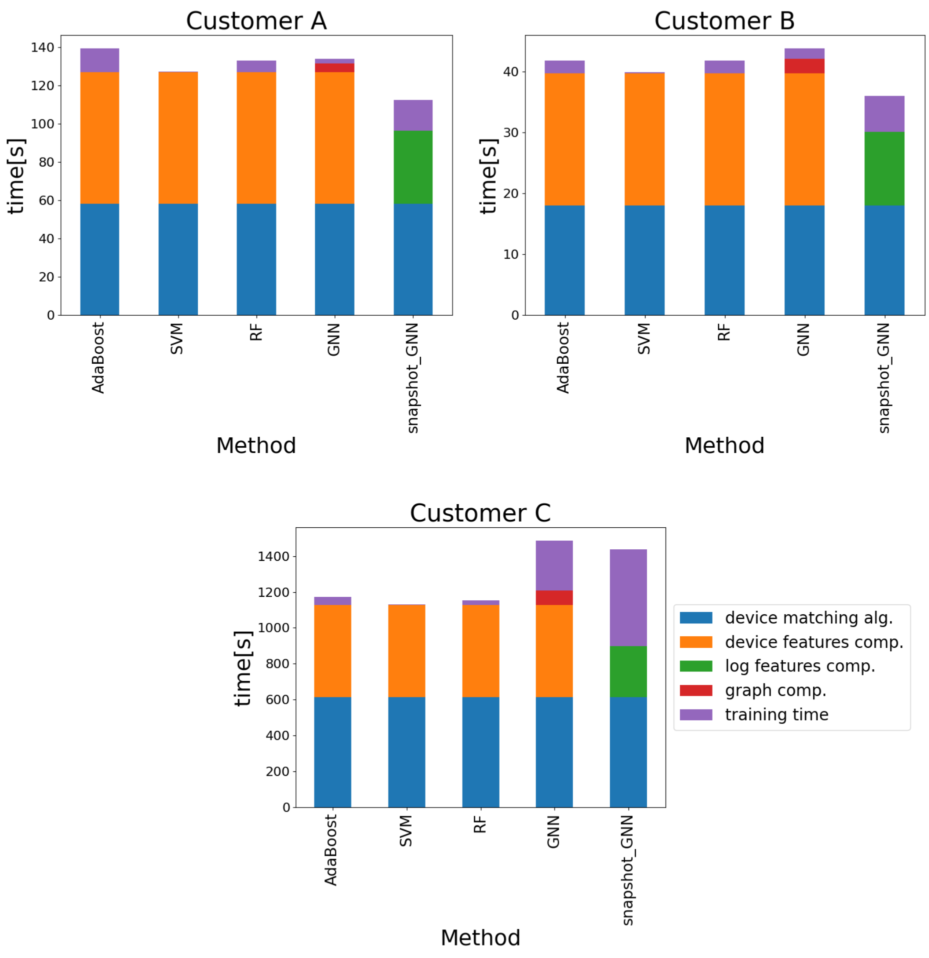


Figure 5.14: End-to-end processing times for each of the methods.

Figure 5.14 puts the time spent by each part of the pipeline from Figure 5.12 into perspective. The blue part of the plots, the device matching algorithm, is a constant specific to the endpoint data. For data where both devices are identified equally in the logs, the device matching procedure would be redundant, and no time would be consumed by it. Due to the scale of the data, all of the data processing methods, except for model training and graph construction, all of the data preprocessing is implemented in Python with the Dask [47] framework, which is a parallel computing platform able to work with large distributed clusters in the cloud.

Overall, most of the time is not spent on actually training the models but rather on processing the data into the respective dataset. The majority of time consumes the device matching algorithm and device features computation (or log features in the case of *Snapshot GNN*). It also shows that while the training time of the *Snapshot GNN* model is longer than for the rest of the models, it benefits from working with enriched communication logs directly with respect to total processing time. This, for instance, results in lower end-to-end processing time than the rest of the methods for two out of three customers while achieving still acceptable processing times even on the largest customer.

The results show that the training time of each method accounts for only a rather small percentage of the overall end-to-end processing times for small and medium customers. For the largest network, the network of Customer C, the *GNN* takes longer due to issues discussed, for example, in [48]. These longer training times could be reduced for example by neighborhood sampling mechanism, employed for instance by GraphSage architecture [4], or a simpler neighborhood aggregating scheme. That would result in smaller computational graphs and faster forward and backward passes during training. The classification results of *Snapshot GNN* model, which utilizes the sampling in a similar form as well (building the graph from the last 30 interactions), provide a solid ground for employing this method to reduce the computational burden for the *GNN* model while retaining the performance.

One of the key takeaways from the presented comparisons is that most of the time is consumed processing the raw logs into the datasets. These datasets must be created for both the training process as well as the inference process. The section analyzing models' performance retention suggests frequently retraining the models as the underlying data change leads to performance loss, especially when the graph structure of the network is crucial for classification. Therefore, if the datasets are already created, retraining the models for the current data would be in general beneficial with respect to the total processing times. Furthermore, the models do not have to be retrained from scratch but only fine-tuned for the current data, reducing the training time significantly.

Chapter 6
Conclusion

My thesis aimed to develop a method for the behavioural classification of network devices working in a semi-supervised manner.

I have performed an extensive study on the state-of-the-art methods that are used for node classification on static graph-structured data and dynamic graph-structured data. Based on my research, I have selected three models that serve as baseline models. Furthermore, I have implemented an advanced graph neural network model that uses the static graph structure for inference. Preliminary experiments with state-of-the-art methods for node classification in dynamic graphs showed that the methods were unsuitable for the task. Therefore, I have also developed a new model for classifying nodes in dynamic networks, the *Snapshot GNN*.

I have leveraged deep domain knowledge to craft meaningful device representations and developed new node positional features that capture the global position of a node in a graph. The additional value of these features was confirmed in the presented ablation study. The device representation was then used to perform thorough experiments with the implemented methods.

The advanced static graph-based model *GNN* and the newly developed method *Snapshot GNN* were the best performers when evaluated on a test dataset from the same day. The *Snapshot GNN* reached a similar classification performance level as the static graph-based neural network while providing the ability to track the devices in the network throughout the day due to working with dynamic graphs. However, the second experiment showed that the change of the underlying graph (when trained models are evaluated on consecutive days) led to more severe performance degradation of the graph-based models. Finally, both the advantages and disadvantages of the implemented methods were analyzed in terms of both the performance and the computational requirements. The analysis showed that frequent retraining would be beneficial to obtain the best performance, even with respect to the training requirements of the best-performing models.

Suggestions for further research are divided into two main directions. The first one is a different approach to constructing the underlying graphs. In this work, all the graphs are unweighted, but a graph where edges are weighted by the frequency of the communication could be utilized. The weighted graph could potentially improve the new positional features and enable using graph neural network architectures accounting for the edge weights. The edge feature information, in general, enables the usage of architectures leveraging both the edge features and the node features in the neighborhood aggregation procedure.

The second direction of further research is eliminating the need for manually assigned labels. Instead of relying on manually assigned labels, unsupervised clustering algorithms could categorize the network behaviour of a device automatically. The algorithms would require a very carefully selected feature set and post-processing steps but could lead to network-agnostic clusters of devices. The clusters would ideally describe the types of devices occurring within the network even without the manually assigned labels, which are hard to obtain and could contain errors.

Bibliography

1. CHANG, Chih-Chung; LIN, Chih-Jen. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*. 2011, vol. 2, no. 3, pp. 1–27.
2. CRIMINISI, Antonio; SHOTTON, Jamie; KONUKOGLU, Ender, et al. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends® in Computer Graphics and Vision*. 2012, vol. 7, no. 2–3, pp. 81–227.
3. FREUND, Yoav; SCHAPIRE, Robert E. A desicion-theoretic generalization of on-line learning and an application to boosting. In: *Computational Learning Theory: Second European Conference, EuroCOLT'95 Barcelona, Spain, March 13–15, 1995 Proceedings 2*. Springer, 1995, pp. 23–37.
4. HAMILTON, Will; YING, Zhitao; LESKOVEC, Jure. Inductive representation learning on large graphs. *Advances in neural information processing systems*. 2017, vol. 30.
5. TANG, Lei; LIU, Huan. Leveraging social media networks for classification. *Data Mining and Knowledge Discovery*. 2011, vol. 23, pp. 447–478.
6. JAYALATCHUMY, D; THAMBIDURAI, P; KADHIRVELU, D. A Novel Approach to Overcome the Limitations of Power Iteration Algorithm Designed for Clustering. In: *Inventive Computation Technologies 4*. Springer, 2020, pp. 774–781.
7. *Benchmarking performance and scaling of python clustering algorithms* [`https://hdbscan.readthedocs.io/en/latest/performance_and_scalability.html`]. [N.d.]. [visited on 2023-03-12].
8. MIKOLOV, Tomas; CHEN, Kai; CORRADO, Greg; DEAN, Jeffrey. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*. 2013.

9. PEROZZI, Bryan; AL-RFOU, Rami; SKIENA, Steven. Deepwalk: Online learning of social representations. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 701–710.
10. TANG, Jian; QU, Meng; WANG, Mingzhe; ZHANG, Ming; YAN, Jun; MEI, Qiaozhu. Line: Large-scale information network embedding. In: *Proceedings of the 24th international conference on world wide web*. 2015, pp. 1067–1077.
11. GROVER, Aditya; LESKOVEC, Jure. node2vec: Scalable feature learning for networks. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864.
12. ANGLADE, Thomas; DENIS, Christophe; BERTHIER, Thierry. *A novel embedding-based framework improving the User and Entity Behavior Analysis*. 2019. Available also from: `https://hal.sorbonne-universite.fr/hal-02316303`. working paper or preprint.
13. KIPF, Thomas N; WELLING, Max. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*. 2016.
14. VELIČKOVIĆ, Petar; CUCURULL, Guillem; CASANOVA, Arantxa; ROMERO, Adriana; LIO, Pietro; BENGIO, Yoshua. Graph attention networks. *arXiv preprint arXiv:1710.10903*. 2017.
15. BRODY, Shaked; ALON, Uri; YAHAV, Eran. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*. 2021.
16. CUI, Hejie; LU, Zijie; LI, Pan; YANG, Carl. On positional and structural node features for graph neural networks on non-attributed graphs. In: *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 2022, pp. 3898–3902.
17. ROSSI, Emanuele; CHAMBERLAIN, Ben; FRASCA, Fabrizio; EYNARD, Davide; MONTI, Federico; BRONSTEIN, Michael. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*. 2020.
18. SANKAR, Aravind; WU, Yanhong; GOU, Liang; ZHANG, Wei; YANG, Hao. Dynamic graph representation learning via self-attention networks. *arXiv preprint arXiv:1812.09430*. 2018.
19. PAREJA, Aldo; DOMENICONI, Giacomo; CHEN, Jie; MA, Tengfei; SUZUMURA, Toyotaro; KANEZASHI, Hiroki; KALER, Tim; SCHARDL, Tao; LEISERSON, Charles. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In: *Proceedings of the AAAI conference on artificial intelligence*. 2020, vol. 34, pp. 5363–5370. No. 04.
20. YU, Bing; LI, Mengzhang; ZHANG, Jiyong; ZHU, Zhanxing. 3d graph convolutional networks with temporal graphs: A spatial information free framework for traffic forecasting. *arXiv preprint arXiv:1903.00919*. 2019.

21. KUMAR, Srijan; ZHANG, Xikun; LESKOVEC, Jure. Predicting dynamic embedding trajectory in temporal interaction networks. In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2019, pp. 1269–1278.

22. TRIVEDI, Rakshit; FARAJTABAR, Mehrdad; BISWAL, Prasenjeet; ZHA, Hongyuan. Dyrep: Learning representations over dynamic graphs. In: *International conference on learning representations*. 2019.

23. XU, Da; RUAN, Chuanwei; KORPEOGLU, Evren; KUMAR, Sushant; ACHAN, Kannan. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962*. 2020.

24. CLAISE, Benoît. *Cisco Systems NetFlow Services Export Version 9* [RFC 3954]. RFC Editor, 2004. Request for Comments, no. 3954. Available from DOI: `10.17487/RFC3954`.

25. MOSKOWITZ, Robert; KARRENBERG, Daniel; REKHTER, Yakov; LEAR, Eliot; GROOT, Geert Jan de. *Address Allocation for Private Internets* [RFC 1918]. RFC Editor, 1996. Request for Comments, no. 1918. Available from DOI: `10.17487/RFC1918`.

26. WING, Dan. Network Address Translation: Extending the Internet Address Space. *IEEE Internet Computing*. 2010, vol. 14, no. 4, pp. 66–70. Available from DOI: `10.1109/MIC.2010.96`.

27. DROMS, Ralph. *Dynamic Host Configuration Protocol* [RFC 2131]. RFC Editor, 1997. Request for Comments, no. 2131. Available from DOI: `10.17487/RFC2131`.

28. SIVANATHAN, Arunan; GHARAKHEILI, Hassan Habibi; LOI, Franco; RADFORD, Adam; WIJENAYAKE, Chamith; VISHWANATH, Arun; SIVARAMAN, Vijay. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing*. 2019, vol. 18, no. 8, pp. 1745–1759. Available from DOI: `10.1109/TMC.2018.2866249`.

29. GORALSKI, Walter. Chapter 11 - User Datagram Protocol. In: GORALSKI, Walter (ed.). *The Illustrated Network (Second Edition)*. Second Edition. Boston: Morgan Kaufmann, 2017, pp. 289–306. ISBN 978-0-12-811027-0. Available from DOI: `https://doi.org/10.1016/B978-0-12-811027-0.00011-4`.

30. COTTON, Michelle; EGGERT, Lars; TOUCH, Dr. Joseph D.; WESTERLUND, Magnus; CHESHIRE, Stuart. *Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry* [RFC 6335]. RFC Editor, 2011. Request for Comments, no. 6335. Available from DOI: `10.17487/RFC6335`.

31. F. DE S. SOARES, Elton; V. CAMPOS, Carlos Alberto; C. DE LUCENA, Sidney. Online travel mode detection method using automated machine learning and feature engineering. *Future Generation Computer Systems*. 2019, vol. 101, pp. 1201–1212. ISSN 0167-739X. Available from DOI: `https://doi.org/10.1016/j.future.2019.07.056`.

32. LIN, Timothy. *Benchmark of popular graph/network packages V2* [`www.timlrx.com/blog/benchmark-of-popular-graph-network-packages-v2`]. 2020. [visited on 2023-04-14].

33. STAUDT, Christian L.; SAZONOVS, Aleksejs; MEYERHENKE, Henning. *NetworKit: A Tool Suite for Large-scale Complex Network Analysis*. arXiv, 2014. Available from DOI: `10.48550/ARXIV.1403.3005`.

34. YOU, Jiaxuan; YING, Rex; LESKOVEC, Jure. Position-aware graph neural networks. In: *International conference on machine learning*. PMLR, 2019, pp. 7134–7143.

35. LI, Pan; WANG, Yanbang; WANG, Hongwei; LESKOVEC, Jure. Distance encoding: Design provably more powerful neural networks for graph representation learning. *Advances in Neural Information Processing Systems*. 2020, vol. 33, pp. 4465–4478.

36. PALMER, Christopher R; GIBBONS, Phillip B; FALOUTSOS, Christos. ANF: A fast and scalable tool for data mining in massive graphs. In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2002, pp. 81–90.

37. PAGE, Lawrence; BRIN, Sergey; MOTWANI, Rajeev; WINOGRAD, Terry. *The PageRank citation ranking: Bringing order to the web*. 1999. Tech. rep. Stanford InfoLab.

38. RIONDATO, Matteo; KORNAROPOULOS, Evgenios M. Fast approximation of betweenness centrality through sampling. In: *Proceedings of the 7th ACM international conference on Web search and data mining*. 2014, pp. 413–422.

39. KATZ, Leo. A new status index derived from sociometric analysis. *Psychometrika*. 1953, vol. 18, no. 1, pp. 39–43.

40. KING, Gary; ZENG, Langche. Logistic regression in rare events data. *Political analysis*. 2001, vol. 9, no. 2, pp. 137–163.

41. VAN RIJN, Jan N; HUTTER, Frank. Hyperparameter importance across datasets. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 2367–2376.

42. FEY, Matthias; LENSSEN, Jan E. Fast Graph Representation Learning with PyTorch Geometric. In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

43. KINGMA, Diederik P; BA, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 2014.

44. LIAW, Richard; LIANG, Eric; NISHIHARA, Robert; MORITZ, Philipp; GONZALEZ, Joseph E; STOICA, Ion. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv preprint arXiv:1807.05118*. 2018.

45. HELMERT, Friedrich Robert. Über die Berechnung des wahrscheinlichen Fehlers aus einer endlichen Anzahl wahrer Beobachtungsfehler. *Z. Math. U. Physik*. 1875, vol. 20, no. 1875, pp. 300–303.

46. HOSSIN, Mohammad; SULAIMAN, Md Nasir. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*. 2015, vol. 5, no. 2, p. 1.

47. ROCKLIN, Matthew. Dask: Parallel computation with blocked algorithms and task scheduling. In: *Proceedings of the 14th python in science conference*. Citeseer, 2015. No. 130-136.

48. ZHANG, Hengrui; YU, Zhongming; DAI, Guohao; HUANG, Guyue; DING, Yufei; XIE, Yuan; WANG, Yu. Understanding gnn computational graph: A coordinated computation, io, and memory perspective. *Proceedings of Machine Learning and Systems*. 2022, vol. 4, pp. 467–484.